7-29-2004

An Adaptive Mechanism for Improving File Transfer Performance

Eric Moss
University of Nebraska

Leen-Kiat Soh
University of Nebraska, lsoh2@unl.edu

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2004-0012
Issued 7/29/2004

An Adaptive Mechanism for Improving File Transfer Performance

Eric Moss, Leen Kiat Soh

May 7, 2004

Abstract

A variant of instance-based learning is described which detects periodic patterns in the presence of sparse data. A weighted average gives higher weight to values in the recent past as well as those at expected periods in the past. After each new measurement, the space of weights near the current set is searched for a set that mimimizes the error of the prediction, thus providing a learning mechanism. The method is described in terms of an application which minimizes file download times by choosing between available servers.

Keywords: feature detection, sparse data, instance-based learning.

1 Introduction

Performance of certain tasks, such as file downloads in the presence of a choice of servers, can be improved over time if a history of performance is analyzed for patterns. For example, when choosing a server from which to download, it may be that one server is thousands of miles and dozens of hops away, while another is just across town. This knowledge alone is not enough to predict the better performer, though, as the closest server may have a very slow network connection. It may also be that the user is downloading during the daily backup or the monthly maintenance period of either machine. [15, 9]

In this paper, we describe a learning mechanism for predicting from past performance what server would be the best choice given parameters such as time of day and file size. Test code was written entirely in Common Lisp using the CLISP and CMUCL implementations. [4]

2 Motivation

The FreeBSD operating system installs software from source code using the "Ports" system. Ports fetches source files from a list of some 100 ftp and http servers called "MASTER_SITES". For each file, servers are checked in order until one is found containing the file. The file is fetched and the algorithm moves

on to the next file, repeating the search through the entire MASTER_SITES list. MASTER_SITES is statically ordered, with the main FreeBSD site last, the goal being to fetch from less-used servers before overwhelming the main site. [12]

This design is inefficient. Because it does not account for network performance at a given user location in the INTERNET, some users are saddled with slow downloads. Manually re-ordering the MASTER_SITES list ignores the dynamic nature of the INTERNET, making that solution brittle. Random ordering has been introduced to FreeBSD, but its main function is to not overly favor users in the continental United States. In nations where server domains end in country codes (e.g. '.jp' for Japan), ordering by country code is useful. Most domains do not follow such conventions, however, and it falls short in border regions of large nations such as Australia and China, where the nearest server is in another nation.

When only a slow internet connection is available, users wanting to download large Ports (e.g. the Mozilla web browser) often find themselves waiting for hours to get the files. Not only is this frustrating, but poor phone lines often mean that connections die before the file is completely downloaded, forcing a retry from the beginning. Thus, if better server choices can be made, the user will benefit doubly. Moreover, to whatever extent download speed is correlated to (hop-wise) nearness of the chosen download server, choosing the fastest server will benefit the entire Internet by localizing traffic.

In an initial version of the system, we ordered the download servers by increasing *ping* times. Although this is a form of prediction of future performance, it has several drawbacks. Many servers do not respond to *ping* messages, and so are not ranked. Even when *ping*able, the times are not necessarily representative of real download performance. *Ping* packets are given lower transmission priority than normal packets, and even with large data payloads, do not exercise the TCP congestion window mechanisms as do real, large downloads using thousands of packets.

However, even though the *ping*-based ranking of server speeds is inaccurate and incomplete, it *does* tend to rank the slowest sites as being slow. Figure 1 shows the relative performance of 22 servers from which the same 1.1 MByte file was downloaded over a modem connection. The servers are ordered by decreasing predicted speed. Question marks denote servers that were not *ping*able. The results show that while *ping*-based ranking does not necessarily choose the fastest server to use, it rarely chooses from among the slowest. As a result, average download times are cut by some 20%.

Comparison of estimates and actual download speeds shows that what is needed most is a version of *ping* that accurately measures performance. Then the list of possible servers could be quickly *ping*ed and sorted according to actual performance *now*. In the absence of such a tool, we are forced to incorporate feedback of actual download performance in order to make reasonable predictions and adapt to changes in performance.

Site Name (for wget-1.8.2.tar.gz)	Est. Rate (bytes/sec)	Actual Time (sec)	Actual Rate (kbytes/sec)
ftp.wustl.edu	5742	228.7	4.93
mirrors.usc.edu	5554	220.9	5.10
gatekeeper.dec.com	5170	222.0	5.08
ftp.FreeBSD.org	4688	228.6	4.93
ftp.de.uu.net	4400	236.5	4.77
ftp.informatik.rwth-aachen.de	4390	228.2	4.94
ftp.leo.org	4368	253.9	4.44
ftp.tuwien.ac.at	4142	239.2	4.71
ftp.lip6.fr	3992	228.0	4.95
ftp.mirror.ac.uk	3932	258.5	4.36
ftp.informatik.hu-berlin.de	3922	237.4	4.75
ftp.funet.fi	3880	228.1	4.94
ftp.chg.ru	3432	697.4	1.62
ftp.dti.ad.jp	3338	234.0	4.82
www.t.ring.gr.jp	3294	224.9	5.01
ftp.gnu.org	?	241.8	4.66
ftp.kddlabs.co.jp	?	225.7	5.00

Figure 1: An ordering of sites based on ping times.

Thus, the goal of the system pursued in this paper is to provide a feedback mechanism for learning which servers provide the best performance. The system should analyze the performance history of every server, choose the best among them, and correct its model of each server's performance history as new measurements are taken.

3 Methodology

Figure 2 depicts the feedback model chosen. The process begins at the leftmost box, with the user choosing to build a given FreeBSD Port. The Ports system specifies a list of servers from which a given Port may be downloaded. The server names and file size are used to select the performance histories of interest. The predictor assumes a model of download performance containing possible periodic components. The parameters of this model are used to predict performance for time *now*, where *now* is the time a new download is expected to commence.

Having made performance predictions for each of the candidate servers, the list of servers is sorted, in fastest-first order. The file is fetched from the best choice, and the actual download speed is measured. This download speed is then used in a feedback loop to adjust the model parameters. This adjustment is intended to track changes in actual internet performance and to better model each server's performance as more data is gathered.

One factor that is insurmountable is that servers which start out with poor predictions are unlikely to be used, and thus their feedback loop is effectively

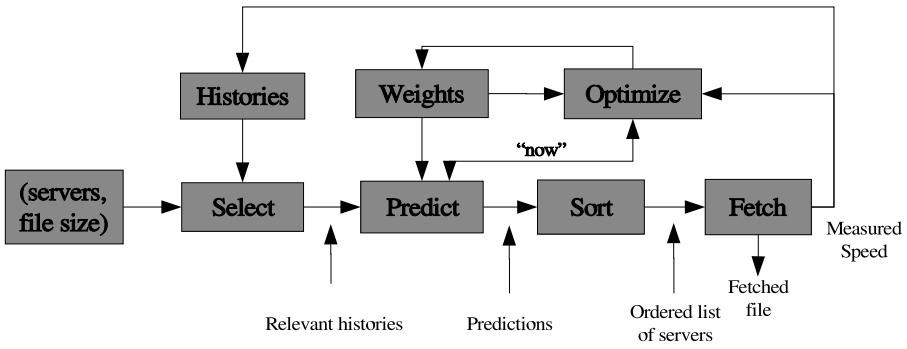

Figure 2: Feedback Mechanism

cut. Unless such servers are periodically used, their actual performance might increase significantly without being noticed. While our design does not require a training period as such, all servers should be exercised at least once, and poor performers periodically in case they have been upgraded. If not done, the user must be willing to live with the performance of the commonly used servers.

4 Maintaining History

Learning requires a history of performance. In this section we describe what needs to be recorded and how it can be efficiently recorded.

4.1 What to Record

In the case of file downloads, a performance history should capture the server name, time of download, file size and measured download speed. The file size is important, which may come as a surprise. As an example, suppose a slowdown tends to occur for some (but not all) servers at 1 minute after the hour. Transfer of small files starting at the top of the hour will likely not be affected for these servers. By contrast, downloads of large files will be drastically affected by slowdowns beginning shortly after commencement. Recording the file size can take this phenomenon into account.

Optionally, the file's MIME type can be included, accounting for performance variance where one server name is aliased to separate servers for .mpeg, .jpeg and other files. It could also be used to choose servers when simultaneous downloads are undertaken, and it is known that total bandwidth is insufficient to meet all optimal download choices. In that case, QoS decisions can be made if MIME types are known. For example, an audio download has delay limits that must be observed, but a text file can be downloaded at leisure. To reduce

complexity, the recording of MIME types is not done in the current incarnation of the project.

4.2 Storage

Because download metrics are likely to be sparse, the overhead of a full DBMS is unlikely to pay for itself. A human-readable flat-file is easier to maintain and exacts a minimal penalty for time spent reading and writing to it.

The next question is whether the data should be binned in RAM for quick indexing. Several schemes exist, including for temporal data. [13, 14] These include R-trees, AD-trees, F-indexing and $D - HS^T$ indexing. However, access is always keyed by server name first, and once fetched, that dataset is iterated exhaustively, and thus fast indexing is of little use. Additionally, binning requires educated guesses as to the center of gravity for features of interest. We do not know what features exist *a priori* in our time series [19], and so this is likely to be a complicating factor rather than an aid.

Thus, we keep the history for a particular type of metric (e.g. download speed) in a flat-file of Lisp s-expressions, one per line. Each record is of the form

(*host utime filesize metric*)

where *host* is the name of the server, such as "ftp.gnu.org"; *utime* is the unix time in seconds since the epoch; *filesize* is in bytes and *metric* is in some convenient unit, such as bytes per second. We chose to use integer values only. This improves the speed of reading from and writing to the text file, as well as that of computations.

Early versions of this system maintained complex data structures for each host, containing said host's historical data. This proved cumbersome, and the simpler data format was chosen despite the extra CPU time needed to form cross-sectional views of the data. A simple cache mechanism was implemented to minimize disk access.

5 Data Analysis

In this section we discuss how predictions are made, and how they are improved.

5.1 Background

The history of sparse time series analysis lies mostly in the domain of astrophysics, where one goal has been to detect periodicities in gamma ray data that is sparse. [19] Several techniques have been used. The Rayleigh Test assumes a periodicity and phase of the data samples, and varies the assumed amplitude and phase until a best fit is achieved. [11] This is equivalent to the Maximum Likelihood Method assuming a given periodic distribution. [1] Epoch-folding accomplishes the same thing by assuming an integer sampling period, projecting each sample time modulo the assumed periodicity, and binning the data. If the

bins fill many to a few bins and few or none to others, a periodicity has been discovered. This has been proposed as more sensitive than Rayleigh testing for narrow signal pulses, but less-so for sinusoidal or broad duty cycle pulses. [10]. Since we may be faced with both feature widths, we cannot rely on just one, but using both and choosing the best at each prediction is computationally expensive.

A combination of a modified Discrete Fourier Transform, auto-correlation and cross-correlation can detect periodicities in unevenly sampled data, but still suffers from the aliasing that hobbles standard Fourier analysis. [18] In addition, the computations are relatively expensive, and require more samples than we are likely to be provided.

Variations on an entropic approach claims to be intuitive, mathematically correct, and fast to implement. [3, 2] They are quite elegant, but still require more samples than we are likely to have available. Bayesian approaches allow several multiple-periodicity models to be assessed, with more complicated models discarded in favor of simpler models of the same predictive quality. [6]

While each of the above approaches allows one to detect unknown periodicities, they are overly complex for our needs, as we can assume that periodicities come in only a few flavors on the INTERNET. This is explored in the Data Analysis section. Because of this, we can start with a simpler model of performance, and use a learning technique to determine, as data comes in, which of the typical periodicities are reflected in the facts. This is important, because we must make predictions even before we have enough data to reasonbly do so.

5.2 Data Clustering

Having stored tuples of (*host utime filesize metric*), we need to choose which data are relevant to the prediction at hand. In our case, we are likely to be given a list of servers from which a file may be downloaded. We are also likely to know the size of the file (though this need not be true). Thus we want to analyze the history of downloads for each of several servers when downloading files of a certain size. The host with the best predicted performance will be chosen for the actual download.

We collect from our history all records for a given host where the file size is roughly that of the file we are downloading. By "roughly" we mean files within some percentage of the target file size. This tolerance must be flexible. Too-tight tolerances could eliminate all samples, especially since data is likely to be sparse. Too-loose tolerances might include samples whose download times would overlap periodic slowdowns while the current file's download would not (or vice versa), again skewing predictions.

Our chosen heuristic is to first collect all samples for a given server. Every sample within some size tolerance (e.g., $\pm 20\%$) of the target file size is automatically kept. Each sample falling outside that size window is examined to determine its relevance. To do this, a time window (of, say, a minute) is defined around the sample in question. If there is another sample within that

time interval that is also within the size tolerance, the sample in question is discarded.

If there is not another sample within the size tolerance and time interval, the sample in question is kept. With this technique, we try to concentrate on samples very close to the exact category we desire, but with the idea that a sample should be kept if no better is available.

Having selected the relevant samples, we then strip away all but the *utime* and *metric* information, leaving us with pairs of the form (*utime metric*). This reduces the memory footprint, greatly speeding processing.

It is worth noting at this point that this data selection technique is quite different from the usual clustering methods. One common technique is that of kernel regression. [5] In that case, samples are *all* used in a weighted average, the weights determined by some function rewarding points "near" a target point in some p-norm sense. Here, by contrast, our universe of data is highly irregular, with some points in space effectively disappearing if other, "better" points are filled. It is as though the target filesize can switch right-sized samples into "black holes" that swallow nearby wrong-sized samples. In the next section, we shall see that this data universe also contains "wormholes" (gravitational lenses?) that make some samples in the distant past appear to be close to the targeted time of prediction. Such samples are thus weighted much more heavily than would otherwise be expected. This contrasts strongly with the norm-based clustering techniques.

5.3 Approaches to Analysis

At this point we decide how to predict what value of metric will be measured at this instant, given the history just collected. There are several approaches, not all useful.

5.3.1 Simple Averaging

Simple averaging of all collected metrics is fast, but is a poor predictor if the data has a large crest factor. That is to say, infrequent but large peaks or dips throw off the prediction and can lead to the selection or rejection of a server on false grounds. We can ameliorate this problem by tossing out samples more than one sigma away from the mean, but this assumes their variance is caused by noise, and thus fails when the history has periodic components. For example, a sine wave is perfectly predictable, but its average is zero, and nearly every value will be an outlier. Simple averaging also fails when the statistics of the samples are not stationary, *i.e.* when a trend exists.

5.3.2 Fourier Analysis

Fourier analysis detects periodicities, but only if data is sampled at or above the Nyquist frequency, and for long enough to detect the lowest frequencies of interest. [16] If we are willing to accept poor predictions for low-frequency events

(say, Christmas Eve lulls in traffic), we can shorten the history to the longest periodicity we care about. This might be weekly, effecting a large savings in storage when downloads are frequent.

If it were possible to use Fourier analysis, predictions would be accomplished in three steps:

- Perform a Discrete Fourier Transform (DFT) on the data.
- Remove frequency components above a desired range by zeroing them from the DFT coefficients. This removes high frequencies likely to be noise, such as minute-to-minute variations.
- Perform an Inverse DFT and extrapolate from the filtered history the value that would occur *now*.

This would be ideal, but as mentioned earlier, the data is typically sparse, making Fourier analysis impossible. Wavelet analysis makes no such assumption about regular sampling, but still requires a mesh of points that capture features of the data. As we do not know the features, this approach is also not usable. [8]

5.3.3 The Holt-Winter Method

There is a way to work around this, if we can assume typical periodicities in the data. In the case of the INTERNET, we can. For example, business schedules affect the INTERNET greatly. The time zone system ensures that bursts of traffic will occur as people in each time zone get to work, go to lunch, leave work, and so on. Further, system backups are typically done at the same time every day because they are automated. The same is true of most weekly and monthly backups. Bursts of traffic on Monday mornings as people catch up with the weekend's email also have their effect. [15]

The result is that there are likely to be hourly, daily, weekly and monthly periodicities, in addition to yearly trends related to school and holidays. Some periods will not be multiples of others, as not every month has the same number of days. Thus it will be necessary to catch events that occur on, for example, the first of each month, and not just every 30 days.

The Holt-Winter Method [7] allows one to make predictions in the face of data with linear and exponential trends, and with an assumed periodicity. H-W is computed much like a traditional moving weighted average. However, given data for at least one, and preferably two assumed periods, H-W gives added weight to values that occurred at multiples of said period in the past. H-W can include multiple periodicities in its weighting, making it seemingly perfect for our purposes. However, as noted earlier, some periods are irregular, determined by a date of the month, not by a multiple of a smaller period such as a week. H-W fails to account for such irregular periods.

Worse still, as with Fourier analysis, H-W requires regularly sampled data, as a missing datum will cause a divide-by-zero error unless an assumption is made about its value. [7] Such assumptions are unlikely to be meaningful –

if they were meaningful, we could assume the value we are trying to predict. Assuming *possible* periodicities, we must emphasize evidence of them that exists in sparse data, but *not* imagine it where it is not.

5.3.4 An Alternative Method

Consider now a weighted averaging scheme with two sets of weights. The first set emphasizes samples taken in the recent past over those in the distant past. Samples taken within the last minute start with a weight of 10; the past 15 minutes, 8; the past hour, 6; the past day, 4; the past week, 2; all others, 1. These initial values codify the assumption that a recent sample is likely to be a better predictor than older samples. For example, the older samples may have occurred before a change in a server's network card or location in a network.

The second set of weights scales these initial weights for each assumed periodicity a sample may fall on. We will call samples that fall on assumed periodicities "anniversary" samples. For example, if a sample was taken a multiple of 60 seconds ago, it is a minute-anniverary sample. A one minute anniversary sample's initial weight (10) is scaled by 2, resulting in a weight of 20. Any sample taken a multiple of an hour ago will be scaled by 2, and similarly for anniversariy periods of 1 day, 7 days, same date of the month, and same month of the year. These increases in weight are cumulative. Thus, a sample from exactly 2 weeks ago will start with a weight of 1, but be doubled for falling on periodicities of one minute, one hour, one day and one week, resulting in a total weight of $1 \times 2 \times 2 \times 2 \times 2 = 16$.

Once all the data has been weighted, the weighted average is computed, and this is the prediction for the performance metric if a measurement were to be taken *now*. This weighting scheme has a few interesting properties.

- If the metric for a given download site is actually constant, this fact will not be misinterpreted, despite the assumptions about periodicities.
- Suppose there are weakly delineated trends, such as slowdowns through out the month of December, and that *now* is in the month of December. Last December's samples will get modest additional weighting, gently emphasizing any data that might contain information about a yearly periodicity occurring each December.
- Suppose there are sharply defined trends such as slowdowns at 5pm, December 24th. Predictions at that moment will weigh one-year anniversary samples heavily, as they are periodic by month and date and hour and minute. Lack of a sample from exactly a year ago will allow us to miss such a periodicity, but that is unavoidable when the data simply does not exist.
- Very recent samples are still weighted more heavily, as they reflect what is happening *right now* more than does historical data, which merely gives hints as to what *might* happen if *now* falls on some period.

5.4 Optimizing

Our initial weights are likely to need adjustment, and on a per-server basis. Firstly, they were merely reasonable guesses. Also, individual servers will vary in their periodicities. Moreover, noise in the data caused by sporadic events will skew predictions, and some form of recovery will be needed. Finally, a server may get its network connection upgraded, fundamentally shifting its performance level. Thus, every server should be given its own set of weights, which should be adjusted after every measurement. This will improve predictions as more samples are taken, and adjust to trends in the INTERNET's performance.

Once a prediction is made for each of a set of servers to be considered, the server with the best predicted performance is selected, the download is performed, and the actual download speed is measured. The measured value is then used as a target value, weights being adjusted until they produce a prediction as close to the measured value as our model allows.

5.4.1 The Search Space

To adjust the n weights in our model, we first define an n-dimensional search space, where each point represents a tuple of weights. The search region should be centered around the current tuple. It should also be limited in size. This keeps us from jumping wildly between weights, encouraging us to instead slowly adapt. The rationale for this is explained below.

In the algorithm's current form, the tuple-space to search is the 1-norm sphere centered on the current weight set. That is, the neighborhood to search is an n-dimensional cube centered on the current weight set, varying by $\pm$ 1 in each weight coordinate. The values to check in that neighborhood are taken to be the points with integer coordinates. For example, in 2-dimensional space, with a center of {3, 4}, we will check the following tuples including and surrounding that point: {2, 3}, {2, 4}, {2, 5}, {3, 3}, {3, 4}, {3, 5}, {4, 3}, {4, 4} and {4, 5}.

If a better set of integral weights (better than, say, 5% decrease in absolute error between the prediction and measured performance) is found in that set, that tuple is made the center of a new cube, and the process is repeated. This loop terminates when no improvement of more than 5% percent can be found in the search cube. The best set is stored for use when the next ranking of servers is to be made.

As mentioned above, the search space is not allowed to be infinite, reducing the chance of a lucky, but wrong, set of weights being chosen. For example, we know that recent measurements are the most important, but suppose there are only a few data points, and one happens to have nearly the same metric as the measured value, and is *coincidentally* at some assumed period in the past. Allowing an infinite search space could produce a set of weights that are all zero except for that one periodicity. The next prediction is likely to be thrown way off, especially if it falls on a real, but now ignored, periodicity. Thus, a minimum value of 1 is enforced for each weight, limiting the search space.

The search is also kept local by termination when improvement is deemed too small to be of real meaning. Our initial choice of weights assumes uniform weighting for all anniversaries, so none is given more emphasis than others before evidence accrues that such a preference is warranted. A global search could, as in the above example, force weights to immediately emphasize samples at nonexistent periodicities. A local search starting from our initial conservative assumptions is less likely to stray without reason.

5.5 Pruning Data

It is possible that some server choices will produce nearly constant measurements. Keeping more than even one datum is superfluous in such cases. Of course, one cannot feel safe in believing near-constant performance without many measurements over a long timespan, but keeping an infinite history can bog down the system. One compromise is to simply delete values of more than some age, say 3 months. This is unduly strict if few measurements exist, so one could instead delete samples beyond some number, n. This, however, could lead to ignoring important larger periodicities if samples are taken fairly often.

Another approach is to, at the time of each new measurement, run a prediction/optimization cycle for an entire data set of the given server, and then for the data set with the oldest datum removed, and then with the next oldest removed, and so on. As soon as the optimized prediction fails to predict within some pre-determined tolerance, the previous (one datum larger) data set is kept, as are the parameters that made that successful prediction. Gradually shortening the history removes extraneous values without being fooled by a lucky sample in the recent past that might lead us to prune most of the history and thus ignore periodic events that are known to have occurred. Obviously, for any anniversary of interest (one month, one year, etc.), a history at least that long must be kept.

There is inevitably a trade-off, as more optimization/pruning cycles require more time that could be spent downloading. There is likely to be an optimal dataset size for a given server, below which prediction is fast but misses important trends, and above which spurious data are mistakenly kept, continuing to bias predictions. The time spent pruning must be balanced against the expected download savings, which is a difficult feat, given the need to compare predictions for multiple servers, each of which will have a different history.

Given the murky nature of the trade-offs involved, no pruning is done in the current algorithm, as the datasets are expected to be small, and so each optimization will not be too burdensome.

6 Performance

There were several obstacles to analyzing the performance of the system, preventing a thorough conclusion about its effectiveness. We enumerate them and draw what conclusions we can without extensive experimental results.

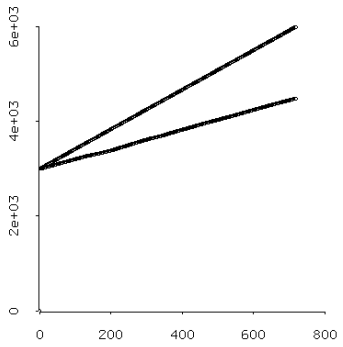

Figure 3: Tracking of Linear Trend.

6.1 Test Data

The ultimate test is running the sytem in a real environment such as the FreeBSD Ports system [12] mentioned earlier, and doing so for the nearly 10,000 Ports in FreeBSD, for a year. The size of this task is not the only obstacle.

The *fetch* utility of FreeBSD is written in a way that makes it impossible to provide a useful Lisp wrapper to work with the existing code. That effectively cuts the feedback loop where we measure actual download speed and use it to re-optimize the weights. Similar problems were encountered with the other file transfer applications like *wget*, and rewriting them in Lisp or modifying the C code was deemed out of scope for this project.

As a result, synthetic data was chosen as a substitute. A test harness was designed to simulate a history of regularly sampled performance metrics, and the model was tested to see how well it tracked changes in performance.

6.2 Linear Trends

Figure 3 shows the tracking of a 100% linear rise from 3000 to 6000 bits/sec. The lower curve is the system response. The horizontal axis has units of hours, with one sample per hour. There is no oscillation evident in the tracking, but the tracking does not approach closely to the curve of simulated measurements. This occurs because we don't let weights take on a zero value. Old samples all get a weight of at least one, and as more accumulate, they overwhelm recent samples. Because the local search used during weight optimization stops when improvement is less than 5%, the coefficient for recent and anniversary samples is never increased enough to overcome this problem.

Figure 4 shows the response to the same linear trend as before, but corrupted by white noise. Noise in the history is well tolerated, not significantly altering the response.

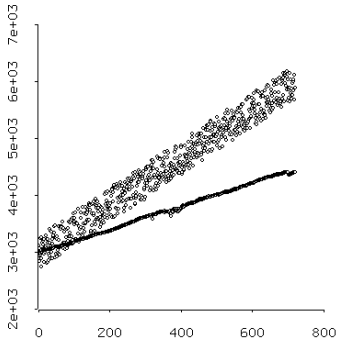

Figure 4: Tracking of Linear Trend, with Noise.

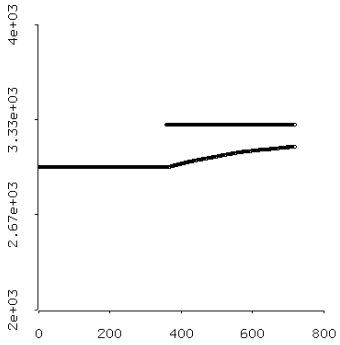

Figure 5: Tracking of 10% Jump.

6.3 Level Shift

If a server connection is improved, there will be an abrupt change in its performance. To measure our adaptation rate, we generated step functions from 3000 bits/sec to 3300 bits/sec (a 10% jump) and from 3000 to 6000 (a 100% jump). Figure 5 shows the results of the 10% jump and Figure 6 shows the results for the 100% jump. The horizontal axis has units of hours in both figures.

The first section of data was tracked perfectly after the first sample, as our predictor is not misled by steady-state samples. When the step occurs, the predictor follows an exponential approach toward the new value. The weights immediately jump to values that de-emphasize anniversary samples and heavily emphasize recent samples. As before, noise is well rejected.

Figure 8 shows the response to a 100% linear increase superimposed on a short term peak. As with the simpler cases, response to changes begins immediately, but is overwhelmed by large numbers of samples in the distant past.

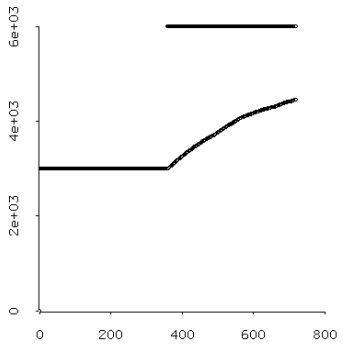

Figure 6: Tracking of 100% Jump.

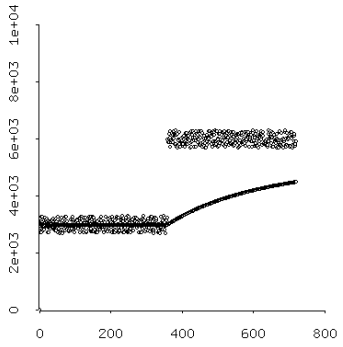

Figure 7: Tracking of 100% Jump with Noise.

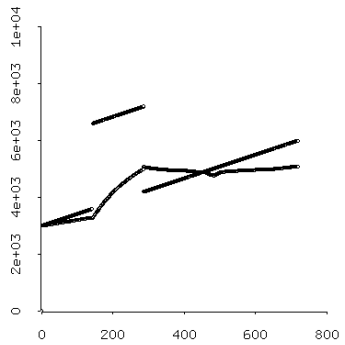

Figure 8: Tracking of 100% linear increase plus short term burst.

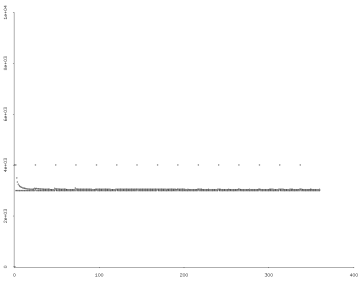

Figure 9: Tracking of 100% spikes every 24 hours.

We cannot use windowing to reject these samples. Doing so loses data needed to provide evidence of periodic behavior.

6.4 Periodic Components

Here, the algorithm shows promise of success, but still suffers from the inability to weight anniversary samples heavily enough to overcome the bulk of older samples. Figure 9 shows that the periods are being detected, but not weighted heavily enough for their amplitudes to be tracked accurately.

As before, noise is well rejected. No tests were performed with multiple periodicities present, or with simultaneous trends, shifts and periodicities.

7 Conclusion

We have described a predictive model which, unlike other models, works with sparse data sets. We assumed that only certain periodicities might be present,

and coined the term "anniversary" to refer to any time that is a multiple of one or more of these assumed periods in the past. By forming a weighted average that emphasizes very recent samples and those occurring on anniversaries, we attempt to detect periodic events increasingly well as our history lengthens, while recognizing that very recent samples are better predictors than old ones.

Our model rejects noise well. It responds to trends and sudden shifts, but does not adjust quickly. It detects periodicities, but doesn't track their magnitudes accurately. In both cases, the cause is that the search mechanism is local, and sample weights are at least 1. As more data is collected, samples with a weight of 1 (not anniversary or recent samples) accumulate and overwhelm anniversary samples in the weighted average.

We could allow weights to become zero in order to prevent significant samples from being overwhelmed. However, this would allow oscillation of weights as mentioned earlier – samples with coincidentally the same value as that actually measured could be falsely treated as periodic events. We could also allow for global search, but this is time-consuming and is still not guaranteed to solve the problem.

The current model is unsuitable for fast-changing data where fast decisions are needed, as re-optimization is too slow. Thus, its use is limited. When used for its original purpose of improving download times, the algorithm is faced with little in the way of trends. It deals with relatively small (less than 50%) periodic slowdowns that affect an otherwise steady level of performance. The re-optimization time is negligible compared to the download times. Thus, despite its weaknesses, the model should be useful for applications such as the FreeBSD Ports system and file sharing nets. A side benefit is that by optimizing server choices, INTERNET traffic is localized, reducing cross-country traffic jams, and improving INTERNET performance for all.

8 Further Research

A trend-sensing mechanism such as Holt's Linear Exponential Smoothing predictor [7] should be incorporated. Otherwise linear trends will have a lower bound on their error, and their detection will be sensitive to noise. This is discussed more fully in the next section.

Comparison of additive *vs* multiplicative sample weighting should be more carefully investigated to determine how each affects predictive accuracy and weighting oscillation.

Wrappers for file fetching utilities should be written to make this sytem of practical use. A previous version that relied only on a single *ping*-based prediction improved download speeds by some 20%, just by avoiding the worst performing servers. By incorporating our feedback mechanism, we can deal with servers which reject *ping* requests, and thus use real measurements to not just avoid the worst choices, but pick the best.

8.1 Yet Another Approach To Feature Detection

Our pattern detection currently makes the mistake of giving a year-anniversary sample extra weighting because it falls on hour, day, week and month anniversaries, as well. This correctly rewards a sample for being close to an exact anniversary (e.g. exactly 2 months ago), but blindly gives it weighting similar to a sample taken a minute ago. Such preferences should depend on corroborating evidence, such as other month-anniversary samples that agree with it. With such clusters, evidence of trends can be more credibly inferred.

A more sophisticated approach would begin by defining a "feature" as any evidence of a periodic event (weekly, daily, etc.). A given feature might exhibit constant behavior (e.g. the same value every Monday morning at 8am), or exhibit a trend (e.g. linear increase from month to month). One might build, as much as possible, every likely feature from the data, apply trend analysis to each feature, and weight each of the possible trends' predictions proportional to the number of samples that could be counted as part of each feature.

As an example, suppose that 5 samples fell on week anniversaries, 4 on daily (one sample shared between week and day anniversaries), and 3 on hourly. A simple linear trend could be assumed for each feature, and the next value predicted for each feature. The prediction of the weekly anniversary feature would be weighted by $5/(5+4+3)$, the daily by $4/(5+4+3)$ and the hourly by $3/(5+4+3)$.

With this approach, we might overcome our current inability to follow a trend (as per the Performance section). We might also pick up on periodicities more selectively, as we give more weight to those for which more corroborating evidence exists.

As a modification, we might assume that weekly features are more likely to be "big" events than daily (e.g. weekly vs. daily backups), and weight the trend predictions accordingly. We might even decide to ignore any trend that was "too noisy" to represent an actual trend. We could also allow samples to participate only in feature cluster(s?) where they did not stand out "too much".

Again, these are prescriptions for improving our predictive capability, but pale in their effectiveness compared to a version of *ping* that would accurately measure actual current performance.

A Experiments

This appendix describes a series of small tests and thought experiments run to determine good approaches to this problem. Some of those have already been mentioned; others are implementation details and optimizations not previously mentioned. Some serve as cautionary tales.

A.1 Fourier Analysis

The SAPA textbook [17] has inspired a Common Lisp implementation of Fourier and statistical routines, called "SAPA"[1]. We used the DFT and inverse DFT routines from SAPA.

These routines expect data input in arrays, and produce arrays of complex results. Lisp wrappers were written so that input could be provided as lists of reals, and output massaged into lists of reals stripped of epsilon errors. Otherwise, no changes were required for the use of the package. Should data become available in regularly sampled form, sapa will be the preferred package to use, based solely on its ease of use.

The lack of regularly sampled data is, of course, what prevents the use of Fourier analysis. Therefore, there is little more to say about this topic.

A.2 Means

Prediction by average is fraught with trouble for any dataset not clustered closely about a stationary mean. There are several datasets that display this problem. One is the dataset with stationary statistics (fixed mean and variance), but with a large variance, or equivalently, noise. Another is the dataset with periodic fluctuations. There is a continuum of such sets, ranging from the generally steady (large steady-state component with small low-frequency components) to those with pronounced periodic components.

Another category which appears to be (but is not) a periodic dataset with pronounced low frequency components, is the one with non-stationay statistics. For example, consider a new ISP whose service suffers the typical new-provider peformance problems, but whose service gradually improves. A trend is present, superimposed on content that may be either DC or strongly periodic.

Capturing a trend with averaging can be done by averaging differences between samples, and adding the average difference to the last sample to predict the next.

$$\frac{(s_1 - s_0) + (s_2 - s_1) + ... + (s_{n-1} - s_{n-2})}{n-1} + s_{n-1}$$

A (very) little algebra will show, however, that this reduces to

$$\frac{s_{n-1} - s_0}{n-1} + s_{n-1}$$

[1] The SAPA package is available from the Association of Lisp Users, www.alu.org.

which merely draws a straight line between the first sample and the last, and extrapolates. This misses any but linear trends, of course. Holt's Linear Exponential Smoothing is the preferred method when periodic phenomena are not exhibited. [7]

B Performance Issues

Data storage and prediction, even with inefficient algorithms, turned out to be more than fast enough, using far less than a second in total per prediction. The main performance obstacle encountered was the speed (or lack thereof) of the optimization algorithm. This was tackled on two fronts, algorithmic and code-centric.

B.1 Algorithm Simplifications

The optimization of weights through exhaustive search, even if only a local search, is computationally expensive (exponential in the number of parameters). Any reduction in the number of parameters is thus a great help. To begin, the number of assumed periodicities can be reduced, on the assumption that periods of a minute are unlikely to exist, and periods of a year and a date of month are rarely significant compared to an hour, day, week and month.

The separate weights for distance in the past (10 for being within the last minute, 8 for the last 15 minutes, and so on) provide fine-grained adjustment, but can be aproximated by a single weighting

$$max(k/x, 1)$$

where k is the weight parameter and x is the absolute difference between *now* and the sample's *utime*. k can be initialized so that

$$k/x$$

approximates the weights assumed in our original multi-weight formulation. Thus our weight-space can be reduced from some dozen dimensions to 5, providing a major reduction in time spent per estimate. Note again that we ensure every sample is given a weight of at least 1, since every sample provides *some* information and should not be ignored.

Taking that simplification one step further, the integer division for the

$$max(k/x, 1)$$

weighting can be replaced with

$$max(k - x, 1)$$

where k is initialized to approximate our original model. This keeps integers smaller than does the reciprocal formula, increasing the chances of using small-integer arithmetic. It does, however, give less weight to very recent measurements than does the reciprocal formula, and k should be chosen so that a very

recent measurement strongly outweighs a single measurement from, say, exactly one year ago. The learning mechanism will eventually correct a poor choice of k, but only if the choice is not so poor as to prevent a given server from ever being chosen to participate in a download and have its actual performance measured.

Even more savings can be accrued by adding the periodic weights rather than using them as factors in multiplication. This changes the model significantly, but should not prevent reasonable predictions (especially given that we are faced with a larger obtacle — sparse data).

Less significant than search space size, but more than the algorithm tweaks is the method used for search. Our original search technique was exhaustive within a small neighborhood, which makes the reduction in dimensionality vital. This has been changed to a steepest-descent search that follows improvements as long as they are to be found. Since local search is important to damping coefficient oscillation, switching to an any-time technique such as random walk or simulated annealing, both of which search for global optima, is not considered desirable. The current algorithm has also been adjusted to not re-compute predictions for coordinates shared between one neighborhood and the next. Previously, between $(2/3)^n$ and $2/3$ of already computed values were re-computed whenever a new neighborhood was searched, where n is the number of weights being optimized.

The result of all these improvements has been a speed increase on the order of tens of thousands over the original brute-force approach. Most of the credit goes to the change in local search from an exhaustive search in a neighborhood to a steepest-descent approach. Following in rough order are the reduction in dimensions and elimination of re-computation. The change from a 1/x model to a k-x model was of minor importance to performance.

B.2 Code Optimization

There are several techniques common to the optimization of Lisp code. We examine a few here, mostly related to typing. Type declarations are required so that the compiler can optimize the code it generates. It is up to the coder to make sure those type restrictions are not violated. While specializing of types is important for efficiency, it provides a second order effect compared to reducing the search space and using a better search algorithm. Note also that compiler optimizations are optional for Lisp implementations, so that they can be specified but ignored by compilers not written to handle them. Check your Lisp implementation before spending time on such declarations.

B.2.1 Numerical Types

By declaring numerical types, the compiler can avoid the use of generic arithmetic. Lisp can process any kind of number (*e.g.* integer, float, complex with rational components) without declarations, but at the cost of type-checking that value and function-dispatching at run time. Integer math is typically much faster than floating point, so the more we can use integer measurements and

weights, ignoring fractional values, the more the Lisp compiler can optimize the executable code.

Thus we specify time as Unix time — a positive integer number of seconds. Weights are specified as integers of value at least 1 and at most 65535, thus fitting into 2-byte boundaries and guaranteeing that intermediate results will be of a size suitable for optimized arithmetic. If it is found that weights can be further restricted to the range of [1, 255], the compiler can optimize even further. Avoiding multiplication and division in favor of integer addition and subtraction is also a big gain.

B.2.2 Container Types

Surprisingly, use of structs rather than lists for data storage did *not* improve speed, indicating either good list accessing or poor structure accessing on the part of CMUCL. Use of arrays is not advantageous, either. Array access requires bounds-checking to be safe, and that overhead is avoided in list traversal. If every element of a set is to be accessed sequentially, small lists as we have here are faster than small arrays. Even for random access of sets of 5 parameters, the CMU Lisp compiler produced faster code for lists than for arrays.

References

[1] Bai, T. Methods of Periodicity Analysis: Relationship Between the Rayleigh Analysis and a Maximum Likelihood Method. The Astrophysical Journal, 397:584-590, 1992, October 1.

[2] Cicuttin Andres, Alberto A. Colavita, Alberto Cerdeira, Radu Mutihac, and Silvio Turrini. A Simple Method for Detecting Periodic Signals in Sparse Astronomical Event Data. The Astrophysical Journal, 498:666-670, 1998, May 10.

[3] Cincotta, Pable M., Mariano Mendez and Josue A. Nunez. Astronomical Time Series Analysis. I. A Search for Periodicity Using Information Entropy. The Astrophysical Journal, 449:231-235, 1995, August 10.

[4] Links to these implementations are available from www.alu.org.

[5] Deng, Kan and Andrew W. Moore. Multiresolution Instance-Based Learning. IJCAI-95 preprint.

[6] Gregory, P.C. and Thomas J. Loredo. A New Method For the Detection of a Periodic Signal of Unknown Shape and Period. The Astrophysical Journal, 398:146-168, 1992, October 10.

[7] Robinson, Tony. Forecasting Techniques. 2002. Document available from www.bath.ac.uk/ masar/math0118/forecasting/node1.html.

[8] Daubechies, Ingrid, Igor Guskov, Peter Schroeder and Wim Sweldens. Wavelets on Irregular Point Sets. Philosophical Transactions of the Royal Society. London. 1999 (submitted).

[9] Labovitz, Craig, G. Robert Malan and Farnam Jahanian. Internet Routing Instability. Proceedings of the ACM SIGCOMM '97.

[10] Leahy, D.A., R.F. Elsner and M.C. Weisskopf. On Searches for Periodic Pulsed Emission: The Rayleigh Test Compared to Epoch Folding. The Astrophysical Journal, 272:256-258, 1983, September 1.

[11] Mardia, K.V. Statistics of Directional Data. Academic Press, 1972.

[12] Mock, Jim. The FreeBSD Handbook, 1st edition (2000).

[13] Moore, Andrew and Mary Soon Lee. Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. Journal of Artificial Intelligence Research 8 (1998) 67-91.

[14] Patterson, David, Mykola Galushka and Nial Rooney. An Effective Indexing and Retrieval Approach for Temporal Cases. American Association for Artificial Intelligence (FLAIRS), 2004.

[15] Paxson, Vern. End-to-End Routing Behavior in the Internet. Proceedings of SIGCOMM '96, LBNL-38866.

[16] Proakis, John G. and Dimitris G. Manolakis. Introduction to Digital Signal Processing. MacMillan, 1998. ISBN 0-02-396810-9.

[17] Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques. Donald B. Percival and Andrew T. Walden, Cambridge University Press, Cambridge, England, 1993.

[18] Scargle, Jeffrey D. Studies in Astronomical Time Series Analysis, III. Fourier Transforms, Autocorrelation Functions, and Cross-Correlation Functions of Unevenly Spaced Data. The Astrophysical Journal, 343:874-887, 1989, August 15.

[19] Swanepoel, J.W.H. and C.F. de Beer. A new powerful test for periodic pulsed emission of high-energy photons. The Astrophysical Journal, 350:754-757, 1990, February 20.