An Architecture for Integrated Caching and Prefetching Mechanisms for Distributed Parallel File Systems

Evgueni Dodonov*
Federal University of São Carlos, Computer Science Department
Brazil, São Carlos (SP), 13565-905
eugeni@dc.ufscar.br

and

Hélio C Guardia
Federal University of São Carlos, Computer Science Department
Brazil, São Carlos (SP), 13565-905
helio@dc.ufscar.br

Abstract

This paper presents the architecture of an integrated caching and prefetching mechanism for distributed parallel file systems. By combining techniques for efficient data placement in the cache and the appropriate anticipation of the data to be requested by the application the mechanism proposed aims to provide reduced I/O latency and higher throughput.

A parallel file system, *NPFS*, is introduced in this paper and an adaptative prefetching algorithm *CPS* used in the architecture proposed is also presented.

Keywords: Parallel I/O, Cache and Prefetching mechanisms, High Performance Networks, Parallel and Distributed Systems, Computer Networks and Architecture.

*M. Sc. student sponsored by CNPQ Brazil.

1 Introduction

The need to store large amounts of data along with the difficulties involved in transferring such data between hard disks and memory at high speed required by applications lead to the creation of *parallel files*. Parallel files handle the splitting of a logical sequential file into a series of segments and their distribution among different servers. With simultaneous operation of several disks higher transfer rates, increased throughput and decreasing latency are obtained. The creation of an efficient parallel file system, however, involves the optimization of system policies and efficient data distribution in order to get all the advantages from parallel input-output *(i/o)* operations for the applications.

Several parallel file systems can be found in the literature, either *hardware controlled (RAID [21] [20])* or *software controlled (PFS [1], software RAID [7], NPFS [12])*. Hovewer, such filesystems are mostly local filesystems, i.e., all files are distributed between local disks on a server. A network parallel file system combines the i/o operations of several servers to store data. Such a file system brings the advantages of a parallel file system into existent networks, without additional costs.

The low latency of requests and a high transmission rate are the most favorable characteristics of such kind of filesystems. Hovewer, most of the existing networks are based on a shared transmition medium, which limits the benefits that can be offered by a parallel filesystem. Efficient use of the transmition medium is necessary and, in order to accomplish that, it is possible to use some technics, like caching and prefetching mechanisms.

This paper is structured as shown next. First, we show some background on parallel files (section 2) and caching and prefetching mechanisms (section 3). In section 4 we introduce the *NPFS* file system. Our architecture for an integrated caching and prefetching mechanism is presented on section 5. And, finally, the section 6 concludes the paper and shows our plans for the future work.

2 Parallel files

At present time, it is usual for applications to manipulate data in the magnitude of hundreds terabytes. Hovewer, most of the existing storage devices are unable to offer access time low enough to process the data in an effective manner. This becomes a limitation factor when scientific or multimedia data are processed in real time. Taking into consideration that such data cannot be stored in memory, it becomes necessary to create mechanisms to manipulate those data efficiently.

A possible solution to this problem is to use parallel files, on which the data is divided into *segments* composed by the *stripes*, as shown on figure 1. Such a structure allows the data to be read in parallel, thus decreasing the access time and maximizing the throughput. In the same way, during the write operations the data is transmitted to servers that store it into local segments.

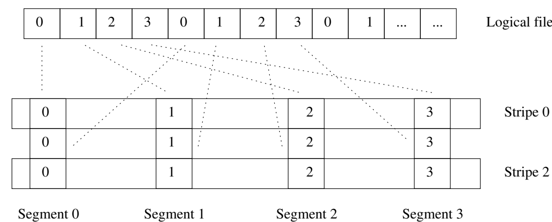

Figure 1: Parallel file structure

Thus, parallel files consist of *segments* and *stripes*, composed by *striping units*. Using this structure, instead of reading a huge sequential file, it is possible to read a parallel file, increasing the performance of data *input/output* (I/O) operations.

The process of manipulating the parallel files is called *Parallel I/O*. Considering the data organization, parallel files can be stored under hardware control, or using a *parallel file system*.

I/O requests arrive at a file server very frequently, possibly several ones at the same time. When data is shared among processes, the same portion of a file may be requested at the same time by different processes, or just after it was retrieved from the storage device. When it happens without a *caching* mechanism, the filesystem must execute several disk requests in order to read the same data several times. Hovewer, this wouldn't be necessary if this data were kept in main memory. The use of a network with a shared medium access (Ethernet networks) also suggests that the number of transmissions should be reduced for improved performance.

During the data writing process the similar situation occurs. The data can be written, read and written again. In such a case, if no caching mechanism is being used, the data must be written on disk in order to be read just after. Using a cache policy it is possible to reduce the number of disk or network accesses: the data may remain in the memory for future accesses, with no need to access the disks several times in a short period of time. As the memory

is much faster then the disk subsystem, a caching mechanism allows one to increase the performance of input/output operations.

Another method that is capable of increasing the file system performance is a *prefetching* mechanism. Using the prefetching, when an application's future data requests are known *a priori*, it is possible to anticipate the data read requests.

An integrated caching and prefetching mechanism usually allows a faster data management, as shown in [1].

3 Caching and Prefetching

The performance of storage devices, like hard disks or network devices, has a much lower transmission rate when compared to the speed of memory accesses. In most cases, the performance of an input/output operations for a networked system is limited by the disk and network performances.

In that context, we can define caching as a mechanism that allows to store the most relevant data in memory, reducing the number of disk or network accesses, increasing the performance of the I/O operations.

Prefetching is a mechanism that allows to perform anticipated data read operations, loading the obtained data into main memory or into the cache subsystem before the data read request is actually issued. The prefetching operations are controlled by a *prefetching algorithm*.

It is necessary to choose an adequate prefetching policy, as an inadequate algorithm can decrease the performance of an application, According to [4], there are some basic rules that the prefetching algorithms must follow in order to have an efficient behavior.

3.1 Cache organization

In a parallel file system, the servers are able to communicate with several clients that can be user processes requesting data or even different machines communicating via the network. In order to access the files with a caching system, it is relevant to determine how to share the cache space between all these clients. It is also important to choose which *policy* must be used to organize the data inside the file system.

We suggest four different methods to share the cache space:

Creating a *separate* cache space for each one of the clients is one possible way to distribute the cache space. By doing so, all the *caches* will be independent, improving the handling and the organization of the cache.

It is also possible to divide the cache space into a fixed number of fixed size blocks. Thus, each client will have an independent block of cache space with fixed size. This method is quite similar to the last one, however, it is not as flexible as it. The main aspects of this method are the speed and the ease of implementation. Hovewer, there is a possibility to leave some blocks unused or to have too few blocks available for each client.

Using a statically divided cache, the cache space is calculated according to the number of clients, in order to offer the same cache size to every one of them. The main difference between this method and the last one is the lack of unused blocks. Hovewer, there is still a possibility to have too few blocks for every client.

Finally, it is possible to allocate the cache space dynamically, on client's demand. The main problem with this approach is the difficulty to manage the entire cache and cache manipulation speed that will be a bit slower than in other methods. Hovewer, using this method, a client will always have all cache space it needs.

It is also possible to use several cache levels. By doing so, it is possible to have an independent cache for each one of the clients and a bigger cache that can be shared between all the clients.

In caching systems, it is usual to maintain two data structures, one for data, containing the list of all blocks that are being in use, and another one to contain the list of free blocks that can be used to load new data.

3.2 Cache management

As the cache space has a limited size, it is necessary to remove the "unused" data when new data has to be loaded. In order to do so, it is necessary to find out what data is "unused", and can be moved out of cache. It is necessary to create an efficient policy to maintain the cache updated, containing the most relevant data.

There are several cache management policies, well described in the literature [3]: LRU, LFU, FIFO, Clock, Random, Segmented FIFO, 2Q and LRU-K. The policy of cache updates depends on data contained in cache: an algorithm can be more adequate in some cases and may perform poorly in other situations.

The data contained in cache eventually must be written back to disk. There are several methods used to write data on disk [16]: *write-through*, *write-back*, *write-full*, *write-free*.

Having several processes accessing the same data simultaneously is a very common situation in parallel file systems. Hovewer, if one of these processes modifies the data it has read and writes it back, the *cache coherence* problem occurs – the data read by a second process will be different from the real data contained on disk and, when the second process writes the data back the modifications of the first process will be overwritten. This problem is called *cache coherence problem* [13].

In order to solve this problem, it is possible to use different mechanisms, like *incoherence detection* during the compilation and *coherence enforcing*. Incoherence detection mechanisms aim to detect possibly incoherent blocks during the execution of the system and it is possible to invalidate such blocks (i.e., the system must re-read such blocks) using a coherence enforcing mechanism. Hovewer, it is necessary to establish a tradeoff between the error precision and the size of cache blocks, as it is much easier to occur an incoherence in a large block that in a small one.

The basic mechanisms used to keep the cache coherent are:

- **Snoopy coherence** — this mechanism allows constant information exchange between all the clients and servers, constantly updating their *caches*. This mechanism requires a fast communication medium due to the huge amount of data to be transfered.

- **Directory–based mechanisms** — a process is created in order to maintain the information about all the data in all *caches*. Thus, it becomes necessary to check if the data in a cache is valid communicating with this process.

The main difference between these two mechanisms is that one of them is distributed and the other relies on a centralized directory service.

3.3 Prefetching access patterns

There are several different manners to read a file — the file can be read sequentially, from the beginning until the end; it can be read by different processes; it can be splitted up into several segments, and just one of those segments can be read; the file can even be read without any visible reading pattern.

Knowing *a priori* the application needs, the filesystem may preload the data into memory, anticipating the application's requests. By doing so, the filesystem is able to increase the performance of read operation, decreasing the waiting time between the request to read the data, the reading of the data itself and the data processing. The mechanism that allows such anticipated data preload is called prefetching, and the operation of reading data only on application's demand is called *post-fetching*.

Each file can be read in a different way: the file can be read from the beginning to the end, it can be split in several segments, and so on [17]. The manner and the sequence of data reading characterize different *access patterns*. Knowing those patterns, a *prefetching algorithm* can be used to perform the anticipated data preload.

The main sequential access patterns are *one-block look-ahead, n-block look-ahead* and *infinite-block look-ahead* [17]. Beside those access patterns, an application may split the file in blocks and read some of them (*non-sequential reading*).

3.4 Design of a prefetching mechanism

According to [4], there are four basic rules that must be followed by any prefetching algorithm:

- **Optimal Prefetching** — Every *prefetch* should bring into the cache the next block in the reference stream that is not in the cache.

- **Optimal Replacement** — Every *prefetch* should discard the block whose next reference is furthest in the future.

- **Do No Harm** — Never discard block A to *prefetch* block B when A will be referenced before B.

- **First Opportunity** — Never perform a *prefetch–and–replace* operation when the same operation (fetching the same block and replacing the same block) could have been performed previously.

When used together, these four rules allow to define what must be loaded into the cache using the prefetching algorithm or unloaded from the cache in order to store new data. These rules must be followed in order to create an effective prefetching algorithm.

In order to define the *strategy* of a prefetching algorithm, we must select it's way of operation. Basically, there are just two prefetching policies: *aggressive prefetching* and *passive prefetching* [4]. All other policies are based on these two.

Both policies have pros and cons:

- **Aggressive prefetching** — this policy always executes the prefetching, no matter what data is being read. This can bring significant performance gains (in the best case, all the data will be preloaded into memory) but it also can decrease the performance (in the worst case, no data will be preloaded correctly or, even worse, data that will be referenced in the future will be unloaded from cache to load data that will never be used. In that case more disk accesses will be necessary in order to load the correct data).

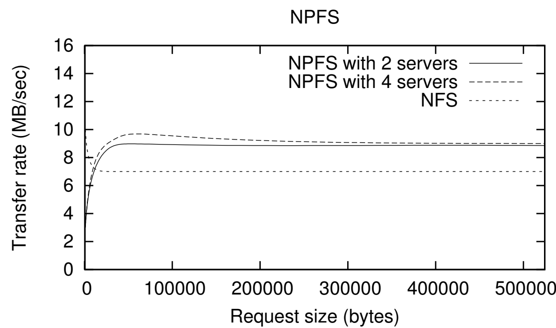

Figure 2: Comparing *NPFS* and *NFS* on read operations

- **Passive prefetching** — this policy executes the prefetching only when absolutely necessary. There is no big performance boost, but the application will never need more disk accesses to read some data than an application without prefetching.

An *optimal* prefetching policy is situated between these two policies. It is necessary to make a choice between performance boost in some cases and eventual additional disk accesses and a little performance gain, but without additional disk accesses.

According to benchmarks ([8] [10] [6]), *aggressive prefetching* is, usually, more efficient than *passive prefetching*. Hovewer, this result is highly application dependent and a case study is essential in order to find out which policy is the best for each application.

3.5 Prefetching algorithms

Each access pattern suggests a different method to read data. It is not possible to use the same method to execute prefetching for every type of access. A relevant question is how to find out which method must be used in every case.

The creation of an *adaptative* prefetching algorithm resolves this problem: analyzing different access patterns of the application, an algorithm may be able to choose the method that should be used to read the data. This algorithm must be capable to determine a change in an access pattern, modifying the method used to read the data.

Among the prefetching algorithms it is possible to mention the *supervisor* algorithm [14], *NOM* and *GREED* algorithms [2] and *full–file–on–open* algorithm [9].

3.6 Integrating caching and prefetching

It is possible to use the caching and the prefetching algorithms in a more efficient way, integrating them.

Without an *integrated* caching and prefetching solution, the cache is used just to contain the data that has already been read; and the prefetching algorithm loads new data directly into the local memory of the process, thus decreasing the advantages of the caching. Also, without an integrated caching and prefetching system, the prefetching algorithm drops the performance of the input/output operations, requiring an independent place in the memory to be used to store data retrieved by the prefetching algorithm.

In an *integrated caching–prefetching* system, the data read using anticipated reading is stored in cache, and the application retrieves data from this cache. So, it is possible to increase the performance of the I/O system, using a caching to keep the most relevant data and prefetching to keep the newly obtained data in the same cache memory.

4 Network Parallel File System (*NPFS*)

NPFS[11] is a software controlled parallel file system. The target architecture for *NPFS* consists of a series of network workstations, each containing a local shareable hard disk. Using the parallel file techniques, the filesystem presents a better performance when compared to NFS filesystem, as shown on figures 2 and 3.

Data distribution is decided when a file is created and it is possible to specify the number of segments and the size of the striping units.

This filesystem uses the client–server model to create servers on the network. At all, there are three kinds of processes — **master**, **server** and *client*, as shown on figure 4.

The *master* is responsible for the system initialization functions; it starts up and shuts down the servers. It is also responsible for file opening and closing.

The **servers** allow access to data segments that are stored in their local disks. An instance of the server process runs on each computer whose disk is being used by the *NPFS*.

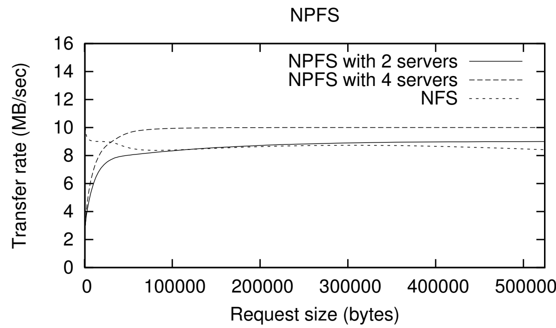

Figure 3: Comparing *NPFS* and *NFS* on write operations

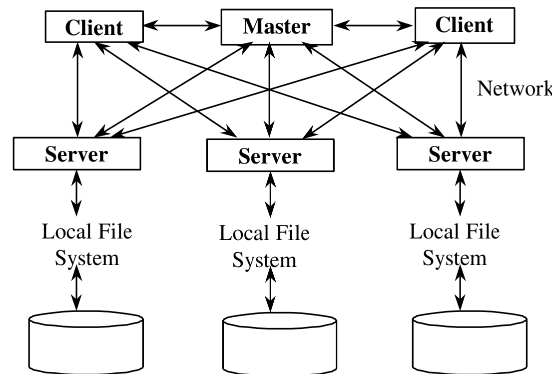

Figure 4: NPFS architecture

The **clients** are the user processes. Their interaction with the *NPFS* is performed with help of series of functions. These functions are part of the *NPFS* Application Programming Interface (*API*).

The system activation consists in creation of a *master* process. After that, the master activates all the *servers* and waits for clients. The clients connect to the *master* in order to open the files and, as soon as the files are opened, the clients interact directly with the **servers**.

The Application Programming Interface of *NPFS* is based on the *C* language basic syntax used in filesystem interactions: *open(), read(), write()* and so on. As the names and arguments of *NPFS* function are quite similar to standard primitives for the file operations, it becomes easy to create a program which can use *NPFS* filesystem, or convert any existent application. The NPFS functions are *p_open(), p_read()*, etc.

During the reading and writing operations, the client determines automatically which servers contain the needed file segments and sends them the appropriate requests.

Actually, both reading and writing operations are *synchronous*, there is no caching or prefetching algorithm.

5 An Integrated Cache and Prefetching Architecture for Distributed Parallel File Systems

Considering the potential benefits of the caching and prefetching mechanisms, this papers presents an architecture for the implementation of caching and prefetching algorithms to be used in parallel file systems for clusters of workstations. This work focuses mainly on the NPFS file system, due to its source code availability. However, the modular design of the caching and prefetching mechanism allows a fast integration with other file systems.

As the transmition medium is usually a shared medium (as in *Ethernet* networks), this paper attempts to create mechanisms that can reduce the performance drop and interferences between file systems and distributed applications.

For performance reasons, most of the *NPFS* operations are performed by the clients, leaving the servers to simply read and write the requested blocks on their disks. In such a case, a caching system implemented on the clients may increase the performance of I/O operations and a prefetching mechanism may decrease the latency, the time elapsed to read data, by preloading the data into the memory.

On the servers, a caching mechanism can also be useful. In this case, the cache management system comes into action, in order to define cache policy — wherever it is a common buffer or several per-client buffers.

Making an analogy with the processor cache architecture, we can call the client side cache as a *level 1 cache* and server side cache as *level 2 cache*.

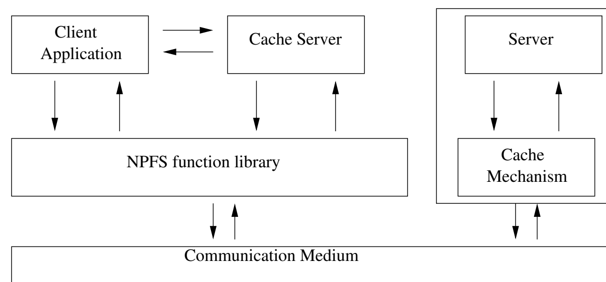

Figure 5: Cache server architecture

In the NPFS filesystem, the master just coordinates all the file opening and closing operations. There is no need for any caching or prefetching mechanism in its operation, since it does not handle the data requested by the clients.

Using a prefetching algorithm designed specifically for *NPFS* filesystem, it is possible to optimize the communication medium utilization, anticipating the transmition of the needed data and storing it in a cache, avoiding unnecessary retransmitions.

Different mechanisms of caching and prefetching must be used on the servers and clients, in order to use effectively the system resources.

5.1 System Architecture

In order to make the caching and prefetching system more flexible, a modular design is used. A *cache server* is implemented as a process running on the same machine of every client process. On the servers, the cache system is integrated into the *server* process, in order to increase its performance. This architecture is shown on figure 5.

In this architecture, the cache server receives requests to read data from clients, reads the data from servers, possibly using a prefetching mechanism, stores the read data in cache space and sends the data to client. On further requests the cache server simply retrieves the data from cache and sends it to the client.

Using this architecture, the implementation of our caching and prefetching mechanism is highly independent on the file system's internal structure, allowing quick adaptation and modification without any need to rewrite the entire file system or even recompile the client applications.

Hovewer, as several new requests are needed in order to transfer data from cache server to the client process, the transfer rate decreases.

In order to increase the transfer rate between the *cache server* and the client processes, several communication mechanisms may be used, such as *named pipes*, *shared memory* structures, *unix-domain sockets* [22] and *zero-copy* mechanisms, like the *sendfile()* [25] [24] primitive, which are available on most modern operational systems, like Linux 2.4 [25], FreeBSD [19] and Sun Microsystems Solaris [5] [15].

A generic communication mechanism, that uses only the basic functions, such as *open_connection*, *close_connection*, *send_data* and *receive_data*, and chooses the most adequate transfer method according to the operational system, is implemented in order to obtain the best performance on each operational system without the need to recompile.

5.2 Using *threads* with the cache server

In order to implement different cache write policies, the cache server uses several *threads* [23] in order to enable the *delayed write* operations, such as *writeback*, and to improve the communication with different clients.

The following threads are used during the cache server operations:

- **Main thread** — receives the connections from the clients and dispatches new threads in order to attend them.
- **Client threads** — receive read and write requests from the clients. On *read* requests it retrieves the data from the server, stores it in cache and sends it to the client process. On *write* requests it puts the data to be written into cache and marks such blocks as *dirty*.
- **Write thread —** sends the *dirty* blocks to the appropriate servers.

5.3 Cache and prefetching operations

In this paper, our prefetching mechanism is used together with the caching system, on the *cache server*, discussed above.

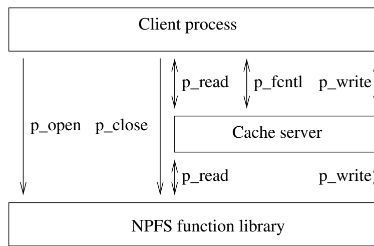

Figure 6: Cache server operation

```
main thread:

    receive client request;
    create a new client thread;
    pass request to the client thread;

client thread:

    contact the master process to determine the servers;
    contact the servers;
    wait for read or write requests;
```

Table 1: Cache server operation algorithm for *open* requests

The caching and prefetching mechanism may be activated when opening a file and operate transparently, similarly to the *assynchronous I/O* mechanism. It can also be controlled via the *p_fcntl* (file control) function, which allows to define the cache policy, the prefetching algorithm, and other parameters.

The *p_fcntl* function allows to define the caching and prefetching mechanism operation using primitives and constants that are transmitted to the cache server. In order to simplify the operation of such primitives, we define several commands that can be transmitted to the cache server, such as: *NPFS_LCACHE_SIZE, NPFS_SCACHE_SIZE, NPFS_PREFETCH_METHOD* and others. These commands are transmitted along with their parameters, that represent, for example, the local cache size, the server cache size and the prefetching algorithm.

The algorithm for caching operations on *read* requests is shown on figures 3. The caching operations with *write* requests is different: the cache server uses a separate thread in order to write the requested blocks back to the servers, according to the cache write policy (i.e., *writethrough*, *writeback*, etc) and returns to the client just after it has updated its cache with the data to be written. The write operation itself is handled by another thread, as shown on figure 4. The open and close operations are responsible for the creation of new threads, as shown on tables 1 and 2.

In order to provide a transparent integration with existing filesystem functions, the cache server *overrides* the data access functions from the original file system, as shown on figure 6.

For example, if we use the *NPFS* filesystem, we just have to override the *p_read* and *p_write*. When the client application uses, for example, *p_read* function to read a text file, it must:

- Contact the cache server using an adequate communication mechanism.
- Send the request to read data to the cache server.
- Read data from the cache server.

When the cache server receives the data read request, it uses its algorithm (shown on table 3) in order to get the data and sends it to the client. This process operates transparently to the client. The client can use the *f_fcntl* function to change the behavior of caching and prefetching operations, contacting the cache server directly.

When the write data request is received, the server uses its algorithm (shown on table 4) in order to send the data to be written to the servers.

```
client thread:

    write blocks marked as dirty;
    remove dirty blocks from cache;
    close all open connections;
    destroy thread;
```

Table 2: Cache server operation algorithm for *close* requests

```
client thread:

    determine the needed disk blocks;
    for each block in list:
        check for block in local cache;
        if data found in cache:
            put the data into buffer;
            update cache hit counter;
            remove block from list;
    for each remaining block in the list:
        request block from appropriate server;
        free space in cache for the new blocks;
        put blocks read into local cache;
        initialize the cache hit counter;
        put blocks read into the buffer;
    update CPS access pattern;
    execute the prefetching algorithm;
    adjust the Probability Success Counter;
```

Table 3: Cache server operation algorithm for *read* requests

```
main thread:

    determine the needed disk blocks;
    free space in cache for these blocks;
    for each block in list:
        put block into cache;
        mark block as dirty;
    return to the client program;

cache write thread:

    wait for cache write policy;
    for each block in list:
        send block to corresponding server;
        remove block from cache;
```

Table 4: Cache operation algorithm for *write* requests

5.4 CPS Prefetching Algorithm

In order to perform the prefetching operations we introduce a prefetching algorithm, called CPS (stands for Probability Success Counter). The CPS algorithm is an adaptative algorithm, that counts the number of successful prefetching operations and number of failures. If the number of failures is too high, CPS selects a different prefetching access pattern or, if there are no more suitable patterns, it stops all prefetching operations.

The algorithm used by CPS is the following:

1. When a file is opened, a counter is initialized by the cache server with a default starting value. The counter is called the *success probability counter*.

2. After the first read request, the default access pattern is evaluated. According to the pattern, the CPS algorithm sends a *cache load* request to the appropriate server, which commands the server to load the data block that may be read next into the server's cache. In order to avoid unnecessary data transmissions, no data is being transferred effectivelly.

3. If the client application's next read request has been correctly "guessed" by the current access pattern, the success probability counter is incremented by **one**. Otherwise, it is decremented by **two**. This allows a more accurate access pattern guessing, avoiding errors.

4. If the success probability counter drops below a certain value, the current access pattern is substituted by the next known access pattern and the success probability counter is reset to its original value. If there are no more known access patterns, the algorithm stops its operation, efectivelly canceling the prefetching mechanism.

5. When the success probability counter passes a certain value, the algorithm stops sending *cache load* commands and starts its prefetching operations, sending *read* commands to the server.

6. If the success probability counter passes another predefined value, the algorithm stops incrementing it further.

7. On a file close request, the algorithm resets the counter for that file.

The main objective of the CPS algorithm is to provide a transparent prefetching mechanism activation using the most adequate prefetching access pattern for data access. Using different values for the CPS operations (when the access pattern should be changed, how many *cache load* requests must be sent and others) it is possible to customize the algorithm according to the application needs.

5.5 Evaluating the CPS algorithm

In order to prove the efficiency of the CPS algorithm, we should consider on which case it allows performance gains and when it may drop the speed of I/O operations.

Given that the CPS algorithm sends a small amount of data to the servers, we can consider the following:

- On *small* files operations the use of *cache load* operations may drop the I/O performance: there is no real need to prefetch the caching operations on the server with small chunks of data.
- On *big* files operations the *cache load* operations may allow the server cache to load the data antecipatelly and send it to the client without the need to access the storage device (i.e., the hard disk).
- On *small* intervals between subsequent data read requests (for example, when the application only reads the file without processing the read data), the *cache load* operations won't offer any significant gain of performance. In this case, the *cache load* request will be followed by the *read* request, and no real gain with the prefetching will be obtained.
- On *big* intervals between subsequent data read requests (for example, when the application reads a set of numbers and processes them), the *cache load* operations allow the server to load the data into the cache before the read request is received, improving the I/O performance.

Considering the size of the files and the time elapsed to process each piece of data, it is possible to select the most adequate prefetching algorithm. When the kind of file operations is unknown to the application, the CPS algorithm allows a smooth prefetching activation and selection of most suitable access pattern for the application.

This algorithm can be used in order to determine application's access patterns. After that, it is possible to implement a prefetching algorithm that is most suitable for that application.

5.6 Case study

The architecture for an integrated caching and prefetching mechanism introduced above can easily improve the overall performance of a parallel file system and its applications. This architecture can be beneficial for different kinds of applications. As an example, we can use it together with a distributed *video on demand* server [18]. In this case, our prefetching mechanism may be specially useful in order to improve the video acquisition rate, anticipating the video frames to be retrieved and storing them in a local cache.

It is possible to determine the type of a video file when opening it. After that, we can determine the size of each frame or the size of each request block. Optimizing the size of the cache to hold, for example, **15** seconds of video, we can use the prefetching algorithm in order to keep the cache updated with newest data, eliminating the loss of frames caused by a sudden drop of network performance.

6 Conclusion and plans for future work

In this paper, we have presented the architecture for an integrated caching and prefetching mechanism for a parallel file system.

As has been discussed, a mechanism for combined caching and prefetching operation can effectively improve the performance of I/O, increasing the throughput and decreasing the latency of operations. An integrated caching and prefetching mechanism is an important mean for improving the performance of a filesystem.

Due to the concurrent network accesses in a parallel file system over a cluster of workstations such mechanism may provide even greater performance gains, as unnecessary network accesses may be avoided.

The measurement of the overall performance gains of the methodology presented is in progress. Providing support for specific classes of applications, such as a parallel *video on demand* server is also under development [18]. The mapping of the same functionalities on other parallel file systems is being concerned.

References

[1] Meenkashi Arunachalam, Alok Choudhary, and Brad Rullman. Implementation and evaluation of prefetching in the Intel Paragon Parallel File System. In *Proceedings of the Tenth International Parallel Processing Symposium*, pages 554–559, April 1996.

[2] Rakesh Barve, Mahesh Kallahalla, Peter J. Varman, and Jeffrey Scott Vitter. Competitive parallel disk prefetching and buffer management. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–56, San Jose, CA, November 1997. ACM Press.

[3] Nathan C. Burnett, John Bent, Andrea C. Arpaci Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *Proceedings of the 2002 USENIX Technical Conference*, June 2002.

[4] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching stategies. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 188–197. ACM Press, May 1995.

[5] H. K. Jerry Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[6] T. Cortes and J. Labarta. Linear aggressive prefetching: A way to increase the performance of cooperative caches. In *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, pages 45–54, San Juan, Puerto Rico, April 1999.

[7] Toni Cortes. Software RAID and parallel filesystems. In Rajkumar Buyya, editor, *High Peformance Cluster Computing*, pages 463–496. Prentice Hall PTR, 1999.

[8] Toni Cortes, Sergi Girona, and Jesús Labarta. PACA: A cooperative file system cache for parallel machines. In *Proceedings of the 2nd International Euro-Par Conference*, pages I:477–486, August 1996.

[9] Toni Cortes, Sergi Girona, and Jesús Labarta. PACA: A cooperative file system cache for parallel machines. Technical Report 96-07, UPC-CEPBA, 1996.

[10] Toni Cortes, Sergi Girona, and Jesús Labarta. Avoiding the cache-coherence problem in a parallel/distributed file system. Technical Report UPC-CEPBA-1996-13, UPC-CEPBA, May 1997.

[11] Hélio Crestana Guardia. *Considerações sobre as estratégias de um Sistema de Arquivos Paralelos integrado ao processamento distribuído*. PhD thesis, EPUSP, 1999.

[12] Hélio Crestana Guardia and Liria M. Sato. Soluções paralelas para entrada e saída de dados com alto desempenho. Technical report, Dept. of Computing, Federal University of São Carlos, June 1998.

[13] Emil Gustafsson and Bruno Nilbert. Cache coherence in parallel multiprocessors. Technical report, Dept. of Computing Science, Uppsala University, February 1997.

[14] Mahesh Kallahalla and Peter J. Varman. Optimal prefetching and caching for parallel I/O systems. In *Proceedings of the Thirteenth Symposium on Parallel Algorithms and Architectures*. ACM Press, July 2001. To appear.

[15] Yousef A. Khalidi and Moti N. Thadani. An efficient zero-copy i/o framework for unix. Technical report, Sun Microsystems Research, May 1995.

[16] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[17] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[18] Carla R. Figueredo Lara and Hélio Crestana Guardia. Projeto de um servidor de vídeo sob demanda paralelo e distribuído. Technical report, Universidade Federal de São Carlos, July 2002.

[19] Ken Merry. Zero-copy sockets and nfs patches for freebsd. http://people.freebsd.org/ ken/zero_copy/, Dec 2001.

[20] David Patterson, Peter Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). In *Proceedings of IEEE Compcon*, pages 112–117, Spring 1989.

[21] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[22] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall PTR, 1998.

[23] Andrew S. Tannenbaum. *Modern Operating Systems, 2nd edition*. Prentice Hall, 2001.

[24] Alexander Tormasov and Alexey Kuznetcov. Tcp/ip options for high-performance data transmission. http://builder.com.com/, 2002.

[25] Alexander Tormasov and Alexey Kuznetcov. Use sendfile() to optimize data transfer. http://builder.com.com/, 2002.