Modelling Techniques and Tools for Performance Analysis
D. Potier (Editor)
Elsevier Science Publishers B.V. (North-Holland)
© INRIA, 1985

A Synthetic Driver for File System Simulations

M. Satyanarayanan

Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213,
U.S.A.

Abstract

A driver is a software module that mimics an actual usage environment by generating a stream of requests for use in simulations or in stress testing of an implemented system. This paper describes the design of a computationally efficient driver for use in performance evaluation of file systems. Based on observations of a timesharing system in a research environment, it is shown that files can be naturally classified into three equivalence classes: System files, Temporary files, and User files. The usage patterns of each of these classes is shown to be radically different, and a separate locality model is developed for each class. The driver is parameterized, and actual values for these parameters are derived from the experimental observations. Parameters repesentative of other environments may be used with the driver to extend its domain of applicability. Insights from the experimental data presented here may also be of use in other applications such as capacity planning and file system design.

1. Introduction

The work reported in this paper arose in the context of a file system design for a network of personal computers. The goal of this design is to provide location-transparent access to a large shared space of files, using personal computers which are diskless or have relatively small disks. This goal is attained by applying the concepts of caching and virtual memory to the storage of files. Further details on the design can be found elsewhere [1, 7].

In order to evaluate alternative caching strategies by simulation, one needs a driver that generates file system requests with interarrival times and address locality properties similar to those expected in the target file system. This paper describes the design of such a driver, based on experimental observations of a PDP-10 timesharing system in the Department of Computer Science at Carnegie-Mellon University. An earlier study of this file system [6] analysed its file population in detail.

The users of this system typically perform one or more of the following activities:

- Developing programs for research projects.
- Producing documents, typically research papers and dissertations.
- Reading and updating electronic mail and bulletin boards.

The system is almost never used for database or scientific computing activity. Strictly speaking, the results of this paper only apply to the system on which observations were made. However, it is likely to be of value to other academic computing sites with similar usage characteristics.

Previous work on file reference patterns has focussed on long-term locality, typically at the granularity of one day [3, 9, 10]. The questions of interest there have usually been of the form "Has this file been touched today? Can I migrate it to an archive?." This migration typically takes place once a day and usually involves intervention by a human operator. In contrast, our proposed network file system design involves short-term file migration decisions, at a granularity of seconds or milliseconds. The migration occurs continuously, without human intervention. No prior published work exists on the subject of short-term locality in file reference patterns.

The first half of this paper describes the structural aspects of the driver and the modelling assumptions used in it. The latter part of the paper presents experimental measurements which are used to derive parameter values for the driver. A concluding section discusses ongoing work and extensions.

2. Methodology

To permit accurate performance analysis of a multi-level memory hierarchy, a driver should meet the following criteria:

- The *relative frequency* of use of different functions of the storage system and correlations in their usage patterns should be reflected in the requests generated by the driver.
- The *interarrival times* of generated requests should reflect those observed in the environment being modelled by the driver.
- The requests generated by the driver should access storage addresses in a manner that accurately reflects the *locality patterns* observed in practice.

The best way to rigorously meet all these criteria is to instrument a file system and record every request made to it over a period of time. Such a trace of requests would be analogous to an instruction execution trace of a program. Unfortunately, this approach poses problems:

- The programming effort involved in instrumenting a file system is nontrivial, unless such instrumentation was included as an original part of the file system design. Debugging and testing the instrumented file system has to be done stand-alone, in the absence of regular users. This is difficult to do on a computing resource which is heavily used all day. On the other hand, instrumenting a lightly-used system may not yield realistic observations.[1]
- While a trace accurately portrays the context it was recorded in, there is no obvious way to extend it to represent other situations. In predicting performance for unimplemented systems or for anticipated loads, one needs an easily-parameterized source of requests.

Motivated by these considerations, the approach taken here has been to develop a synthetic driver based on an understanding of the way in which typical file systems are used. Macroscopic observations of an actual file system are used to provide one set of parameter values for this driver. The driver may be extended to represent other situations by changing these parameter values. Work currently in progress to obtain microscopic observations (actual traces) is discussed in Section 5.

Since simulations are notoriously expensive in their consumption of memory and cycles, the computational efficiency of a driver is a matter of no small concern to a performance analyst. Many aspects of the driver design described here are expressly intended to address this issue.

[1] A possible way to solve this dilemma is to develop the instrumentation on an identical, lightly-loaded system and move it to the target system for actual measurements. Such a development system was not available to us.

3. Structure of the Driver

3.1. Primitives of the Model

A file is viewed as one-dimensional array of bytes, on which *Read* or *Write* operations may be performed on a fixed-sized quantum called a *Page*. No further structure is imposed on a file. Operations within a file are assumed to be sequential, since most files are used that way in the observed system. However, it would be simple to extend this driver to model non-sequential references by incorporating a *Seek* operation.

Most file systems use a two-level abstraction to model a file: that is, a distinction is made between files which are in active use and those which are merely resident on secondary storage. An *Open* operation has to be performed on a file to make it active, and a *Close* has to be done on it after it has been used. The specific semantics of *Open* and *Close* vary from system to system, but are unimportant from the point of view of the driver. A few file systems, notably Multics [4], integrate the file system into the virtual memory system. In such systems, Open and Close operations are not available at the user level though equivalent actions are being performed implicitly by the operating system.

The driver design described here uses a two-level file system model, with the following primitives:

- Opening a file for reading ($Open_R$) or writing ($Open_W$).
- Closing a file (*Close*).
- Reading from a file (*Read*) or writing to it (*Write*).

In addition to these basic primitives, file systems may offer other functions. We ignore those functions for one or more of the following reasons:

- *They are used rarely enough to be ignored.*
  For instance, certain file systems have a "scavenge" operation in which every disk block is examined in order to reconstruct mapping tables. Such an operation is time-consuming, but occurs rarely.
- *They are sufficiently undemanding of computational resources that they may be ignored for the purposes of performance analysis.*
  Some file systems have a primitive that returns information on how much secondary storage has been used by a logged-in user. The data needed to answer such a query is usually maintained in main memory for the duration of a user's log-in session. Consequently the use of this primitive hardly impacts file system performance.
- *They may be represented in terms of one or more of the operations listed above.*
  A frequently used operation such as listing a directory can often be represented in terms of more basic operations. In file systems in which directories are implemented as files, a directory listing primitive may be viewed as a read operation on the file representing the directory.

In this paper, therefore, we restrict our attention to the primitives of the two-level file system model and ignore all the other operations which contribute to the complexity of file systems.

3.2. Domain of the Model

Every file is assumed to be uniquely identified by an integer, its *FileId*. The file address space is initially empty, and files are alloted monotonically increasing FileIds as they are created. The resulting flat file name space is devoid of any structure meaningful to human users. However, a separate *Name Server* may be used to map an arbitrary name space onto the FileId space. A hierarchical file naming convention, meaningful to users, can thus be supported on top of FileIds. From the point of view of the file system, such a name server would appear to be an application program, and file system requests generated by it in the course of name translation would not be distinguished in any way.

All files are assumed to be *invariant*, the data entered into a file at the time of its creation being never altered thereafter. This assumption models write-once storage media such as optical disks, whose use as an archive was pioneered in the file system design for whose analysis the driver was developed [1]. The classical abstraction of a long-term storage entity whose contents vary with time is achieved by associating a fixed *filename* with the FileId of the invariant file corresponding to the current state of the entity. In this model, overwriting a file is represented by the creation of a new invariant file, bound to the original filename. Extending this driver to the more conventional case where files may be overwritten would involve the development of a locality model for Open_W requests, similar to the model for Open_R requests described in Section 3.3.

Each of the five file system operations is assumed to be generated by an autonomous stochastic process, independent of the other processes. As mentioned earlier, files are read and written sequentially in fixed-sized pages. It would relatively simple to modify the driver so that the offsets and sizes of such requests are distributed according to chosen statistical distributions.

3.3. Modelling Locality

The assumption of sequential access automatically determines the locality of Read and Write operations. The first such operation on a file after it is opened uses a byte offset of zero. Each succeeding request to this file uses a byte offset which differs from its predecessor by the page size.

The assumption of invariance determines the assignment of a FileId to an Open_W operation — a number one higher than the FileId of the previously created file can be used. For reasons explained in Section 3.3.4, a slightly more complicated address assignment is done in practice.

Modelling the locality of Open_R operations is a more interesting problem. The approach taken here is motivated by the following observations, presented earlier [6]:

- The *Functional Lifetime* (F-Lifetime) of a file is defined as the time difference between the creation of a file and the last occasion on which it was read. This quantity is a measure of the usefulness of the data in the file.
- In general, files tend to have short f-lifetimes, of the order of a few days. However, the lengths of the tails of the f-lifetime distributions are strongly dependent on the type of file in question. Certain files, such as those corresponding to the executable object modules of commonly used programs are used long after they are created: the 90-percentile value of the cumulative distribution function of f-lifetime for this class is nearly 1500 days. On the other hand, there are files with radically different usage properties: the corresponding 90-percentile value for files containing the output from document processors is only about 80 days. More than 60% of such files have an f-lifetime of one day or less.

There is also evidence to indicate that the rate of reference to files decreases with age [5, 8]. Although this hypothesis has not been verified on our file system, it seems intuitively clear that older files are less likely to be used than recently-created ones. In the absence of information to the contrary, it is assumed that other file systems also possess this property.

Based on these observations, the files modelled by the driver are postulated to fall into one of three classes:

System Files	These are files corresponding to the executable modules of system programs (such as compilers, linkers, and editors) or to files used by an operating system to maintain relatively static information (such as the list of users who may use the system, encrypted password files, and files containing documentation of system programs and commands). Files of this class are frequently read, rarely written, and usually remain in use long after their creation.
Temporary Files	Many programs create files which are used at most once after their creation. For example, intermediate output created by different phases of a compiler are typically not accessed after the compilation is complete. The object files of programs being debugged, error message files created by compilers or document processors, and files generated by document processors to drive printing devices are other examples of this class. Files of this class are often created, but rarely read long after their creation.
User Files	This category corresponds to files which belong to neither of the two preceding classes. Be-

sides the properties mentioned earlier about files in general, there is little that can be assumed about this class of files.

The locality models used for these three classes of files are discussed in detail in the next three sections.

3.3.1. System Files

For the purposes of this section it is necessary to distinguish between the logical function fulfilled by a file and its FileId. Consider a typical system program, such as a Pascal compiler. From the point of view of users, it is appropriate to speak of "the Pascal compiler." However this logical entity is bound to different invariant files at different instants of time, each corresponding to the executable object module of a different release of the compiler. In order to distinguish between such a logical entity and the set of invariant files that represent that entity at distinct points in time, we introduce the term *System Data Object* to stand for the former.

It is assumed that there are N_{Max} system data objects identified by the integers $1, 2, \ldots, N_{Max}$. Without loss of generality, the system data objects may be ordered in non-increasing order of usage.[2] As Figure 1 illustrates, the relationship between system data objects and the corresponding invariant files is analogous to that between array indices and array element values. The set of system data objects can be viewed as an array S, the element of index i representing the i^{th} system data object. The value of $S[i]$ at any instant of time is the FileId of the invariant file which is currently bound to the i^{th} system data object. The assumption about ordering of system data objects implies that if $i<j$, then $\text{Usage}(S[i]) \geq \text{Usage}(S[j])$. Since the number of system data objects is constant over time, the size of the array S is fixed. Changing the value of element $S[i]$ corresponds to replacing the existing version of the i^{th} system object by a new one.

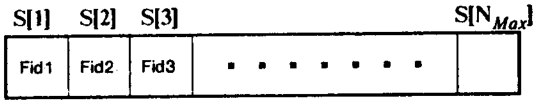


Figure 1: Relationship between System Data Objects and their Invariant Files

Intuitively, a small fraction of the system data objects accounts for much of the usage. For the file system considered in Section 4.2.2, 75% of the usage is accounted for by about 4% of the system data objects; a further 20% is accounted for by about 16% of the system data objects, and the remaining 5% of the usage is accounted for by 80% of the system data objects. While these obcervations are strictly valid for one file system, one would expect a similar pattern of usage to be true for other file systems too. Based on this assumption, the driver partitions the set of system data objects into three classes, with uniform probability of access within each class. The 75-fractile and the 95-fractile of the cumulative distribution function of usage are parameters of the driver. Figure 2 shows the assumed probability density function of accesses, N_{75} and N_{95} corresponding to the 75-fractile and 95-fractile respectively. 75% of the accesses to system files are to system data objects in the range $S[1]$ to $S[N_{75}]$, a further 20% to objects in the range $S[N_{75}+1]$ to $S[N_{95}]$, and the remaining 5% to objects in the range $S[N_{95}+1]$ to $S[N_{Max}]$.

The FileId corresponding to this object is used as the address of the file referenced by the generated request. On the assumption that frequently used system data objects are more likely to have new versions created, the driver also uses the same probability distribution to direct Open_W accesses to system data objects — the newly-created FileId is assigned to the selected system data object.

In summary, the locality model developed above for system files consists of a slowly-changing set of FileIds, a small fraction of which is very frequently accessed, a slightly larger fraction being less frequently accessed, and the majority being rarely accessed.

[2]The number of opens for reads is used as a measure of usage.

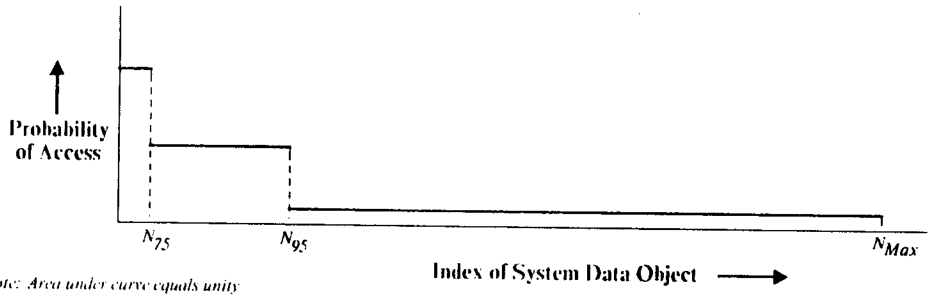


Figure 2: Probability Density Function of Accesses to System Files

3.3.2. Temporary Files

The properties that characterize temporary files are:

- Most of them are rarely used after being created. Quite a few of them, in fact, are never read!
- The few that are used tend to be read within a very short time after creation, typically within a day.

An additional assumption that is made is that the probability of access of a file of this class falls off linearly with time. Although there is no experimental evidence to support or contradict this hypothesis, a linear fall off is the simplest and most reasonable assumption that is consistent with the observation made in Section 3.3 that the rate of reference to files decreases with age. However, it would be relatively simple to modify the driver to incorporate a nonlinear falloff.

These assumptions are adequate to uniquely determine a locality model for temporary files. Suppose all files created are temporary files. At any instant of time τ let F_τ be the Fileld of the most recently created file. Let λ be the time one day before τ, and let F_λ be the corresponding maximum Fileld. Then the Fileld of the file accessed by Open$_R$ request generated at τ is a random variable in the range F_λ to F_τ, with a density function as shown in Figure 3. To implement this model literally, one would have to keep a running history of Filelds between F_λ and F_τ. Since the average number of temporary files created during a day, k, is known, an approximation to F_λ is given by $F_\tau - k$. Using this approximation obviates the need to keep track of a large number of Filelds. The constant of one day is, of course, arbitrary and may be made a parameter of the driver.

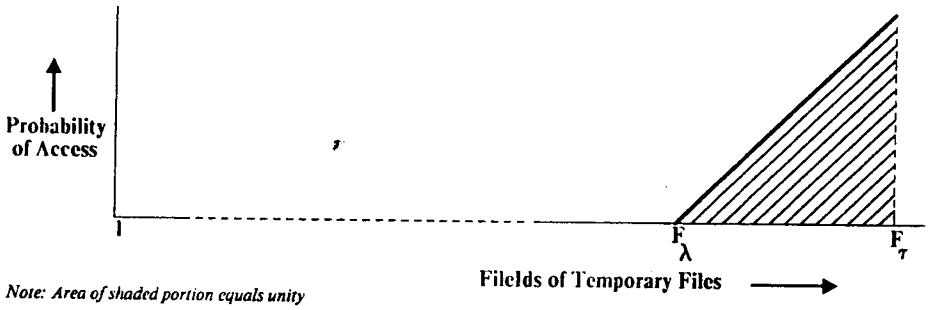


Figure 3: Access Probability to Temporary Files

3.3.3. User Files

The primary difference between user files and temporary files is that the former tend to be accessed over a significantly longer period after their creation. By making the same assumption of linear falloff as for temporary files, the locality model for user files closely resembles that for temporary files. The only difference is that the falloff is spread over a much longer period than one day. The arguments presented in Section 3.3.2 are valid here too, using Figure 4 instead of Figure 3, and $F_{\tau_{cutoff}}$ instead of F_λ. The cutoff interval, $\tau - \tau_{cutoff}$ is a parameter of the driver. Since the interval τ_{cutoff} to the present time τ is quite large, it is advisable to use an approximation to $F_{\tau_{cutoff}}$ rather than its exact value: $F_{\tau_{cutoff}} = F_\tau - (\tau - \tau_{cutoff}) \times k$ where k is the average number of user files created per day.

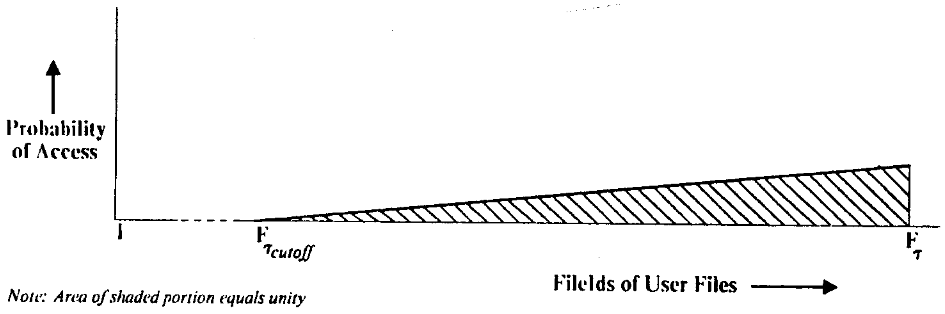


Figure 4: Access Probability to User Files

3.3.4. Distinguishing File Types

In Section 3.3.2 it was assumed that only temporary files were being generated; a similar assumption about user files was made in Section 3.3.3. These assumptions were necessary in order to avoid having the driver maintain a history of FileId versus file type bindings. One way of satisfying these assumptions while generating all three types of files is to have a separate, contiguous FileId space for each file type and to map these three spaces into a global FileId space.

The mapping scheme adopted for the driver is simple: a file with local FileId x is assigned a global FileId $3x$ if it is a system file, $3x+1$ if it is a temporary file, and $3x+2$ if it is a user file. Figure 5 illustrates this mapping scheme. It should be noted that the local FileIds are purely local to the driver — all FileIds used outside the driver are global FileIds. This mapping scheme has two virtues: it is simple, and it allows the type of a file to be easily deduced from its FileId.

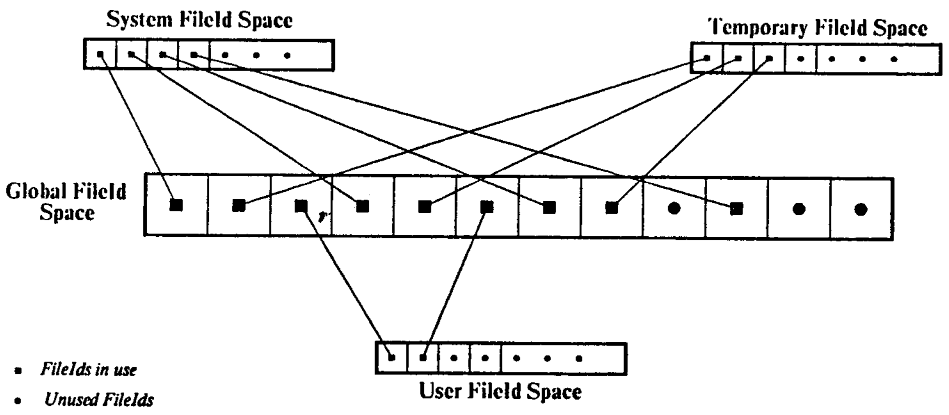


Figure 5: Encoding of FileIds

3.4. Driver Implementation

A driver built on the principles discussed hitherto has been built and used in simulations. The implementation is in C on Unix, and uses a preprocessor providing Simula-like facilities for discrete event simulation.

The driver is economical in its use of space and time. The main internal state of the driver consists of a table containing information about currently open files in the system, which usually number in the tens or hundreds. The total storage used by the driver is typically a few tens of kilobytes at runtime. The amount of time needed to generate a new request is quite small: of the order of a few hundred microseconds. The dominant source of overhead in the driver is, in fact, random number generation!

We have thus been quite successful in building a computationally inexpensive driver that intuitively models the way files are used in real systems.

4. Experimental Observations of a File System

Building a specific instance of the driver requires the following pieces of information which have been left unspecified so far:

- *The interarrival time distributions of each of the five request types.*
  The functional forms and means of these distributions are needed.
- *The fraction of Open$_R$, Open$_W$, Read, and Write requests that are directed to each of the three file types.*
  It is assumed that the fraction of Close requests to a given file type is equal to the sum of the fractions of Open$_R$ and Open$_W$ requests to the same file type — this prevents an unbounded growth in the number of open files.
- *The 75-fractile and 95-fractile of the system file usage distribution.*
  This is needed in order to determine the 3-part distribution discussed in Section 3.3.1.

For consistency with the data reported in [6], the measurements described here were made on the main timesharing PDP-10 system in the Department of Computer Science at Carnegie-Mellon University. However, while the earlier study involved examination of a snapshot of the file system, the results presented here were obtained by collecting data over a period of time. The discussion in this section is divided into two independent parts: one concerning interarrival times, and the other dealing with usage and type-dependent file activity.

4.1. Interarrival Times of Events

The TOPS-10 operating system on the PDP-10 provides the following primitives for performing input-output: *Lookup, Enter, Close, In,* and *Out*. Lookup and Enter correspond to the Open$_R$ and Open$_W$ operations discussed earlier, while In and Out correspond to Read and Write. Close does not distinguish between files which are closed after reading, and those which are closed after writing.

4.1.1. Experimental Technique

The operating system was modified so as to intercept every input-output request to it, and to present it for examination to a metering routine before resuming normal processing of the request. The metering routine noted the time elapsed since the immediately preceding request of the same kind — this quantity, t, was one sample of the corresponding interarrival time distribution. For each kind of event, the metering routine kept track of the total number of events observed, and the sums Σt, Σt^2, Σt^3, Σt^4, and Σt^5, in order to compute the first five moments of the observed distributions.

At intervals of half an hour, a data collection program appended the contents of the histogram tables and moment information to a file and then cleared this data in the metering routine. The choice of half an hour was a compromise between two conflicting requirements:

- Intervals which were too close together would have yielded too few sample points in each interval. Postulating statistical distributions on the basis of too few sample points is likely to lead to large errors.

- Intervals which were too far apart might have missed significant variations in system behaviour, caused by varying user activity during the course of the day.

The choice of half an hour yielded 48 distinct observation intervals per day, while the typical number of Enter events (which were consistently the least frequent kind of event) per interval was about 400. The sampling program was automatically started by the system after crashes, and the sampling epochs were the same from day to day. For instance, if a crash occurred at 12:17 P.M., the next sample would be recorded at 12:30 P.M. and not at 12:47 P.M. as the half hour sampling interval would lead one to expect. The reason for this was that it was judged to be more important to be able to correlate data from sampling intervals across different days than for every sampling interval to be exactly the same length.

Besides histogram and moment information, the information recorded by the sampling program included event identification, a time stamp, sampling interval length, and a count of the number of jobs that had been active during the sampling interval. The data collection was carried on for a little over five weeks, after which the data reductions described in the following section were performed.

4.1.2. Data Reduction

Pertaining to each of the five different events, and each of the 48 half-hour time slots, there is one observation for each day in the data collection period. It is assumed that system activity is different on weekdays and weekends, but is similar on all days within each of these two classes. The collected data can then be viewed as samples drawn from $5\times48\times2$ distinct stochastic processes. Using these samples, the observed mean and variance for each of five moments of these stochastic processes as well as their observed cumulative distribution functions can be calculated.

Figures 6 and 7 show, for weekends and weekdays respectively, the means of the five distributions as a function of the time of day. Since these are means of interarrival times, high system activity is indicated by a low ordinate on the graph. During weekdays, the period from about 10 A.M. till about 6 P.M. shows the most activity — this corresponds to the normal working day. The period just after midnight also shows very high activity, because the operating staff run an archiving program every night. Since input-output operations for archiving are generated by the file system itself and are not caused by events external to the file system, we ignore this period of high file system activity after midnight and assume that the actual user-generated activity during this period is similar to that in the morning, before the working day begins.

A day is partitioned into two periods, a *Peak Period* and a *Lean Period*, and each of these periods is characterized by one of the time slots associated with it. The peak period corresponds to the time slots in the interval 10 A.M. to 6 P.M., while the rest of the day corresponds to the lean period. During the peak period on weekdays, there were about 43 active users on the system. This number represents a rounded average over all peak period time slots. There were about 21 active users during the lean period on weekdays, and about 20 active users during weekends.

For each period, the characterizing time slot is chosen to be the one at which the average rate of Lookup events attains its median value for that period — this occurs at 3 P.M. for the peak period and at 10 P.M. for the lean period. In view of the fact that the activity on weekends is similar to the activity at lean periods on weekdays, we treat these two periods identically in the rest of the analysis. Thus, for each of the five events there are two models that have to be built: one for peak periods on weekdays, and the other for all other periods.

4.1.3. Analytical Models

The means and standard deviations of the five different events during peak and lean periods are shown in Table 1 and Table 2 respectively. An immediately apparent observation is that the observed coefficients of variation are significantly larger than one. Since hyperexponentials are the simplest Markovian models exhibiting this property, attention was focussed on them in fitting closed-form distributions to the data. The heuristic

	Mean (ms.)	Std. Dev.	Coeff. of Var.
Lookup	345.4	622.5	1.80
Enter	2495	2918	1.17
Close	339.9	582.4	1.71
In	209.7	410.0	1.96
Out	557.4	1182.1	2.12

Table 1: Peak Period Interarrival Time Parameters

procedure used for curve-fitting has been described in the literature [6].

	Mean (ms.)	Std. Dev.	Coeff. of Var.
Lookup	562.1	1332.3	2.37
Enter	3389	3684.3	1.09
Close	527.1	1212.9	2.30
In	341.4	906.0	2.65
Out	1144	2341.2	2.05

Table 2: Lean Period Interarrival Time Parameters

The reader can get an idea of the quality of the fits from Figures 8 and 9, which are the best and worst peak period fits to cumulative distribution functions (CDFs). To aid in visually assessing the quality of a fit, one can use a P-P plot [11] which plots the fitted CDF against the observed CDF — a perfect fit would appear as a straight line at 45 degrees and the extent of the deviation from this line indicates the quality of the fit. For example, the P-P plots corresponding to Figures 8 and 9 are shown in Figure 10. The complete set of observed and fitted CDFs and P-P plots for each event in both the peak and lean periods is available elsewhere [7], and is omitted from this paper in the interests of brevity.

Almost all the fits are two-stage hyperexponentials. The sole exception is the distribution of Enter events during the peak period, for which a single stage (i.e., a simple exponential) gives the best fit. Typically, the maximum error between the observed and fitted CDFs is about 3% — in the worst case it is about 7%. However, none of the fits is good enough to pass the Chi-squared test at the 95% level. This implies that the hyper-exponential approximations, while adequate for practical purposes, are not to be assumed to be the underlying statistical models.

Tables 3 and 4 present the parameters of the fitted distributions for peak and lean periods respectively, the α_i's being the probabilities of selection of stages and the M_i's being the corresponding means.

On the assumption that only the means of these distributions change but their shapes remain unaltered, these

	Max Error	No. of Stages	α_i	M_i (ms.)
Lookup	3%	2	.9227 .077	213.2 1812
Enter	3.5%	1	1.0	2102
Close	6.5%	2	.3105 .6894	27.7 474
In	2.5%	2	.8744 .1256	112.2 863.6
Out	3.5%	2	.8530 .1453	175.6 2151

Table 3: Fitted Parameters for Peak Period Interarrival Time Distributions

	Max Error	No. of Stages	α_i	M_i (ms.)
Lookup	3%	2	.09 .966	225.2 2682
Enter	3%	2	.8658 .1342	2066 30000
Close	5%	2	.8851 .1117	209 2429
In	3.5%	2	.8802 .1187	117 1454
Out	5%	2	.7669 .2201	134.6 2376

Table 4: Fitted Parameters for Lean Period Interarrival Time Distributions

fits can be used to model hypothetical loads. To model a load which is k times the actual load observed, the α's of the stages of the fitted distributions remain unaltered while the mean of each stage is $1/k$ times the mean specified in Tables 3 and 4 (recall that a higher load corresponds to a lower mean interarrival time). This load factor, k, can thus be made a parameter of the driver.

4.2. Type-specific File Data

The goal of the experiment described in this section is twofold:

- To determine what fraction of Open_R, Open_W, Read, and Write operations are directed to system, temporary, and user files.
- To obtain the 75-fractile and 95-fractile of the system file usage distribution.

A single data collection procedure yields information on both these quantities. This procedure is described in the next section, and the collected data is examined in Section 4.2.2.

4.2.1. Experimental Technique

A vendor-supplied utility to examine system status is available on the computer system being investigated. The information reported by this utility includes:

- *A list of all the jobs currently in the system and the names of the programs they are running.*
- *A list of system programs for which the shared code ("high segment" in TOPS-10 terminology) is currently resident in primary memory.*
  This list of high segments is a superset of the system programs being currently executed by users. Since nonsystem programs are either absent from this list, or are marked as private, there is a simple way to check if the program being executed by a user is a system program or not.
- *Information on the files currently open in the system.*
  The name of the file and the directory in which it resides, as well as information on whether the file is being read or written, and the number of read or write operations performed on it so far are also reported by the utility.

A batch program was set up to run this utility every hour and to append the data to a file. This hourly sampling of the system was carried out for about two weeks, yielding data on about 15000 files. These samples were then used as input to a program that condenses the raw data.

4.2.2. Data Reduction and Observations

For each sample, the functions performed by the data reduction program are as follows:

- The program being run by each job is examined and classified as a system or nonsystem program. It is assumed that the running of a program involves opening the file containing its executable code and reading it in its entirety. An Open_R for system files (if this is a system program) or user files (in the case of a nonsystem program) is noted. Similarly, the number of Reads to system or user files is incremented by the estimate presented in [6] for the average length of an object file.
- Each open file is classified as a system, temporary, or user file, and the corresponding counts of Open_R and Read, or Open_W and Write, are incremented. The basis for this classification is as follows:
  - Each file has a 3-character suffix to the file name, called its *file extension*, which characterizes the contents of the file. If this extension indicates that a file is a temporary file, it is so classified.
  - Non-temporary files are classified as system or user files on the basis of the directories to which they belong. A subset of the directories in the system are classified as system directories, and the rest are considered user directories. While not perfect, this classification system has not been found to cause errors often.
- A list of system data object names is maintained and each observed use of a system program or an open system file increments the usage count associated with the corresponding system data object.

The quantities of interest mentioned in the beginning of Section 4.2 are obtained from this processed data. Table 5 presents the observed fraction of operations directed to system, temporary, and user files. Table 6 gives the 75-fractile and 95-fractile of the CDF of system data object usage. In accordance with intuition, a small fraction of the system data objects account for the lion's share of the usage.

5. Experience and Extensions

The driver described in this paper has been implemented and used in a simulation study of network file system design tradeoffs. The results of this study are reported in [7]. Unfortunately, the actual implementation of the network file system has not progressed as rapidly as was originally hoped (for nontechnical reasons). Consequently, it is not yet possible to compare the performance predictions made using the synthetic driver with

	$Open_R$		$Open_W$		Read		Write	
	Count	Fraction	Count	Fraction	Count	Fraction	Count	Fraction
System	7638	.632	104	.041	535158	.240	1511	.021
Temporary	709	.059	1580	.620	24781	.011	31197	.443
User	3734	.309	865	.339	1665333	.748	37788	.536

Table 5: Fraction of Input-Output Operations to Different File Types

Fractile	Index of Sys. Data Obj.	Fraction Of Total
75	11	3.8%
95	58	20.3%
100	288	100%

Table 6: System Data Object Usage

actual experience. It is hoped that such a comparison will be possible in the near future.

Another large distributed file system, based on the principle of caching, is currently being implemented at Carnegie-Mellon University [2]. This file system will have an order of magnitude more network nodes, and will serve the entire campus community. It differs in two important ways from the network file system for which the driver described in this paper was developed: entire files (rather than individual pages) have to be cached, and files may be overwritten (they are not invariant). A version of this driver, modified to model variant files, can be used to analyse the performance of this system. This provides another opportunity to validate the structure of the synthetic driver described here.

An extensive experimental effort is also under way to obtain complete trace data from a Unix file system.[3] Some of the questions we hope to answer by this study are:

- How do file properties differ in different systems, assuming that their user communities use files in similar ways?
- Can the driver developed in this paper be used, with minor modifications, to model the reference patterns observed in a different system?
- How closely do the locality properties of the driver correspond to the locality observed in actual traces?

6. Summary

This paper describes the design of a synthetic driver that models short-term locality in file reference patterns. It also reports experimental measurements made on an actual file system, and uses these observations to derive parameters for the driver.

The approach taken here has been to develop a microscopic locality model based on macroscopic observations

[3]This research is being done with Prof. Richard Snodgrass, in the Department of Computer Science at the University of North Carolina.

of a file system. Three file classes (System, Temporary, and User) with strikingly different access characteristics have been identified, and locality models have been developed for each class. Simulations have been performed using the driver, and work is in progress to calibrate it using actual traces.

Acknowledgements

It is a pleasure to express my appreciation to the individuals who have helped me in various phases of the work presented here. Ed Dehart wrote the PDP-10 software to monitor interarrival times of I/O requests. George Robertson and Howard Wactlar were helpful in discussing the design of the experiments. The conference referees offered helpful hints to make the paper more readable. Richard Snodgrass critiqued a close-to-final draft and offered valuable suggestions for its improvement. As always, the paper is the better for his scrutiny.

References

[1] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M.
The Design of a Network-Based Central File System.
Technical Report CMU-CS-80-134, Computer Science Department, Carnegie-Mellon University, August, 1980.

[2] Howard, J.H., King, D.J., MacDonald, D.B., Satyanarayanan, M., Spector, A.Z., West, M.W., Young, M.
The ITC File System Design.
Internal Working Document, Information Technology Center, Carnegie-Mellon University, September 1983.

[3] Lawrie, D.H., Randal, J.M., and Barton, R.R.
Experiments with Automatic File Migration.
Computer 15(7):45-55, July, 1982.

[4] Organick, E.I.
The MULTICS System.
MIT Press, Cambridge, MA, 1972.

[5] Revelle, R.
An Empirical Study of File Reference Patterns.
Technical Report RJ 1557, IBM Research, San Jose, April, 1975.

[6] Satyanarayanan, M.
A Study of File Sizes and Functional Lifetimes.
In *Proceedings of the Eighth Symposium on Operating System Principles.* Asilomar, CA, December, 1981.

[7] Satyanarayanan, M.
A Methodology for Modelling Storage Systems and its Application to a Network File System.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, March, 1983.

[8] Smith, A.J.
Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms.
IEEE Transactions on Software Engineering SE-7(4):403-417, July, 1981.

[9] Smith, A.J.
Long Term File Migration: Development and Evaluation of Algorithms.
Communications of the ACM 24(8):521-532, August, 1981.

[10] Stritter, E.P.
File Migration.
PhD thesis, Stanford University, March, 1977.
Stan-CS-77-594.

[11] Wilk, M.B., and Gnanadesikan, R.
Probability Plotting Methods for the Analysis of Data.
Biometrika 55(1):1-17, 1968.

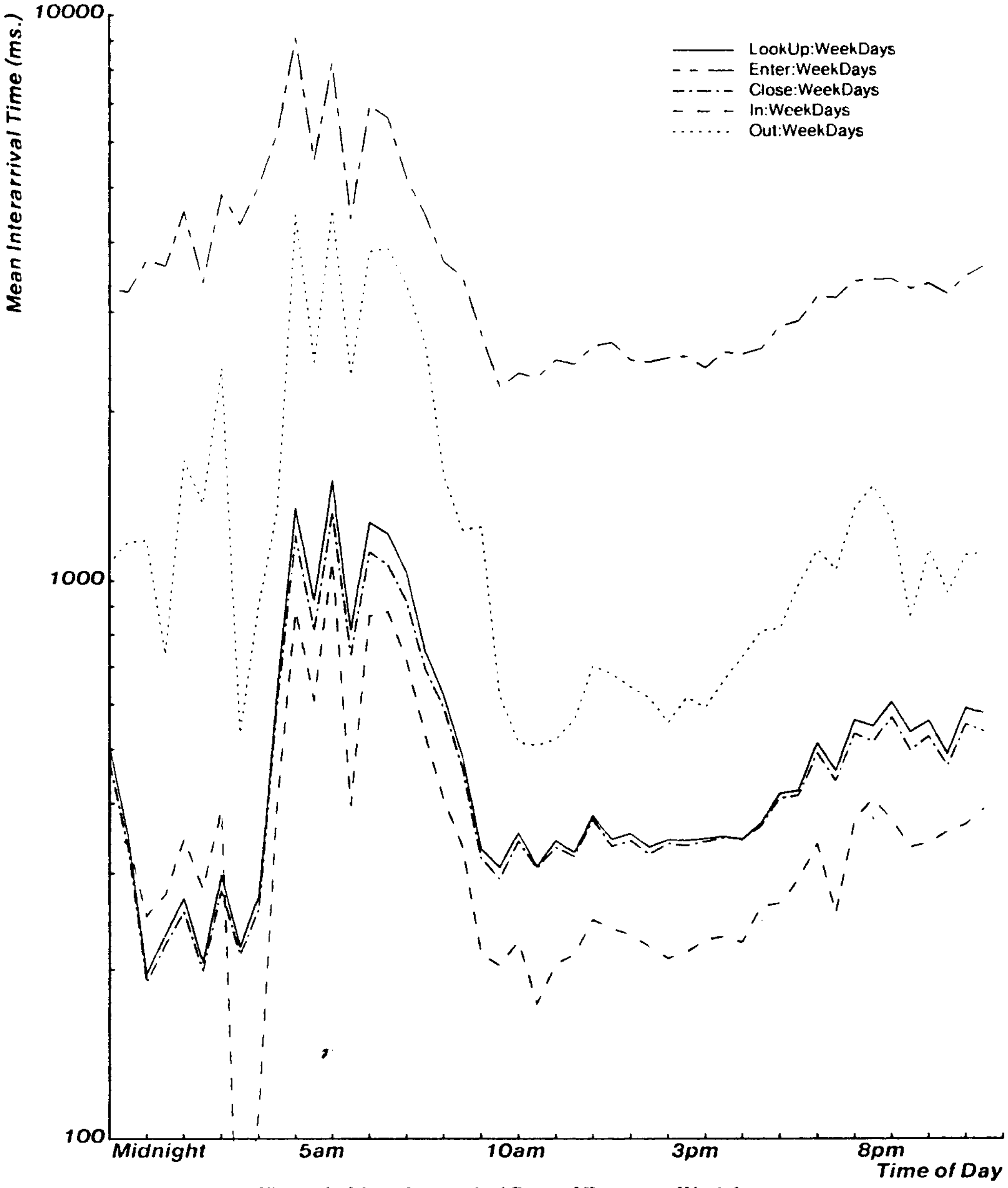


Figure 6: Mean Interarrival Rate of Events on Weekdays

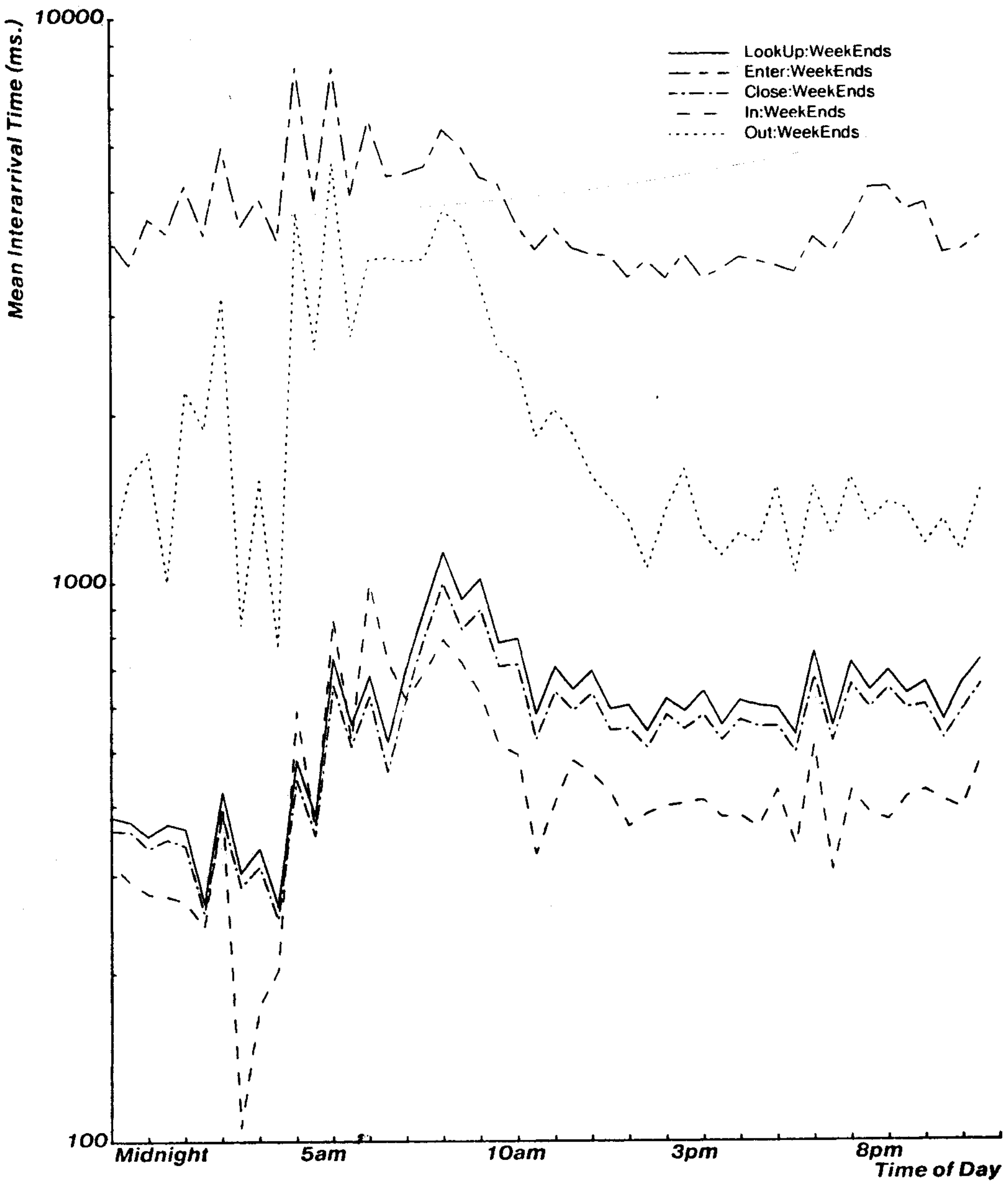


Figure 7: Mean Interarrival Rate of Events on Weekends

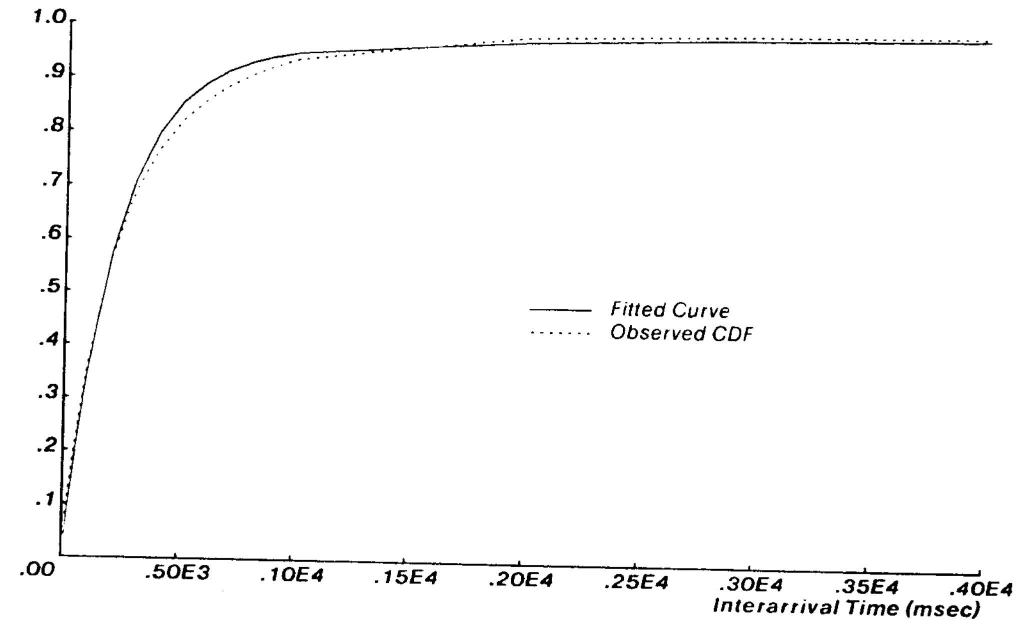


Figure 8: Fit to Interarrival Times of Lookups (Peak Period)

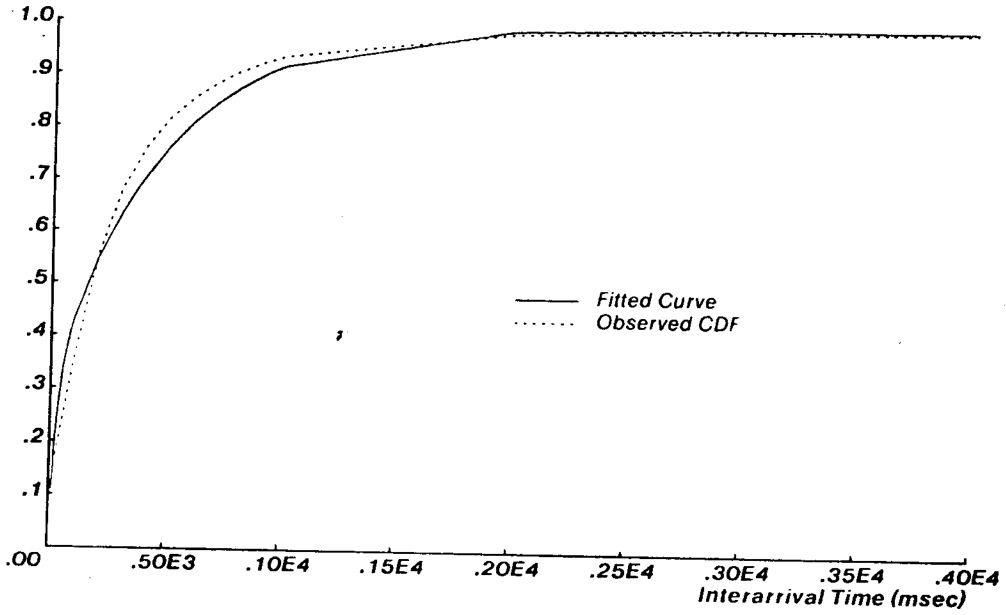
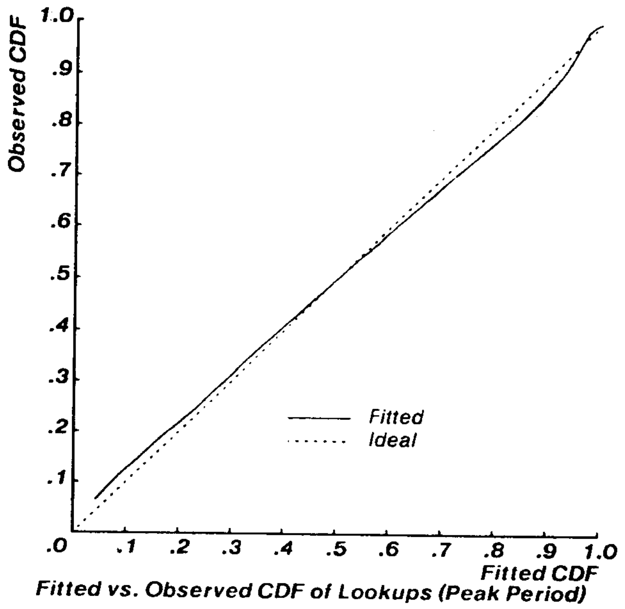


Figure 9: Fit to Interarrival Times of Closes (Peak Period)



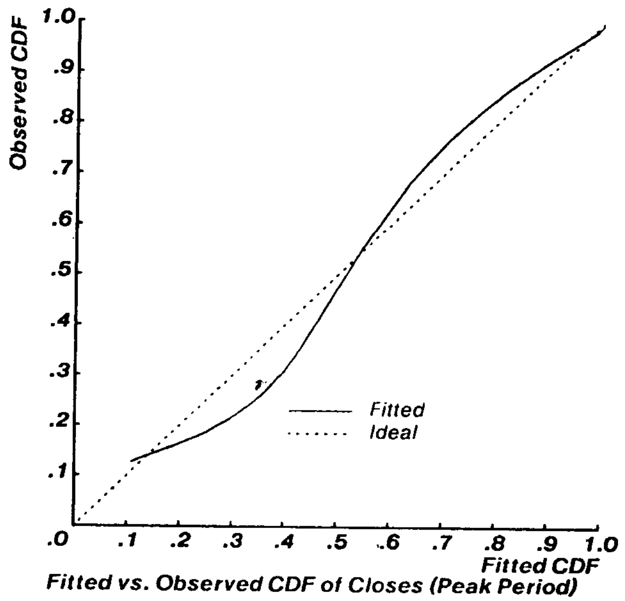


Figure 10: P-P plots of Best and Worst Fits