Requirements of I/O Systems for Parallel Machines: An Application-driven Study *

Mustafa Uysal   Anurag Acharya   Joel Saltz
Department of Computer Science
University of Maryland, College Park 20742
{uysal,acha,saltz}@cs.umd.edu

Abstract

I/O-intensive parallel programs have emerged as one of the leading consumers of cycles on parallel machines. This change has been driven by two trends. First, parallel scientific applications are being used to process larger datasets that do not fit in memory. Second, a large number of parallel machines are being used for non-scientific applications. Efficient execution of these applications requires high-performance I/O systems which have been designed to meet their I/O requirements. In this paper, we examine the I/O requirements for data-intensive parallel applications and the implications of these requirements for the design of I/O systems for parallel machines. We attempt to answer the following questions. First, what is the steady-state as well peak I/O rate required? Second, what spatial patterns, if any, occur in the sequence of I/O requests for individual applications? Third, what is the degree of intra-processor and inter-processor locality in I/O accesses? Fourth, does the application structure allow programmers to disclose future I/O requests to the I/O system? Fifth, what patterns, if any, exist in the sequence of inter-arrival times of I/O requests? To address these questions, we have analyzed I/O request traces for a diverse set of I/O-intensive parallel applications. This set includes seven scientific applications and four non-scientific applications.

1 Introduction

Until recently, most applications developed for parallel machines avoided I/O as much as possible (distributed databases have been a notable exception). Typical parallel applications (usually scientific programs) would perform I/O only at the beginning and at the end of execution with the possible exception of infrequent checkpoints. This has been changing: I/O-intensive parallel programs have emerged as one of the leading consumers of cycles on parallel machines. This change has been driven by two trends. First, parallel scientific applications are being used to process larger datasets that do not fit in memory [4, 10, 15, 37, 45]. Second, a large number of parallel machines are being used for non-scientific applications, for example databases [21, 32, 44], data mining [2, 3, 16], web servers for busy web sites (e.g. Altavista and NCSA [23]).

Efficient execution of these applications requires high-performance I/O systems which need to be designed to meet application I/O requirements. The I/O requirements of I/O-intensive parallel applications are likely to be significantly different from the I/O requirements of Unix/Office applications which have driven the development of most distributed file systems [19, 40, 41].

In this paper, we examine the I/O requirements associated with I/O-intensive parallel applications and examine the implications of these requirements on the design of I/O systems for parallel machines. We attempt to answer the following questions:

*This research was supported by ARPA under contract DABT63-94-C-0049, Caltech subcontract 9503 and by NSF under contract ASC9318183 and under CISE Institutional Infrastructure Award CDA9401151

1. What is the steady-state and peak I/O rates required by the application? These rates indicate how aggressive the I/O system needs to be.

2. What is the degree of intra-processor and inter-processor locality in I/O accesses? These measures indicate whether caching previously accessed data is likely to improve performance and if so, where should the cache(s) be placed. This information is also useful to understand the impact of alternative disk placements on application performance. For example, if all processors in an application access only their own partition, the use of private disks instead of shared disks can be expected to reduce communication requirements; on the other hand, if the data in a partition is written by the owning processor but read by all processors, the use of shared disks might be appropriate.

3. What are the spatial patterns of I/O requests? In particular, what are the common request sizes and whether I/O requests are sequential.

4. Does the application structure allow programmers to disclose future I/O requests to the I/O system? If this is indeed the case, it would provide an opportunity for the I/O system to improve the utilization of the storage devices as well as reduce the latency of I/O requests [24, 35].

5. Finally, are there patterns in the inter-arrival times of I/O requests? This information would allow the I/O system to better schedule prefetches for I/O requests from individual applications as well as from multiple applications.

To address these questions, we have analyzed I/O request traces for a diverse set of I/O-intensive parallel applications. This set includes three parallel scientific applications and four non-scientific applications (including IBM's DB2 Parallel Edition, datamining, a parallel web server and parallel textual search). These applications have been tuned, to various degrees, to achieve good I/O performance. We believe that this is important as studying applications which have not been tuned for I/O performance can lead to misleading conclusions [1]. We ran these applications on an IBM SP-2 and captured the I/O requests using the `trace` facility provided by AIX 4.1. In addition, we have acquired the I/O request traces made available by the Pablo group at the University of Illinois, Urbana-Champaign [34]. These traces correspond to four parallel scientific applications from several domains. We included these traces in our analysis to avoid bias due to choice of application domain.

Previous research on the characterization the I/O requirements of parallel applications has focused exclusively on scientific applications and has taken one of four approaches. French et al [17], Nitzberg [31] and Bordawekar et al [8] used synthetic benchmarks that are intended to emulate what the authors felt are common I/O behaviors. This approach provides information about the I/O capabilities of the machine that the experiments are being run on; does not provide information about application requirements. Kotz et al [25, 36] traced I/O requests from a large number of applications at two supercomputing sites. This approach is similar, in spirit, to the well-known studies of distributed file-systems done at Berkeley [7, 33] and Carnegie Mellon [29]. This approach provides useful information about the manner in which the parallel machines and their I/O systems are used, it does not distinguish between applications for which I/O performance is critical and applications for which I/O performance is not important. Examples of the latter include programs being debugged, programs that are compute-bound or communication-bound or programs that run for relatively short periods. Acharya et al[1], Karpovich et al [22] and Smirni et al[42] have looked at individual I/O-intensive applications to determine what is needed to make them run fast. Many of the conclusions reached by these studies are useful for designing I/O systems of parallel machines. However, these studies were limited by the small number of applications each of them studied. In particular, none of them included non-scientific applications. Finally, Cypher et al[14], Reddy et al[38] and Reed et al [12, 43] analyzed the behavior of groups of parallel applications that perform I/O. Cypher et al determined the steady-state I/O rate for a set of eight programs. This helps answer one part of one of the questions mentioned above. Reddy et al studied parallel versions of the Perfect Club benchmarks and found that file-accesses were sequential. The I/O for these benchmarks was, however, not parallelized. The studies conducted by Reed et al were the most thorough and are the closest to our work. However, they did not examine several of the questions mentioned above and they did not include non-scientific applications. We believe that a significant fraction of parallel machines are used for data-intensive non-scientific applications and that studies that are used to establish requirements for I/O systems for parallel machines should include such applications.

We first describe the applications studied in this paper. We describe the I/O request traces and how they were obtained. We then attempt to answer each question mentioned above using these traces. Finally, we discuss the conclusions and compare them with the conclusions reached by other studies.

2 Application suite and I/O traces

For this study, we selected a suite of eleven applications. Our goal in the selection process was to achieve diversity in application domain as well as application structure. We believe we have been fairly successful in this endeavor. Of the eleven applications, seven are scientific and four are non-scientific. The domains of scientific applications include remote-sensing, linear algebra, electron scattering, rendering planetary pictures, quantum chemistry, and radar imaging. The non-scientific applications include a relational database, data-mining, a parallel web server and textual search. These applications have a variety of application structures and communication patterns. Furthermore, the applications (or the I/O request traces) have been acquired from a variety of sources.

We first briefly describe the applications and the datasets. We then describe what information is captured in the I/O request traces. Finally, we describe how the I/O request traces were captured.

2.1 Applications we traced

We traced a total of seven applications, four non-scientific and three scientific. We refer to these groups as *non-scientific* and *scientific-1*. These applications were run on eight processors of an IBM SP-2 which has multiple disks attached to each processor. We used the AIX `trace` utility to trace I/O-related system calls (open, close, read, write and seek). We also captured a trace of the message-passing activity and context-switches. This allowed us to accurately compute the inter-arrival times for I/O requests and to better understand the application behavior.

DB2 Parallel Edition: this is the commercial-grade parallel RDBMS from IBM [20]. It partitions large relations using one of the attributes or a set of the attributes. To drive this application, we constructed a simulated database containing records for one year's operation of a large department store. It includes records for one million customers, 100 thousand products, 10 thousand employees, 200 departments and 100 million transactions. DB2 uses multiple processes per processor to implement multiple threads of control. These processes are used for performing different database tasks as well as to implement asynchronous I/O. The total database size was 5.2 GB and it was stored in 831 files (including the indices). We ran five consecutive queries against this database. These queries perform complex join, set, and aggregate operations on indexed and non-indexed relations. For these queries, DB2 uses a *directed outer-table join* strategy in which rows of the outer table is hashed (based on the partitioning key) to the node where join is performed [20]. The total execution time for all five queries was 7,688 seconds.

Data-mining: this application tries to extract association rules from retail data [27] – in particular, buying patterns that characterize the shopping behavior of retail customers. This application performs I/O using synchronous `read` operations. We have used a database consisting of 50 million transactions, with an average transaction size of 10 items and maximal potentially frequent set size of 3 (see [3, 27] for details). The dataset size for this program was 4 GB and was partitioned into 8 files, one per processor. This application ran for 679 seconds.

Parallel Web Server: this application uses a parallel web-server based on the round-robin DNS scheme described by Katz et al [23]. Similar schemes are used by most busy commercial web sites. We used the Apache 1.2 server [6] as the base web server which is replicated on the participating hosts. This application uses multiple processes per processor to implement multiple threads of control. Over the period of a day, it creates a large number of processes (about 2000), most of which terminate relatively soon. At any given time, there are no more than ten active processes. We used NASA Kennedy Space Center's *httpd* logs for August 1995 to create the document hierarchy as well as to drive the application. The size of the dataset served was 524 MB which is stored in 13,457 files. To account for the explosive growth in web accesses since 1995, we collapsed the request stream for the entire month to a single day - taking care to preserve the time-of-day variations. The experiment was run over a 24 hour period and a total of about 1.5 million HTTP requests were served, delivering over 36 GB of data.

Parallel text search: for this application, we used a modified parallel version of the `agrep` program from University of Arizona which is capable of partial match and approximate searches [46]. Our version can operate in one of two modes. In the first mode, the list of files to be searched is partitioned, round-robin, among the processors; in the second mode, each input file is block-partitioned among the processors. This application performs I/O using synchronous `read` operations. To drive this application, we used the `/users` document hierarchy on the University of Maryland web server and a complex pattern including wildcards, disjunction and substitution. The hierarchy contained 475 MB in 18,655 files. We searched the dataset twice: once as a hierarchy (using `pgrep` in the first mode) and as a single tar file (using `pgrep` in the second mode). The execution times were 94 seconds for the hierarchy and 41 seconds for the tarfile, respectively.

Titan: is a parallel scientific database for remote-sensing data [9]. In addition to retrieving data, Titan performs navigation, correction and composition to generate a land-cover image of the region of interest. The region of interest is specified by temporal and geographical (latitude-longitude) ranges. Titan contains two months of data from the NOAA-7 satellite. This application performs I/O using the asynchronous `lio_listio` operations. It throttles the number of requests to keep the number of outstanding I/O requests under a user-specified limit. The total database size is 30 GB which spans 60 data-files. To drive this application, we used three queries which span a 30-day period over the entire globe, North America and Asia respectively (see [9] for details). The total execution time for these three queries was 583 seconds.

LU decomposition: this application computes the dense LU decomposition of an out-of-core matrix [18]. This application performs I/O using synchronous `read/write` operations. To drive this application, we used an 8192×8192 double precision matrix (total size 536 MB) with a slab size of 64 columns. The dataset was stored in 8 files, one per processor, and the total execution time was 2,987 seconds.

Sparse Cholesky: this application computes Cholesky decomposition for sparse, symmetric positive-definite matrices [1]. It stores the sparse matrix as panels (instead of blocks). This application performs I/O using synchronous `read/write` operations. To drive this application, we used the *skirt* matrix from NASA that contains over 45 million double-precision nonzeros and 45,361 columns for a total of 437 MB. The dataset was stored in 8 files, one per processor, and the total execution time was 2,696 seconds.

2.2 Application traces we acquired

We acquired four I/O request traces from the web repository provided by the Pablo research group at the University of Illinois [34]. These traces were generated using the Pablo instrumentation software [39] on the Intel Paragon at California Institute of Technology. We refer to these applications as the *scientific-2* group of applications.

Rendering: this application uses a parallel ray-tracing algorithm to combine planetary imagery with topographic information to create a sequence of three-dimensional perspective views of planetary surfaces [12]. This application performs I/O using asynchronous `iread` operations. For this study, we selected the trace corresponding to a Mars flyby. The total dataset size is about 1 GB; the number of files is about 110. This trace was collected from a 128-processor run; the total execution time was 470 seconds.

Ab-initio quantum chemistry (Hartree-Fock): this application calculates the non-relativistic interactions among atomic nuclei and electrons. It consists of three parts - the first two parts initialize out-of-core data that is used in the third part which dominates the I/O requirements [12]. This application performs I/O using synchronous `read/write` operations. For this study, we selected the trace corresponding to the third part working on a 16-atom dataset. The total I/O volume is 4 GB, the total number of files is 130. This trace was collected from a 128 processor run; the total execution time was 1008 seconds.

Electron scattering: this application solves electron-scattering problems using the Schwinger multichannel method [12]. This application performs I/O using synchronous `read/write` operations. For this study, we selected the trace corresponding to a dataset with 13 channels (run 8 on the Pablo web-site). This trace was collected from a 256-processor run; the total execution time was about 6000 seconds.

Synthetic Aperture Radar: this application processes four channels of SAR data to compute high-resolution ground surface images [13]. This application performs I/O using synchronous `read/write` operations. The total input dataset was about 536 MB in four equally-sized files (there are two other small files); the total output was about 51 MB in four equally-sized files. This trace was collected from a 256-processor run; the total execution time was about 114 seconds.

3 I/O demand

Qualitatively, the I/O demand of an application can be characterized as the I/O-to-computation ratio – the amount of I/O required per unit processing time. We computed two measures to quantify the I/O demand of an application: the *peak instantaneous I/O rate* and the *steady-state I/O rate*. The instantaneous I/O rate is computed as the amount of data moved for fixed processing time periods; peak instantaneous rate is the maximum instantaneous I/O rate over the entire execution. The steady-state I/O rate is computed as the ratio of the total data moved and the total processing time. Note that, for both of these metrics, the denominator includes only the processing time and does not include the time spent performing I/O. The peak instantaneous I/O rate of an application is an upper bound on the desired I/O bandwidth of the machine the application is run on. Given sufficient memory for prefetching and information about the sequence of I/O requests the application will make, a machine with a total I/O bandwidth greater than the peak instantaneous I/O rate can completely eliminate the time spent waiting for I/O. The steady-state I/O rate of an application is the corresponding lower bound.

Since the cost of fulfilling an I/O request usually consists of a fixed cost per request plus a variable cost roughly proportional to size of the request, the rate at which an application issues I/O requests is also important. To quantify the rate at which applications issue requests, we computed the *peak instantaneous request rate* and the *steady-state request rate*, which are analogous to the I/O rate metrics described above.

Table 1 presents the I/O and request rates for all the applications. We make three observations. First, these applications present a very small I/O demand for writes; the I/O and request rates for writes are uniformly low. This includes out-of-core scientific applications like `lu`, `cholesky` and `escat`. This is because each block that is both read and written is written once but read multiple times. This is not to argue that no application needs high write bandwidth. Instead, we believe that for a wide variety of applications, write bandwidth is not a major concern. Second, non-scientific applications require far more I/O bandwidth. We attribute this to the fact that most non-scientific applications (including these applications) perform relatively simple operations on the data – check a predicate, look for string matches, transfer data over the network. In contrast, scientific applications, usually perform relatively complex operations on the data they process – e.g., navigation, block multiplication, FFT. Third, non-scientific applications make a very large number of requests – the minimum *steady-state* request rate for non-scientific applications is over 100 requests/s. Within non-scientific applications, the applications that process a large number of files, `db2`, `pgrep-dir` and `web-server`, have extremely large request rates. This can be partially attributed to the fact that many of these files are small and data transferred per request is small. The files for `db2` are not small but the current implementation has been hardcoded to read data in small (4 KB) chunks which results in the generation of a large number of requests.

Analyzing the inter-arrival times of I/O requests, we found that I/O is bursty for all the scientific applications. One common cause of burstiness is the read-process loop structure used in many of these programs. Such loops use a bursty sequence of requests to fill a set of buffers which are then processed without intervening I/O requests. Other causes include processing that takes time non-linear in the size of data (e.g. `lu`, `cholesky`). Of the non-scientific applications, I/O is bursty for only two (`pgrep-dir` and `web-server`). For the other three, I/O demand is more or less steady. The steady demand for `db2` can be partially attributed to the one-query-at-a-time processing. We expect that in a multi-query scenario, I/O demand for `db2` would be bursty.

Given the high steady-state and peak demands for reads, we conclude that I/O systems should be aggressive on optimizing data retrieval; we expect that simple write-behind policies would be effective given the low demand for writing data. Furthermore, given the bursty nature of I/O demand for most applications, we believe that the peak instantaneous I/O rate should be preferred to steady-state I/O rate as the metric of choice for designing I/O systems for parallel machines. Given the heterogeneity of the machines used to gather the traces, we hesitate to estimate a single value for desired I/O bandwidth. However, we do believe that an I/O system that delivers a read bandwidth of about 19 MB/s per-processor ($\lceil\frac{147.6}{8}\rceil$) should meet the requirements of even the most demanding applications in the near future and that a read bandwidth of 10 MB/s per-processor ($\approx \lceil\frac{63.1}{8}\rceil$) should be adequate for most applications. Note that this measures are dependent on the time taken to execute the code between successive requests and as such are sensitive to several factors including processor speed, memory bandwidth and network bandwidth. However, since neither `pgrep` nor `dmine` performs communication in between I/O requests, these are largely dependent on processor speed and memory bandwidth.

4 Request size and sequentiality

One of the key observations that has been repeatedly used in the design of file-systems is that most files are accessed sequentially. All widely-used file-systems available on Unix platforms implement some form of read-ahead to take advantage of this access pattern. The validity of this observation for parallel platforms has, however, been in question. In their analysis of long-term traces from two parallel machines, Kotz et al [25, 36] found that on those machines, in addition to sequentially accessed files, many files were accessed in a *strided* or *nested-strided* pattern. In addition, they found that a significant number of the requests were small (e.g., in the iPSC/860 study, 96% of the reads were smaller than 4 KB [25]). A

Application	I/O Transfer Rate (MB/s) Read	Write	Overall	Request Rate (Req/s) Read	Write	Overall
db2	20.6 (5.2)	0.7 (0.4)	21.1 (5.3)	5,248 (1,308)	164 (100)	5,348 (1,309)
dmine	63.1 (21.7)	-	63.1 (21.7)	493 (170)	-	493 (170)
web-server	25.1 (11.4)	0.3 (0.001)	25.1 (11.4)	1,527 (696)	161 (9)	1,533 (705)
pgrep-dir	123.6 (75.4)	-	123.6 (75.4)	2,044 (669)	-	2,044 (669)
pgrep-tar	147.6 (129.9)	-	147.6 (129.9)	121 (107)	-	121 (107)
lu	10.4 (7.6)	3.8 (0.2)	11.5 (7.8)	51 (22)	195 (8)	195 (30)
cholesky	8.2 (0.2)	4.2 (0.04)	8.2 (0.2)	54 (9)	28 (8)	56 (17)
titan	31.1 (2.8)	-	31.1 (2.8)	174 (20)	-	174 (20)
render	6.3 (2.0)	0.9 (0.2)	7.3 (2.3)	20 (2)	3 (1)	20 (3)
hartree	16.6 (4.1)	0.3 (0.005)	16.6 (4.1)	208 (52)	19 (1)	209 (53)
escat	49.1 (6.5)	0.8 (0.1)	49.1 (7.5)	708 (109)	256 (119)	769 (116)
sar	33.5 (3.9)	3.9 (0.5)	33.5 (4.2)	23 (4)	3 (1)	24 (4)

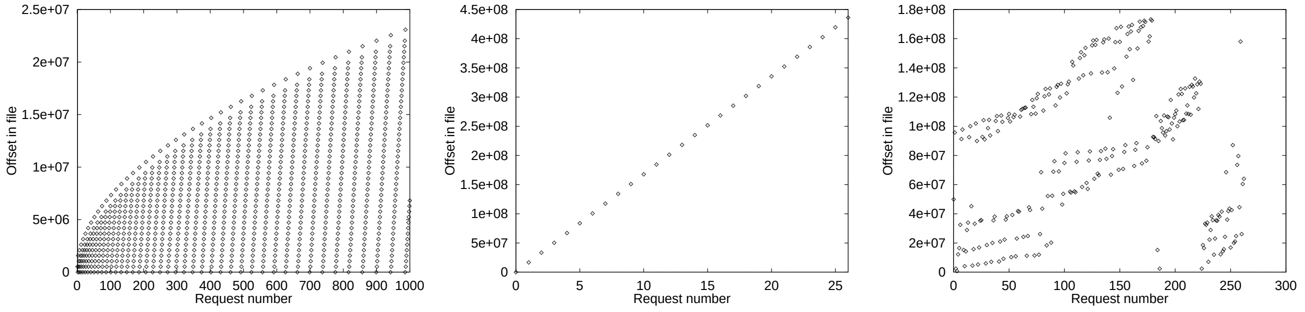
Table 1: Peak instantaneous and steady-state I/O and request rates for different applications. The unparenthesized number in every cell is the peak instantaneous rate; the parenthesized number is the corresponding steady-state rate. The instantaneous rates were computed based on one-second time intervals. The first and second group of applications were run on 8 processors of an SP-2, `hartree` and `render` were run on 128 processors of a Paragon and `escat` and `sar` were run on 256 processors of a Paragon.

later paper by Acharya et al [1] studied several I/O-intensive parallel applications and found that nested-strided patterns and small request sizes in those applications were caused by placement of I/O operations in inner-loops. Loop reordering following by simple coalescing allowed I/O operations to be placed in outer loops and greatly increased the request size. In this section, we provide information about the request size distribution and the access patterns for all the applications we studied. We hope that this information will help address the controversy.

The applications examined in this study display five distinct read request patterns: (1) `single-scan`, where each file is read once without seeks; (2) `multi-scan`, where each file is read sequentially but is read more than one time; (3) `triangle-scan`, where the file is read in gradually increasing sweeps starting at the beginning of the file; (4) `strided-scan`, where each file is read in a strided fashion, there being a seek between successive read requests; and (5) `oscillating-scan`, where the accesses oscillate between two scans. The first two patterns are obvious; Figure 1 illustrates the latter three.

Four applications, `sar`, `render`, `dmine` and `db2`, had the single-scan access pattern. Two applications, `hartree` and `web-server`, had multi-scan access patterns. The two linear algebra applications, `lu` and `cholesky` had triangle-scan access patterns. Finally, strided-scan and oscillating-scan occurred for one application each (`escat` and `titan` respectively). Note that these patterns correspond to read requests from individual processors - i.e., they are per-processor patterns.

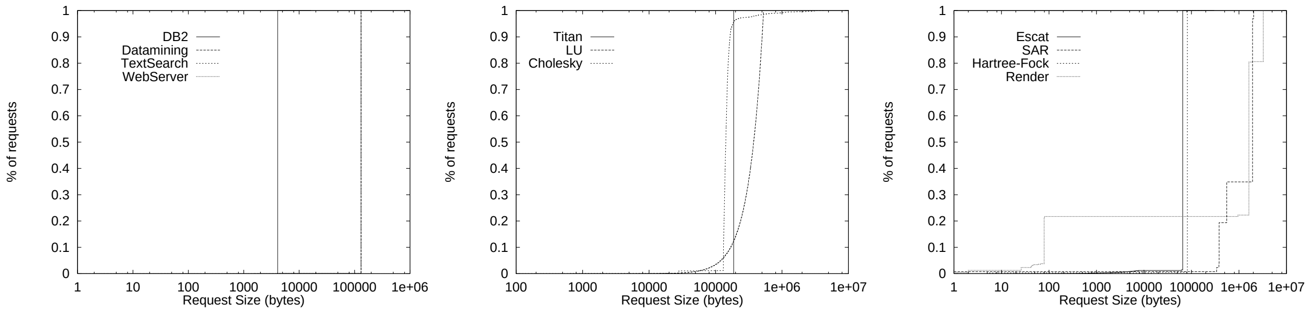
Figure 2 presents the cumulative distributions of the read and write requests for the three application classes. There is no graph for write request size for non-scientific applications – these applications perform few or no writes. We make three observations. First, except for a small number of 100-byte requests for `render`, read requests for all applications are large. All the non-scientific applications used 128 KB read requests except `db2` which used 4 KB read requests; all the scientific applications issued read requests larger than 100 KB. The 4 KB request size observed for DB2 is not inherent in the design of the system – the DB2 manual indicates that the request size can be 4,8,16 or 32 KB. Second, most write requests are also large but not as large as the read requests, especially for the Scientific-2 applications. Recall, however,



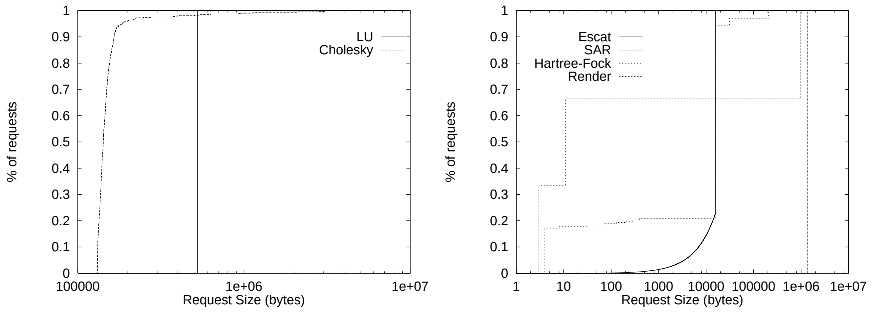
(a) Triangle scan (`lu`) (b) Strided scan (`escat`) (c) Oscillating scan (`titan`)

Figure 1: Examples of file access patterns. Graph (a) displays only the first 1000 read requests for `lu` for clarity. The same pattern persists through all requests.

from Table 1 that for these applications, the number of read requests is several times the number of write requests.



(a) Non-Scientific read requests (b) Scientific-1 read requests (c) Scientific-2 read requests



(d) Scientific-1 write requests (e) Scientific-2 write requests

Figure 2: Cumulative distributions of I/O request sizes. Note that the x-axis in all graphs is log-scaled.

Based primarily on the studies by Kotz et al, recent proposals for file-system interfaces have included support for scatter-gather requests with small component requests and/or for *strided* and *nested-strided* requests [11, 26, 30]. We note that for the variety of applications examined in this study, request sizes were usually large and the access patterns were both simpler and more complex than nested strides. We believe that this reflects the fact that many applications are not structured as regular nested-loops with I/O in the inner loop – which could generate the regular small-request scatter-gather or the nested-strided patterns.

5 Intra-processor and inter-processor locality

Another observation that has driven the design of file-systems is the presence of temporal locality in accesses to files and disk blocks. All widely-used file-systems available on Unix platforms implement some form of caching to take advantage of this. In a parallel machine, file caches can be placed at the servers, at the clients or both. The impact of these caches depend on inter-processor and intra-processor locality in I/O requests. Server caches are useful if multiple clients access the same data; client caches are useful if individual processors repeatedly access the same data. Information about inter-processor locality is also useful to determine placement of disks in a parallel machine. In the absence of inter-processor locality, it is usually better to attach disks to individual processors. Since the data is perfectly partitioned, this would allow to the application to achieve a high aggregate I/O bandwidth. In the presence of significant inter-processor locality, it might be better to share the disks by placing in a central location. In this section, we describe the locality characteristics of the applications we examined in this study and the implications of these characteristics for placement of file caches and disks.

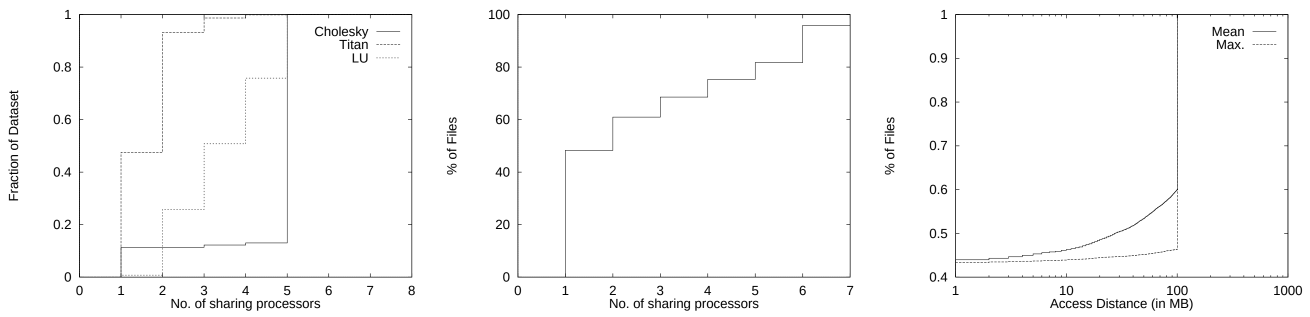
For inter-processor locality, we found that: (1) there is little or no write-sharing between processors and (2) disk-block-level read-sharing occurs only for the Scientific-1 applications – `lu`, `cholesky` and `titan`. Figure 3(a) presents a quantitative measure of the sharing in these applications. It shows that 50% of the dataset is read-shared between 2-5 processors (recall that these experiments were run on an 8-processor machine). In addition, multiple processors running `web-server` share whole files. Figure 3(b) plots the percentage of files that are shared by more than one processors. It shows that more than 50% of files are shared by at least two processors and 25% of the files are shared by 5 or more processors. However, `web-server` has little *concurrent* read-sharing; less than 1% of the files are read by multiple processors at the same time. For the Scientific-1 applications, accesses to the shared data are *concurrent* and explicitly synchronized by the applications.

Data-sets for the remaining applications are perfectly partitioned – each processor accesses data only in its own partition. In particular, the data-sets for all non-scientific applications except `web-server` are perfectly partitioned. This suggests that (1) local disks are an important component of I/O systems for parallel machines and (2) server caches are useful for a small subset of applications.

For intra-processor locality, we found that: (1) temporal locality between read requests exists for applications with triangle-scan and multi-scan access patterns – `lu`, `cholesky`, `hartree` and `web-server`; (2) temporal locality between write and read requests exists for one of the applications (`escat`). Rest of the applications read the data once.

For applications with a triangle-scan access pattern, i.e., `lu` and `cholesky`, the frequency with which data is accessed depends on its position in the file. Data at the beginning of the file is accessed most frequently; data at the end of the file is accessed infrequently (see Figure 1). If the client-cache is not large enough to hold the entire data-set, or a large subset thereof, an LRU-based caching policy would provide little benefit. An alternative cache replacement policy that uses the information about the access pattern is likely to work better. This policy would partition the cache into two segments. The first segment would hold as much data from the beginning of the file as possible; once installed, data in this segment would not be replaced. The second segment would hold data from the rest of the file and would use a MRU replacement policy. [1]

[1] We would like to point out that attractive as this policy appears, we have not yet evaluated its impact on a real implementation.



(a) Data sharing in scientific-1 appls (b) File sharing in `web-server` (c) Locality in `web-server`

Figure 3: Inter-processor file and data sharing, intra-processor locality.

For applications with a multi-scan access pattern, the utility of client-caching depends on the length of the scan, the number of times it is repeated and the size of the cache. If the length of the scan is larger than the cache size, caching provides no advantage. Of the two multi-scan applications, `hartree` is simpler to analyze as the length of its scan is fixed (about 6 MB) and the number of repetitions is fixed (four). The utility of client-caching `web-server` is harder to analyze for two reasons: (1) different files have different lengths and (2) different files are accessed at widely varying frequencies (the average number of requests for a file over the experiment was 77 whereas the maximum number of requests was 19450). To estimate the potential impact of client-caches for `web-server`, we computed the *access-distance* metric which is defined as the average number of *unique* bytes accessed between two successive accesses to the same file. If the same file is accessed more than once, this metric counts it only once (assuming it would be cached for the second and later accesses). This metric provides an indication of the size of cache needed assuming an LRU replacement policy. Figure 3(c) plots the percentage of files with an access-distance less than 100 MB. We note that 40% of all files have an access-distance less than 1 MB. The average access-distance for all files was 20.9 MB (These findings are consistent with the previous research on web workloads [5]). This indicates that client-caching with reasonable-sized caches with LRU replacement are likely to be useful for `web-server`.

Finally, for `escat`, there is read-after-write locality between the two phases. Each processor writes intermediate data to a private file in the earlier phase which it reads back in the later phase [12].

From the variety of caching policies required, we conclude that no single caching policy is best and that, ideally, I/O systems for parallel machines should allow the application to control or specify the caching policy.

6 Foreknowledge of I/O requests

Pre-fetching is one of the primary techniques used to reduce or eliminate I/O latency for read requests. A key requirement for successful pre-fetching is information about future I/O requests. Several techniques have been proposed for efficient pre-fetching (and caching) which assume that such information is available [24, 35]. It is conjectured that for most I/O-intensive applications, it is not difficult to acquire information about future I/O requests [1, 35]. Our experience with the applications examined in this study supports this conjecture. For all applications except `web-server`, it is not difficult to acquire precise information about future read requests. In this section, we briefly describe how this can be done for individual applications.

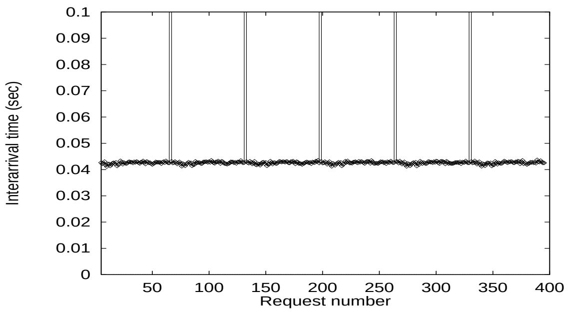
For all of applications examined in this study except `web-server`, the sequence of I/O requests can be determined soon after the execution begins. For some of the applications, `lu`, `render`, `hartree`, `escat`, `sar`, `dmine` and `pgrep`, the sequence of I/O requests can be determined using knowledge about the structure of the application and a few input parameters. For other applications, `db2`, `titan`, `cholesky`, information about the data is needed as well. For the database applications, the list of I/O requests and the order in which they will be issued is computed using indices that describe the contents and location of data. For `cholesky`, the sequence of I/O requests can be computed using the *elimination-tree* which describes the sparsity structure of the matrix to be factorized [1].

7 Possibility of predicting request inter-arrival times

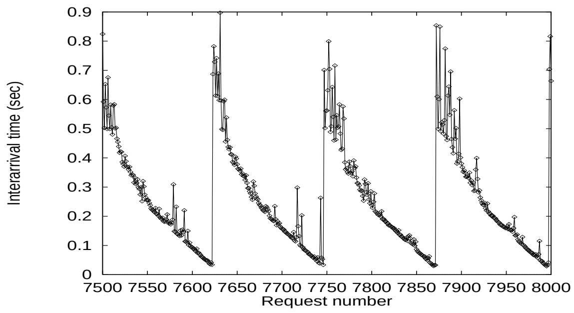
One of the problems with effective use of information about future I/O requests is that usually no information is available about the *time intervals* between successive read requests. In the absence of this information, the I/O system has to guesstimate how far ahead in the request sequence should it prefetch [1, 35]. Patterson et al [35] assume that the time between read requests is constant; Acharya et al[1] use application-specific heuristics to compute the prefetch horizon. The impact of this lacuna can be even larger for I/O systems that serve multiple applications as it is difficult to impose a reasonable total order on request sequences from different applications. In this section, we examine the possibility of prediction of inter-arrival times. We assume that information about future I/O requests is available (as discussed in section 6). We first examine the inter-arrival times with the goal of determining common patterns, if any. We would like to point out at the outset that since the execution time between successive requests depends on a complex set of factors (caching, memory bandwidth, context switch times, delays in inter-processor communication), patterns, if any, in inter-arrival times are likely to be "noisy" to varying degrees. The goal of our examination was to extract approximate patterns which could be of some help in determining the prefetch horizon (and order of prefetches for multiple applications). We then describe common application structures which could/would cause such patterns in inter-arrival times. Finally, we discuss a technique by which the I/O system might be able to estimate the inter-arrival times in an online fashion, with some help from the user (or the compiler).

We found three kinds of patterns in the inter-arrival times of read requests. For two applications, `dmine` and `pgrep-tar`, the inter-arrival time was (approximately) constant. For five applications, `db2`, `pgrep-dir`, `render`, `escat` and `hartree`, the inter-arrival time was piece-wise constant and repetitive. For example, see Figure 4 (a). In this case, the inter-arrival time is constant except for spikes at regular intervals (the spikes have been truncated to better display the constancy of the inter-arrival periods. For two applications, `lu` and `cholesky`, the inter-arrival time was piece-wise quadratic and repetitive. For example, see Figure 4 (b). Finally, for two applications, `titan` and `web-server`, the sequence of inter-arrival times had no discernible patterns.

The repetitive nature of the patterns suggests that it might be possible for the I/O system to determine the pattern during the first few repetitions and to use that information to estimate the inter-arrival times for the subsequent repetitions. This is similar to the *record-replay* technique used by Mukherjee et al[28] for determining the communication patterns for distributed shared memory machines. The primary issue that has to be taken care of is how to detect the end of a repetition. This problem can be solved if the sequence of I/O requests that occur in a repetition are disclosed in a single group. To explore how difficult it would be to do this, we came up with generalized loop structures that would generate the three observed patterns in inter-arrival times and tried to determine: (1) whether the observed patterns were indeed generated by loops similar to the postulated loop structures and (2) whether it is difficult to group the I/O requests that correspond to individual repetitions in the patterns. Figure 5 shows the generalized loop structures. For the applications that we had the source code for, `dmine`, `pgrep`, `lu`, `cholesky` and `hartree`, we were



(a) Piece-wise constant (`hartree`)



(b) Piece-wise quadratic (`lu`)

Figure 4: Examples of piece-wise constant and piece-wise quadratic patterns in inter-arrival times for read requests.

Constant

```
loop
  do I/O with fixed req size
  do processing linear in req. size
end loop
```

Piece-wise quadratic

```
loop
  do I/O
  processing quadratic in req. size
end loop
```

Piece-wise constant

```
loop
  loop
    do I/O with fixed req. size
    do processing linear in req. size
  end loop

  other processing
end loop
```

Figure 5: Loop structures corresponding to the observed patterns in inter-arrival times.

able to verify the correspondence between the structure of the code and the corresponding generalized loop structure. Given these loop structures, we believe it should not be difficult for a programmer (or a compiler) to group the disclosed I/O requests as desired.

8 Conclusions

In this section, we summarize our conclusions.

- Given the high steady-state and peak demands for read requests, I/O systems should be aggressive on optimizing data retrieval; we expect that simple write-behind policies would be effective given the low demand for writing data.

- For the current hardware, an I/O system that delivers a read bandwidth of about 19 MB/s per-processor should meet the requirements of even the most demanding applications and that a read bandwidth of 10 MB/s per-processor should be adequate for most applications. Note that these figures assume zero latency to the I/O system and a perfect interleaving of computation and I/O.

- For the variety of applications examined in this study, requests were usually large and the access patterns were both simpler and more complex than nested strides.

- There is little or no write-sharing between processors.
- Local disks are an important component of I/O systems for parallel machines.
- Different applications require different caching policies – in particular, they need different cache replacement policies and different cache placement (server/client/both) decisions. Ideally, I/O systems for parallel machines should allow the application to control or specify the caching policy.
- For many applications, the sequence of inter-arrival times between read requests can be described by relatively simple patterns. We have seen three patterns: constant, piece-wise constant and piece-wise quadratic. The repetitive nature of the patterns suggests that it might be possible for the I/O system to determine the pattern during the first few repetitions and to use that information to estimate the inter-arrival times for the subsequent repetitions. We are planning to explore this further.

Acknowledgments

We would like to heartily thank the members of the Pablo group for making their traces available over the web. We would like to thank Rusty Lusk for organizing the Scalable I/O code repository. We obtained our copy of LU from the repository. We would like to thank Andreas Mueller for the data-mining application and Wu and Manber for agrep. We would like to thank Chialin Chang for his help with Titan and Jim Humphries and the all of the staff at UMIACS for their help in the logistics of the tracing.

References

[1] A. Acharya et al. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the 4th IOPADS*, pages 15–27, Philadelphia, PA, May 1996.

[2] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–9, Dec 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of 20th Int'l Conf. on Very Large Databases (VLDB)*, Santiago, Chile, Sept. 1994.

[4] H. Allik and D. Moore. BBN corporation, personal communication, Sep 1994.

[5] V. Almedia, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of 4th Int'l Conf. Parallel and Distributed Systems*, pages 96–103, 1996.

[6] Apache 1.2 HTTP Server. *http://www.apache.org*, 1995.

[7] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 198–212, Oct 1991.

[8] R. Bordawekar, A. Choudhary, and J. Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–76, 1993.

[9] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a High-Performance Remote-Sensing Database. In *Proceedings of the 13th International Conference on Data Engineering*, page To appear, Apr 1997.

[10] Dense out-of-core linear solver in a finite element library. *http://www.comco.com/main/phlex/-ProPhlex.html*, 1996.

[11] P. Corbett et al. Proposal for a common parallel file system programming interface. *http://www.cs.arizona.edu/sio/api1.0.ps*, September 1996. Version 1.0.

[12] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proceedings of the Supercomputing '95*, San Diego, CA, December 1995.

[13] P. Crandall, A. Chien, and D. Reed. Input/Output characteristics of a synthetic aperture radar application. *http://www-pablo.cs.uiuc.edu/Projects/IO/sioDir/sar/sar-analysis.ps.Z*, 1995.

[14] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[15] Charbel Farhat. Large out-of-core calculation runs on the ibm sp-2. *NAS News*, 2(11), Jul-Aug 1995.

[16] A. Freitas and S. Lavington. Parallel data mining for very large relational databases. In *Proceedings of High-Performance Computing and Networking HPCN Europe*, pages 158–63, 1996.

[17] J. French, T. Pratt, and M. Das. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1-2):115–21, Jan-Feb 1993.

[18] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15(5), Sept. 1994.

[19] J. Howard, M. Kazar, , S. Menees, D. Nichols, M. Satyanaryanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb 1988.

[20] IBM Corporation. *IBM DATABASE 2 Parallel Edition for AIX - Administration Guide and Reference*.

[21] Informix - online extended parallel server. *http://www.informix.com*.

[22] J. Karpovich, J. French, and A. Grimshaw. High-performance access to radio astronomy data: A case study. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, pages 240–9, Sept 1994.

[23] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2):155–64, Nov 1994.

[24] T. Kimbrel, A. Tomkins, R. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 19–34, Oct 1996.

[25] D. Kotz and N. Nieuwejaar. File-system workload on a scientific multiprocessor. *IEEE Parallel & Distributed Technology*, 3(1):51–60, Spring 1995.

[26] MPI-IO: a parallel file I/O interface for MPI. *http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps*, April 1996.

[27] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, University of Maryland, College Park, August 1995.

[28] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Processing'95*, pages 68–79, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[29] L. Mummert and M. Satyanarayanan. Long-term distributed file reference tracing: Implementation and experience. *Software - Practice and Experience*, 26(6):705–36, June 1996.

[30] N. Nieuwejaar and D. Kotz. Low-level interfaces for high-level parallel I/O. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 9, pages 205–23. Kluwer Academic Publishers, 1996.

[31] B. Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, Dec 1992.

[32] Oracle 7 server - scalable parallel architecture for open data warehousing. *http://www.oracle.com*, Oct 1995.

[33] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems*, pages 15–24, Dec 1985.

[34] I/O request traces from ESCAT, Render, SAR and Hartree-Fock. *http://www-pablo.cs.uiuc.edu/-Projects/IO/io.html*, 1996.

[35] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zalenka. Informed prefetching and caching. In *Proc. of the 15th Symp. on Operating System Principles*, Dec. 1995.

[36] A. Purakayastha, C. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–72, April 1995.

[37] J. Qin, T. Agarwal, O. Storaasali, D. Nguyen, and M. Baddourah. Parallel-vector out-of-core solver for computational mechanics. In *Proceedings of the 2nd Symposium on Parallel Computational Methods for Large-scale Structural Analysis and Design*, Feb 1993.

[38] A. Reddy and P. Bannerjee. A study of I/O behavior of the Perfect benchmarks on a multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 312–21, 1990.

[39] D. Reed et al. Scalable performance analysis: The PABLO performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–13, Oct 1993.

[40] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb 1992.

[41] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–59, April 1990.

[42] E. Smirni, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O requirements of scientific applications: An evolutionary view. In *Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, Syracuse, NY, August 1996.

[43] E. Smirni and D. A. Reed. Workload characterization of input/output intensive parallel applications. In *Modelling Techniques and Tools for Computer Performance Evaluation*, June 1997. to appear.

[44] Sybase MPP: Parallel High Performance for Real World Workloads. *http://www.sybase.com*.

[45] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, May 1996.

[46] Sun Wu and Udi Manber. agrep - A fast approximate pattern-matching tool. In *USENIX Conference Proceedings*, pages 153–162, San Francisco, CA, Winter 1992. USENIX.