Idleness is not sloth

Richard Golding, Peter Bosch,
Carl Staelin, Tim Sullivan, and John Wilkes
Hewlett-Packard Laboratories, Palo Alto, CA

Abstract

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. The commonest 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 hopes that it will be needed later at times of higher utilization, or non-renewable resource like battery power can be conserved by disabling unused resources.

We found opportunities to exploit idle time in our work on storage systems, and after a few attempts to tackle specific instances of it in ad hoc ways, began to investigate general mechanisms that could be applied to this problem. Our results include a taxonomy of idle-time detection algorithms, metrics for evaluating them, and an evaluation of a number of idleness predictors that we generated from our taxonomy.

1. Introduction

Resource usage 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, power conservation in a portable system by shutting down parts of it, or eagerly performing work that might be needed in the future.

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

The overall structure of the idle-detection framework is shown in Figure 1. External work requests arrive and are executed, requiring resources. Potentially useful idle tasks also consume the same resources. An *idleness detector* monitors the external-work arrivals and the state of the server.

When the detector believes the system will be idle enough for long enough, it starts up the idle task. This executes until it completes, or until the detector signals it to stop—typically when new foreground work arrives. The goal of the 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, while making no mistaken predictions of idle periods when the system is not, in fact, idle.

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 benefits from running the idle tasks. These measures use units such as additional 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 useful for guiding the choice of detection mechanism.

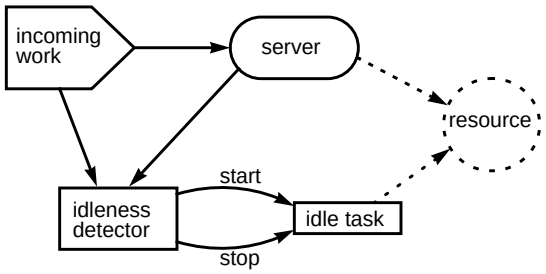

Figure 1: the idle-time processing framework.

The rest of this paper is organized as follows. We begin with a look at several aspects of the problem of making good predictions, including a discussion of external measures and a survey of existing work in this field. We then present an architecture and taxonomy for idleness detectors, and discuss internal measures for evaluating their efficiency. We then use this taxonomy as a tool to generate a suite of detectors, and evaluate their effectiveness under realistic, measured workloads. Some thoughts about opportunities for future work and our conclusions wrap up the paper.

2. Idle-time processing

The purpose of idle-time processing is to improve the system's overall performance during non-idle times. Measuring the improvement requires metrics for the costs and benefits of executing 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. For example:

- The value of powering down a disk drive is the power saved; the cost is that ordinary work may be delayed, extra power consumed during the "recovery" task of spinning up the drive, and the disk lifetime reduced by repeated start-stop cycles.
- The value of disk shuffling (reorganizing data layout on disk) is faster access to frequently-used data. The costs include delaying disk accesses behind a data-move, and buffer-cache pollution.

In addition, the detector itself may intrude on normal system operation: some detectors require significant computation resources to identify idle periods. The work required to determine what to do in the idle time is also a potential resource consumer (for example, disk shuffling is typically based on collected access pattern data; delayed operations are put into a queue that itself consumes resources).

To understand the benefits and costs of idle-time processing, we must first understand how idle tasks can benefit the system, and the ways in which they can create costs: both when executing in an idle period, and if they end up still executing when the system stops being idle. The rest of this section discusses these issues.

2.1. Characterizing idle tasks

Useful idle-time operations fall into a few different categories:

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

2.2. Executing idle tasks

Once it has been determined that an idle task should be executed, a number of operations occur, as illustrated by the time line in Figure 2.

Processing begins when the detector signals the idle task to start executing. (Note that there may be a delay between the end of the last piece of ordinary work and the detector emitting its prediction.) The initial activity

of the idle task may be to run an initialization step that prepares the system for the main idle task that follows. (For example, it may determine which log-structured file system segments are to be cleaned.) This is followed by one or more executions of the idle task proper. Each of these might take a different amount of time or require a different resource. (Breaking the idle task up in this fashion reduces its granularity, which reduces the cost of a bad prediction.)

Eventually, regular work will again enter the system. 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 possible. It may be necessary to execute a recovery task 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.

For eager or delayed activity, one must also consider the steps required to delay work, or to detect and use eagerly-performed work later.

2.3. Detecting idle periods

Recall that the *idleness detector* monitors external work requests in order to find idle periods. Figure 3 illustrates the process for one kind of idleness detector.

At the beginning, the system 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.

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

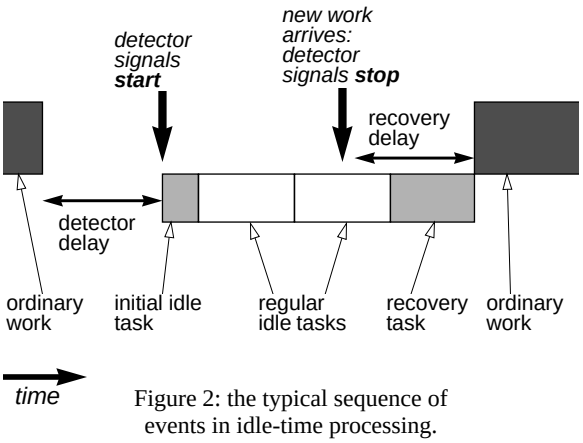

Figure 2: the typical sequence of events in idle-time processing.

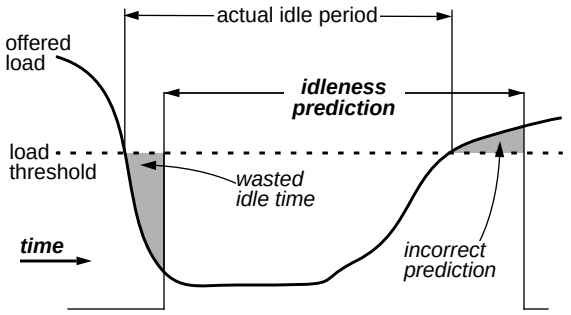

Figure 3: the output from a sample idle detector as the offered load changes.

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.

It is usually good to make conservative predictions since the actual cost of incorrectly starting an idle task is usually higher than the opportunity cost of missing a chance to start it successfully.

2.4. Idle task examples

The model of idle-time processing that we have presented so far is rather abstract. To make matters more concrete, we now introduce three examples of idle-time processing in storage systems.

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 [Baker91b, Carson92a].

This 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* data sectors at a time. (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 here.

This idle task requires no special initialization or recovery actions. A good delayed writeback system minimizes read latency and cache utilization. Table 1 summarizes these characteristics.

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
Measures	max cache space needed change in read latency

Table 1: characteristics of delayed writeback

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 first is best done with an interruptible cleaner that can discard partially-completed operations. Table 2 summarizes these characteristics.

Disk power-down

Several people have investigated powering down disk drives on portable computers to conserve power (e.g., [Cáceres93, Douglis94, Greenawalt94, Marsh93, Wilkes92b]). Using the taxonomy we have developed, the "idle task" is keeping the disk powered off; this is an "optional" task, whose goal is to decrease power consumption; the initial task is to spin the disk down; the recovery task is to spin it back up (Table 3).

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
Measures	foreground cleaning time change in read latency

Table 2: characteristics of LFS segment cleaning

Initial idle task	spin down disk
Idle task	(none): saves 1.5–1.7W
Recovery idle task	spin up disk: takes 1.5s
Granularity	unit: can be aborted at any time loss: power cost of spinning up disk (3.3J) resource: excludes any other disk accesses
Measures	power saved delay caused to I/O operations

Table 3: characteristics of disk power-down

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 power. When a disk I/O request arrives, the disk must be powered up, which uses 2.2W for 1.5s (i.e., 3.3J). Power consumption will decrease if the savings from the powered-down mode outweigh the power cost of spinning it up again. For this disk, it can be achieved if the disk is spun down for as little as 2.2s; 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 mechanism will balance these conflicts.

Other examples

There are many possible uses for idle time beyond the three that we have mentioned. Storage, compilation, user interfaces, and distributed systems all exhibit highly variable workloads—a clue that a system could benefit from idle-time processing. Here are a few examples.

Large data structures such as database indices can often be updated quickly at the cost of gradually-worsening access characteristics. For example, nodes can be inserted into or deleted from a binary tree, unbalancing its branches. However, the initial performance can be recovered by a reorganization step (in this case, rebalancing the tree). Such reorganization

tasks are candidates for idle-time processing, thereby moving the cost of rebalancing out of the critical path for updates.

Likewise, some indexing systems in databases segregate the index into two parts: one for "stable" information, for which an efficient hash or tree structure has been built; and one for "unstable" information that has been added or modified recently [Herrin91]. The unstable portion is periodically merged into the stable portion, which involves significant calculation. Doing this in an idle task can make it appear as if it happened "for free".

Compilation is a particularly rich source of idle-time work. One approach that has been explored is to use an eager make facility [Bubenik89], which detects file changes and rebuilds binaries as soon as possible, in the hope that this work will reduce the latency of performing a make when it is asked for later. Costs here include both the processing and disk-resource contention generated by the background compilations, and also the complexity of hiding intermediate results (which may fail) from the user.

Languages such as Cecil, Self, and Smalltalk involve dynamic code generation [Chambers90a]. In many cases this code can be optimized using information generated at run-time, such as the probability of taking a particular branch of a conditional or the inferred type of an object. Modules can be re-optimized when the processor is not busy with normal work during idle time.

There are many instances of distributed system load-balancing schemes that look for "idle workstations" and attempt to use the large number of spare cycles that can be claimed in this way. To keep the workstation owners' happy (and thus the workstations eligible for being used in this fashion), good predictions of when the users are idle are needed [Litzkow88]. Designers of such schemes have used process migration as a mechanism for avoiding loss of work when an idle task is preempted [Litzkow88].

More closely-coupled systems can perform eager data transmission in periods when the network is lightly loaded. For example, release consistency [Carter91] mechanisms can pre-transmit updates to reduce the time to complete a release, which is often on the critical performance path.

The choice of whether to compress data for network transmission can depend on the utilization of both the network and the processor: if the processor is busy, but the network is not, it may make sense to transmit data uncompressed. Likewise, the rate or quality at which multimedia data are played can be varied to take advantage of extra network or processor resource beyond a guaranteed minimum. These are examples of policy changes as a result of idleness detection.

Forwarding-address chains in a distributed system allow clients to find the current location of a moving object [Shapiro92]. When the object moves, it leaves a pointer to its new location at its old one. Locating the object is done by starting at a place that the object has visited, and following the chain of pointers. Forwarding address chains can be compressed when the system is idle, thereby reducing the time to do future locations.

Disk and file readahead is a common example of eager activity in a storage system [McVoy91, Ruemmler94]. Transparent file compression can improve effective storage capacity, and compressing during idle periods ensures that it does not interfere with ordinary operations. Disk shuffling [McDonald89, Vongsathorn90, Ruemmler91, Akyürek93] involves rearranging the blocks or cylinders on a disk so that frequently-accessed information is located near the middle of the disk with related data close together. Shuffling has non-trivial initial and recovery tasks.

An anonymous reviewer of this paper suggested using idle cycles to perform graphical window-system operations eagerly. For example, hidden or obscured portions of windows could be rendered in idle periods so that exposing them would be quick.

In summary, the existence of burstiness and under-utilized resources in a system is an indication that some form of idle-time processing could be exploited. The opportunities are limited mostly by the complexity of deciding what to do when, and by the potential downside of resource contention with foreground activities. Dealing with both of these requires efficient, accurate idleness detectors, whose construction is the subject of the next section.

3. An idle-detector 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 problem 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 4 shows the overall scheme. An idleness detector is composed of a number of predictors and skeptics, along with an actuator:

- A *predictor* monitors its environment—the arrival process, the server, 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 or DAG with predictors as the leaves and skeptics as the internal nodes. Streams of predictions flow toward the root node, which emits predictions to the actuator.

3.1. Predictors

As with so many problems of this type, optimal idleness detection requires off-line analysis. This is not as useless as it might seem: if usage patterns are stable enough, then a one-time off-line analysis may provide an excellent prediction. For example, "weekends from 1–6 a.m." is a common time to perform system maintenance.

In practice, on-line detectors are of more interest. They can be classified according to the approach used to predict the idle period start time and its duration.

3.1.1. Idle period start time

Simple predictors use little information to make their predictions; more sophisticated ones take advantage of knowledge about the arrival process to make better decisions. We present them here in roughly increasing-complexity order.

- *Timer-based:* whenever the system runs out of work, the predictor begins a timer. If no work comes in before the timer expires, the predictor 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.
- *Rate-based:* the predictor 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", thereby providing some hysteresis. Methods for maintaining the estimate include:
  - *moving average.* The rate is periodically sampled, and the predictor computes a moving average of the samples.
  - *event window.* The predictor 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 rate-based:* like rate-based, but with an arrival-rate load threshold that is adapted according to the quality of the predictions.
- *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 periodic work, a digital phase locked loop or DPLL [Massalin89a, Lindsey81] can be used to synchronize predictions to periodic events in the workload, such as the UNIX syncer daemon. By knowing when work will arrive, it is also known when the system is idle.

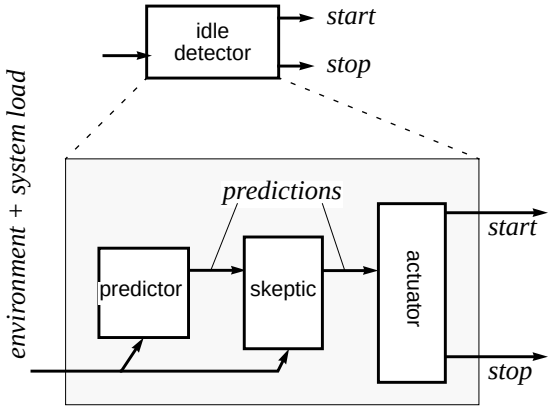

Figure 4: construction of a simple idle detector

3.1.2. Idle period duration

Duration predictors can use a wide range of techniques to adapt 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 include schemes that try to predict the end of an idle period as this happens, rather than at its beginning. These are most useful when the definition of "idle" allows some residual foreground work. For example: rate-based; adaptive rate-based; or rate-change-based.

- *Fixed duration:* a fixed duration is predicted. The simplest form of this is "enough time to run the idle task once."
- *Moving average:* the predictor 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.
- *Filtered moving average:* as for moving average, but only idle periods greater than some lower-bound are considered during the averaging process.
- *Backoff:* after each prediction is used, the 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 an idleness predictor works to keep its predictions slightly shorter.[1]

  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 schemes 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.

[1] We describe backoff algorithms with shorthand of the form Arith+/Geom-: this indicates a predictor with arithmetic increases and a geometric decrease policy.

3.2. Skeptics

A skeptic takes in one or more prediction streams, and emits a new one. They 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 using a moving average of the measures we propose in Section 4, and only pass along predictions when the accuracy is above some threshold.
- *environmental:* discards or modifies predictions according to some external event (e.g., 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 [CesaBianchi94]. This approach has been shown to yield near-optimal prediction in many cases.

4. Evaluating idleness detectors

A good idleness detector produces a stream of predictions that waste little time at either the beginning or end of an idle period, while rarely making a prediction that starts too early, ends too late, or overlaps much real work. We now describe several measures used to quantify how well a particular detector meets these goals. Remember that their

interpretation depends on an application-specified level of acceptable idleness.

We start with several primitive measures:

- The *predicted* time is the total amount of time a detector declared 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 idle when the system was not idle.
- The *violations* count the number of operations that overlapped the declared idle periods.
- The *delay* is the sum of the delay imposed on operations, assuming that any operation that starts during a period declared idle must be delayed until the end of the period.
- The *intrusiveness* of an idleness detector measures how much extra load the detector itself imposes on a non-idle system.

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

  $efficiency = predicted / actual$

- 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. Ideally, we'd like to have a single value that give the overall goodness of a detector. However, this is impossible, because the weighting of the value of idle task processing and the costs of incorrect idleness predictions is highly application-specific, and sometimes even subjective at a personal level. Nonetheless, we found it useful to consider three main candidate *figures of merit:*

1. $merit_1 = efficiency \times (1 - incompetence)$

   This merit figure considers only efficiency and incompetence, penalizing efficient detectors that are also incompetent.

2. $merit_2 = efficiency / violations$

   This merit figure penalizes a detector heavily for each operation that occurs during a predicted idle period.

3. $merit_3 = efficiency - \alpha \times delay$

   This merit figure penalizes a detector for the total time that operations are delayed. The relative importance of delay and efficiency can be scaled by the factor α.

The choice is a function of the application. When spinning down a disk, for example, the power saved is related to detector efficiency, while the cost is the delay to operations. The high subjective impact of the cost suggests a metric that penalizes delays more, such as $merit_3$. Delayed writes, on the other hand, have a less dramatic cost of violations, so $merit_1$ is probably more appropriate.

5. Performance results

To get quantitative measures of the effectiveness of idle-time processing, we used the taxonomy presented in section 3 to design a large number of possible idleness predictor networks. We then implemented the component parts and evaluated how well the networks did in practice. This section reports on what we learned.

5.1. Implementation

We implemented the detectors and idle tasks in the TickerTAIP simulation system [Ruemmler94, Golding94]. 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 I/O access traces taken from real systems [Ruemmler93] to avoid making simplifying assumptions about access rates. We used 1week subsets of these traces.

We added a few new component classes to the TickerTAIP system. These included an overall framework; multiplexers to distribute events to multiple predictors; the predictors themselves; skeptics; and actuators. Our disk and device driver models were changed to accept detectors. The idleness detectors were connected to the disk devices for taking internal measurements, and to the disk device drivers for external measures.

To perform the evaluations, we introduced the notion of a *busyness detector* to determine when the system should be considered busy (i.e., not idle). This was used to evaluate the idle detectors. Busyness detectors can be application-specific. In practice, the simplest busyness detector was the one used for the results we report here: it declared the system busy if there was at least one request anywhere in it.

We looked at several hundred idle detector networks derived from our taxonomy, evaluating them against our internal measures. We also implemented three skeptics: one that filters out predictions during the busiest six hours of the day (the TODSkeptic); one that filters out predictions from a predictor that has a high rate of violations (MeritSkeptic); and one that

combines predictions from several predictors, outputting only predictions on which a quorum agree (QuorumSkeptic).

5.2. Start-time predictors

To make the presentation of the results less intractable, we picked a fixed duration prediction of 25s and combined this with different start-time prediction algorithms.[2] The most efficient detectors under this test are:

- Event window, when busy 10% or less over last 5 operations (efficiency = 1.01)
- A 1s fixed timer (1.01)
- Event window with rate <10 IO/s over last 5–25 operations (1.00–0.998)
- Moving average rate of 10 IOs/s or less (0.993)
- Adaptive timer with Geom+/Arith- or vice versa with 100ms increments (0.990)
- Event window, busy 1% or less over last 5 operations (0.985)

The lowest violation rates (violations per second of predicted idle period) come from:

- Event window, rate <0.1 IO/s over 25 operations
- Same, over last 5 operations
- Moving average, 0.1 IO/s or less
- Adaptive timer Arith+/Arith-, 10s increment
- Adaptive timer Arith+/Geom-, 10s increment
- 10 second fixed timer

5.3. Duration prediction

The next step is to fix the start-time predictor (we used a moving-average rate-based start-time predictor with a threshold rate of 10 IO/s because that gave pretty high efficiencies). The most efficient predictors under this test are:

- Fixed 25s period (efficiency = 0.993)
- Backoff using Arith+/Arith- in 10ms, 100ms and 1s increments (0.992, 0.990 and 0.988 respectively)
- Backoff using Arith+/Geom- and a 1s increment (0.975)
- A moving average over the most recent durations (0.971)
- Backoff Arith+/Geom- with 100ms increments (0.959), or 10ms increments (0.945)
- Moving average + TOD skeptic (0.759)

[2] Because 25s is a little shorter than the 30s sync daemon interval on the systems from which the traces were taken.

From the point of view of violation rates, the following were the best ones:

- Moving average + violation rate skeptic (violation rate = 0.358s)
- Moving average + TOD skeptic (0.722)
- Moving average (0.821)
- Backoff Arith+/Geom- with increments of 10ms (1.02); 100ms (3.61); or 1s (14.6)
- Fixed 25s timer (35.1)
- Backoff Arith+/Arith- with increments of 10ms (36.9), 100ms (44.6), or 1s (48.0).

In some circumstances (such as log-structured file system cleaning), it's useful to have idleness detectors that can reliably predict idle periods of long duration. Under this metric (the mean duration of the idle period predicted), we find the following are best:

- Backoff Arith+/Arith- with increments of 1s (mean duration = 44.8s), 100ms (40.4), or 10ms (34.4)
- Fixed 25s timer
- Backoff Arith+/Geom- with increments of 1s (17.9); 100ms (5.82); or 10ms (2.19)
- Average duration + the merit skeptic (0.982)
- Average duration alone (0.952)
- Average duration + the TOD skeptic (0.944)

5.4. Other combinations

Needless to say, picking the best stand-alone start-time and duration prediction algorithms and combining them doesn't produce the best overall result. Instead, we found (by exhaustive combinatorial analysis!) that the following are among the best pair-wise combinations for efficiency:

- EventWindow, busy 10% or less over 25 events + fixed 25s duration (efficiency = 1.0098)
- The same algorithm over the last 5 events (1.0095)
- Busy 10% or less over 25 events + fixed 25s timer (1.0087)
- Busy 10% or less over 5 events + Backoff Arith+/Arith- with 10ms increment (1.0087)
- Busy 10% or less over 25 events + Backoff Arith+/Arith- with 10ms increment (1.0087)
- Fixed 1s timer, + fixed 25s duration
- Busy 10% or less over 25 events + Backoff Arith+/Arith- with 100 ms increments

Generally, the results were dominated by Event Window predictors with a threshold of busy 10% or less, or 5 IO/s or less, combined with arithmetic backoff. For durations, the best result came from combining these with adaptive timers with 100ms

arithmetic increase and geometric decrease (or vice versa), or by backoff predictors with 10–100ms increment arithmetic increases and geometric decrease.

The best violation rates results came from:

- Adaptive timer Geom+/Geom- + Average duration (violation rate = 0.214/s)
- Moving average 10 IO/s + Average duration + a violation rate skeptic (0.358)
- Event window over 25 events with rate < 0.1 IO/s + Average duration (0.362)
- Event window with rate < 10 IO/s and a moving average, fed into a Quorum skeptic (0.373)
- Event window over last 5 events with a rate < 10 IO/s, + Average duration, fed into a violation rate skeptic (0.431)

Generally, rate-based or adaptive Arith+/Arith- timers with 10s increments combined with Average duration predictors or Backoff Arith+/Geom- and 10 ms increments dominate.

Looking at mean duration predictions, we find the longest values come from:

- Adaptive timers, Arith+/Geom- or vice versa, with Backoff Arith+/Arith- (1s increment) duration (mean duration = 340s).
- Adaptive timer Arith+/Arith- and increments of 1s (306s), 10s (288s) or 100ms (256s) + Backoff Arith+/Arith- with 1s increments.

We learned that there can sometimes be unfortunate interactions between adaptive start-time predictors and their duration counterparts. These feedback loops are best minimized by running the start-time prediction algorithm once per actual idle period, and the duration prediction once per predicted idle period, if this is shorter than the actual idle period, but longer than some minimum threshold. (The first prevents inter-policy conflicts; the second allows duration-predictors to respond gracefully to alternating bursts of high activity and long quiet periods.)

5.5. Skeptics

Skeptics proved more important for busy disks, where the workload is more variable than on quiet disks. They are also best at reducing the mean violation rate.

On a busy disk, the TOD skeptic reduces violation rate; in the samples, usually to about half what the same predictor yielded without the skeptic, except when the rate was already low. This generally causes a loss of about 20–30% efficiency, indicating that much of the time that could successfully be declared idle was not in the busy hours of the day anyway.

The Merit skeptic likewise reduces the violation rate. For predictors with a high violation rate, the Merit skeptic reduces violations to less than half of the rate of using the TOD skeptic, but generally at the cost of about half the efficiency. When the violation rate is already low, the Merit skeptic does not give appreciable advantage over the TOD skeptic, and in one case (MovingAverage 10/s, Average duration) increased the violation rate by about 20%

For the Quorum Skeptic, we combined EventWindow and MovingAverage predictors (both at rates < 10/s); the EventWindow and a PLL; and the Moving Average and a PLL. For these combinations, both predictors had to agree on idleness. Using quorums like this leads to short predictions because they all have to agree. We also combined four predictors using a quorum size of two for agreement: two EventWindows with different queue sizes, the EventWindow, and a PLL.

The combination of an event window and a moving average predictor them agree gives a low violation rate (0.23 on a busy disk). The combination of all four gives 99% efficiency and one of the lowest violation rates.

Overall, our top three picks for workloads such as delayed writeback and segment cleaning are:

- EventWindow 10 IO/s + MovingAverage 10 IO/s + Quorum skeptic (both agreeing); gives 0.98 efficiency, 0.23 violations, 0.52s mean duration.
- Four predictors + Quorum skeptic; gives 0.99 efficiency, 0.48 violations, 0.52s mean duration.
- Moving average start-rate 10/s + average duration; gives 0.98 efficiency, 0.49 violations, 1.05s mean duration.

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 several specialized domains.

The major contribution of this work is to put the previous approaches into a common framework. In turn, the framework suggested opportunities for combining analysis techniques to improve the quality of idle-time prediction. It also allowed us to propose some figures-of-merit for evaluating idle-time detection algorithms, which we then used to evaluate many different idle-time detectors.

Our framework should be helpful to those looking to exploit low-utilization periods in computer 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 were gratified to learn that simple predictors work remarkably well. This is good news: it means that these techniques can be applied successfully in the real world with only moderate effort. Nonetheless, we found that you can't have all three of high efficiency, low violation and long durations, but you can generally get any two.

Acknowledgments

George Neville-Neil suggested predictors based on the first derivative of arrival rate. Fred Douglis's interest in disk power conservation spurred much of our early thinking.

Availability

An extended version of this paper, with more detailed results than we had space for here, is available by anonymous ftp from the directory pub/wilkes on the system ftp.hpl.hp.com.

References

[Akyürek93] Sedat Akyürek and Kenneth Salem. *Adaptive block rearrangement*. Technical report CS–TR–2854.1. University of Maryland, November 1993.

[Baker91b] 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.

[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, May 1992.

[Carter91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, CA). Published as *Operating Systems Review*, **25**(5):152–64, 13–16 October 1991.

[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.

[Chambers90a] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In Urs Hoelzle, editor, *The Self Papers*. The Self Group, CIS 209, Stanford University, Stanford CA 94305, 1990.

[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.

[Douglis87] Fred Douglis and John Ousterhout. Process migration in the Sprite operating system. *Proceedings of 7th International Conference on Distributed Computing Systems* (Berlin, 21–25 September, 1987), pages 18–25, R. Popescu-Zeletin, G. Le Lann, and K. H. Kim, editors. IEEE Computer Society Press, 1987.

[Douglis94] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. *Proceedings of USENIX Winter 1994 Technical Conference* (San Francisco, CA), pages 292–306. USENIX Association, Berkeley, CA, 17–21 January 1994.

[Golding94] Richard Golding, Carl Staelin, Tim Sullivan, and John Wilkes. "Tcl cures 98.3%of all known simulation configuration problems" claims astonished researcher! *Proceedings of Tcl/Tk Workshop, New Orleans, LA*, June 1994. Available as Technical report HPL–CCD–94–11, Concurrent Computing Department, Hewlett-Packard Laboratories, Palo Alto, CA.

[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 62–66. IEEE Computer Society Press, 31 January–2 February 1994.

[Herrin91] Eric H. Herrin II and Raphael A. Finkel. *An implementation of service rebalancing*. Technical report 191–91. University of Kentucky, Department of Math Sciences, July 1991. *Proc. of the XI Intl. Conf. of the Chilean Computer Science Society*, 1991.

[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.

[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.

[Litzkow88] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor—a hunter of idle workstations. *Proceedings of 8th International Conference on Distributed Computing Systems* (San Jose, CA), pages 104–11. IEEE Computer Society Press, 13–17 June 1988.

[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.

[McDonald89] M. Shane McDonald and Richard B. Bunt. Improving file system performance by dynamically restructuring disk space. *Proceedings of Phoenix Conference on Computer and Cmm.* (Scottsdale, AZ), pages 264–9. IEEE, 22–24 March 1989.

[McVoy91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 33–43, 21–25 January 1991.

[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, 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.

[Ruemmler91] Chris Ruemmler and John Wilkes. *Disk shuffling*. Technical report HPL–91–156. Hewlett-Packard Laboratories, October 1991.

[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), pages 307–26, January 1993.

[Shapiro92] Marc Shapiro, Peter Dickman, and David Plainfossé. *SSP chains: robust, distributed references supporting acyclic garbage collection*. Technical report 1799. INRIA, France, November 1992.

[Vongsathorn90] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, **20**(3):225–42, March 1990.

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

Author information

Richard Golding received his M.S. degree in 1991 and his Ph.D. degree in 1992, both in Computer and Information Sciences from the University of California, Santa Cruz. Following a stint at the Vrije Universiteit Amsterdam, he joined Hewlett-Packard Laboratories as a researcher. His research interests include distributed operating systems, wide-area networking, and storage systems.

Peter Bosch graduated with degrees in Electrical Engineering (1988), and Computer Science (M.Sc., 1994) from the University of Twente, Netherlands. He has worked since 1991 for the University of Twente as an research assistant. His current work involves research into file systems and implementation of a high performance file system for the ESPRIT Pegasus project. His hobbies are traveling and spending evenings in nice restaurants.

Carl Staelin received his Ph.D. in Computer Science from Princeton University in 1992. He has worked since 1992 as a researcher for Hewlett-Packard Laboratories. As part HP's Berkeley Science Center he is currently working on the NOW and Mariposa projects at U.C. Berkeley. His current research interests include {high-performance, tertiary, distributed, reliable} storage systems, distributed systems, and electronic libraries. For fun he is also working to provide a library of binary packages of public domain software for HP workstations.

Tim Sullivan has been a member of the technical staff at Hewlett-Packard Laboratories since 1984, when he was an undergraduate at Stanford University. He worked on software, workload characterization, and performance modeling tools for experimental I/O devices while obtaining his BS (1985) and MS (1988) degrees in Electrical Engineering. He spent two years at the Hewlett-Packard Laboratories Pisa Science Center in Pisa, Italy designing parallel languages with researchers at the University of Pisa. He has also done research on operating systems and databases for distributed and parallel systems and on high-performance, high-availability storage systems.

John Wilkes graduated with degrees in Physics (BA 1978, MA 1980), and a Diploma (1979) and Ph.D. (1984) in Computer Science from the University of Cambridge. He has worked since 1982 as a project manager and researcher at Hewlett-Packard Laboratories in Palo Alto, CA. His current research area is high-performance, high-availability storage systems, and he has interests in multicomputer interconnects, operating system design, and performance modelling. He also enjoys learning about Renaissance art and architecture and interacting with the academic research community.

The authors can be contacted through Richard Golding: golding@hpl.hp.com, Hewlett-Packard Laboratories, mailstop 1U13, PO Box 10490, Palo Alto, CA 94303–0969.