Performance Evaluation of the LH*lh Scalable, Distributed Data Structure for a Cluster of Workstations

Vinay Gupta Mohit Modi
Indian Institute of Technology
New Delhi, India
{mau96419,mau96409}@ccsun50.iitd.ac.in

Andy D. Pimentel
Dept. of Computer Science
University of Amsterdam
The Netherlands
andy@science.uva.nl

Keywords
Scalable distributed data structures, performance evaluation, cluster of workstations

ABSTRACT
Scalable, Distributed Data Structures (SDDSs) can provide fast access to large volumes of data. They allow the data structure to grow or shrink without suffering from a penalty with respect to the space utilization or the average access time. In this paper, we present a performance study of one particular SDDS, called LH*LH, which has been implemented for a cluster of workstations. Our experimental results demonstrate that our LH*LH implementation is truly scalable and yields access-times that are of an order of magnitude smaller than a typical disk access. Furthermore, we also show that parallel access to the LH*LH data structure can speed up client applications quite significantly.

1. INTRODUCTION
Modern data intensive applications require fast access to large volumes of data. Sometimes the amount of data is so large that it cannot be efficiently stored or processed by a uni-processor system. Therefore, a *distributed data structure* can be used that distributes the data over a number of processors within a parallel or distributed system. This is an attractive possibility because the achievements in the field of communication networks for parallel and distributed systems have made remote memory accesses faster than accesses to the local disk. So, even when disregarding the additional processing power of parallel platforms, it has become more efficient to use the main memory of other processors than to use the local disk.

It is highly desirable for a distributed data structure to be scalable. The data structure should not have a theoretical upper limit after which performance degrades (i.e. the access time is independent of the number of stored data elements) and it should grow and shrink incrementally rather than reorganizing itself totally on a regular basis. For distributed memory parallel computers, a number of Scalable Distributed Data Structures (*SDDS*s) have been proposed [1, 8, 5]. In these distributed storage methods, the processing nodes are divided into *clients* and *servers*. A client manipulates the SDDS by inserting, removing or searching for data elements. A server stores a part of the data, called a *bucket*, and receives data-access requests from clients. To realize a high degree of scalability, an SDDS cannot be indexed using a central directory since this would form a bottleneck. As a consequence, the clients have an image of how data is distributed which is as accurate as possible. This image should be improved each time a client makes an "addressing error", i.e. contacts a server which does not contain the required data. If a client makes an addressing error, then the SDDS is responsible for forwarding the client's request to the correct server and for updating the client's image.

For an efficient SDDS, it is essential that the network communication needed for data operations (retrieval, insertion, etc.) is minimized while the amount of data residing at the servers (i.e. the *load factor*) is well balanced. An example of an SDDS addressing these issues is *LH** [5]. This SDDS is a distributed variant of Linear Hashing (LH) [4], which will be elaborated upon in the next section. For LH*, insertions usually require one message (from client to server) and three messages in the worst case. Data retrieval requires one extra message as the requested data has to be returned.

In this paper, we present a performance evaluation of a variant of the LH* SDDS, called LH*LH [3], for a cluster of workstations (COW). Previous studies already showed good performance of LH*LH when targeting multicomputers. In [2], an actual implementation for a Transputer-based machine has been evaluated, while in [7] simulation was applied to investigate LH*LH's performance for a PowerPC-based multicomputer. In this study, we try to gain insight into how today's COWs with their commodity networks affect LH*LH's performance. Our final goal is to embed LH*LH into a real-world distributed Web-cache application.

The next section briefly explains the concept of Linear Hashing. In Section 3, we discuss the LH*LH SDDS and describe its implementation for a cluster of workstations. Section 4 describes our experimental setup and presents the performance results. Finally, Section 5 concludes the paper.

2. LINEAR HASHING
Linear Hashing (LH) [4] is a method to dynamically manage a table of data. More specifically, it allows the table to grow or shrink in time without suffering from a penalty with respect to the space utilization or the access time. The LH table is formed by $N \times 2^i + n$ buckets, where N is the number of starting buckets ($N \geq 1$ and

SAC 2001, Las Vegas, NV

$n < 2^i$). The meaning of i and n is explained later on. The buckets in the table are addressed by means of a pair of hashing functions h_i and h_{i+1}, with $i = 0, 1, 2...$ Each bucket can contain a predefined number of data elements. The function h_i hashes data keys to one of the first $N \times 2^i$ buckets in the table. The function h_{i+1} is used to hash data keys to the remaining buckets. In this paper, we assume hash functions of the form

$$h_i(\text{key}) \rightarrow \text{key } mod\ (N \times 2^i) \quad (1)$$

The LH data structure grows by splitting a bucket into two buckets whenever a certain load threshold is exceeded. The bucket that needs to be split is determined by a special pointer, referred to as n. The actual splitting involves three steps: creating a new bucket, dividing the data elements over the old and the newly created bucket and updating the pointer n. Dividing the data elements over the two buckets is done by applying the function h_{i+1} to each element in the splitting bucket. The n pointer is updated by applying $n = (n + 1)\ mod\ N \times 2^i$. Indexing the LH data structure is performed using both h_i and h_{i+1}:

$$\text{index}_{bucket} = h_i(\text{key}) \quad (2)$$
$$\text{if } (\text{index}_{bucket} < n) \text{ then } \text{index}_{bucket} = h_{i+1}(\text{key})$$

As the buckets below the n pointer have been split, these buckets should be indexed using h_{i+1} rather than with h_i. When the n pointer wraps around (because of the modulo), i should be incremented. The process of shrinking is similar to the growing of the LH data structure. In this study, we limit our discussion to the splitting within SDDSs.

3. THE LH*LH SDDS

The LH* SDDS is a generalization of Linear Hashing (LH) to a distributed memory parallel system [5]. In this paper, we focus on a particular variant of LH*, called LH*LH [2, 3]. The LH*LH data is stored over a number of server processes and can be accessed through dedicated client processes. These clients form the interface between an application and LH*LH. For this study, we assume that each server stores *one* LH*LH bucket of data. Globally, the servers apply the LH* scheme to manage their data, while the servers use traditional sequential LH for their local bucket management. Thus, a server's LH*LH bucket is implemented as a collection of LH buckets.

As was previously explained, addressing a bucket in LH is done using a key and the two variables i and n (see Equation 2). In LH*LH, the clients address the servers in the same manner. To do so, each client has its own *image* of the values i and n: i' and n' respectively. Because the images i' and n' may not be up to date, clients can address the wrong server. Therefore, the servers need to verify whether or not incoming client requests are correctly addressed. If a request is incorrectly addressed, then the server forwards the request to the server that is believed to be correct. For this purpose, the server uses a forwarding algorithm [5] for which it has been proven that a request is forwarded at most twice before the correct server is found. Each time a request is forwarded, the forwarding server sends an Image Adjustment Message (IAM) to the requesting client. This IAM contains the server's local notion of i and n and is used to adjust the client's i' and n' in order to get them closer to the global i and n values. Consequently, future requests will have a higher probability of being addressed correctly.

The splitting of an LH*LH bucket (a *global split*) is similar to the splitting of LH buckets. The pointer n is implemented by a special token which is passed from server to server in the same manner as n is updated in LH: it is forwarded in a ring formed by the servers 0 to $N \times 2^i$, where N is the number of starting servers. When a server holds the n token and its load factor exceeds a particular threshold, the server splits its LH*LH bucket and forwards the n token. We apply the threshold proposed in [5], which has shown to be effective. Splitting an LH*LH bucket is done by initializing a new server (by sending it a special message) and shipping half of the LH buckets to the new server (remember that an LH*LH bucket is implemented as a collection of LH buckets). During a global split, it is not required to separately visit or rehash all data elements. In LH*LH, the LH buckets with an odd index are shipped to the new server while the even buckets are compacted and remain at the splitting server. The bucket shipments can be done in a bulk fashion (transferring a whole LH bucket in a single message) or using messages containing one to a few data elements only. In the next section, we will elaborate on the actual design decisions we made for our prototype LH*LH implementation.

3.1 An implementation of LH*LH for a COW

We have developed a prototype implementation of LH*LH for a cluster of workstations (COW) which uses TCP/IP sockets for communication [6]. Currently, it supports only insert and lookup operations. Insertions are always explicitly acknowledged by a server, which allows for determining when insertions have actually been committed at the server side and are not floating around in the network anymore. At startup, a specified number of client and server processes are placed at the workstations within the cluster. For our prototype implementation, at most one client and server process can be placed on a single machine. Clients and servers cannot be dynamically created at run time. This means that in order to allow global splitting, server processes can assume either one of the following two states: active and dormant. The active servers are immediately part of the LH*LH SDDS, while the dormant servers wait to be activated to join the SDDS when a new server is needed in the case of a global split. If there are no dormant servers left, then the global splitting is deactivated and the total number of servers remains constant after that point.

The server processes are composed of two threads which each have their own socket connection to all the other server processes. The first thread, which we refer to as the *server thread*, handles the requests from clients as well as the forwarded requests from other servers. It maintains the local LH table and takes care of forwarding a request in the case of an addressing error. The main responsibility of the other thread, called the *splitter thread*, is the global splitting. Besides this, it also takes care of several other tasks such as forwarding the split token and activating a (dormant) server when it needs to join the SDDS. Once a splitter thread receives the split token, it waits for a signal from the server thread before initiating a global split. The server thread only signals the splitter thread when the server becomes overloaded and there is still a dormant server available.

Because both threads in a server process have a private socket communication link, *concurrent splitting* [2] is allowed. This implies that when the splitter thread ships half of the data to a newly created server, the server thread continues to handle incoming requests. If the server thread notices that a requested piece of data has been or is being shipped, then it forwards the request to the new server. The advantage of concurrent splitting is that it allows for overlapping (parts of) the communication overhead due to the splitting by meaningful computation. However, since both the server and splitter

threads share access to the same local LH*LH bucket, mutual exclusion needs to be enforced to protect the consistency of the shared data. It is imperative that the semaphore locking scheme establishing the mutual exclusion is efficient and does not greatly hamper the parallelism between the two threads. Our locking scheme operates at the granularity of LH buckets, which provides enough freedom for parallel access to the LH data structure. A detailed description of the locking methods is beyond the scope of this paper but can be found in [6].

At a global split, our prototype implementation currently performs the shipment of LH buckets at a granularity of single data elements. This means that for every shipped data element a separate message is used. We decided upon this approach as an initial implementation which is easy to realize but which is probably also not the most efficient one. Hence, we do not address the performance impact of different shipment strategies but regard this as future work.

4. EXPERIMENTAL RESULTS

To conduct the performance analysis of our LH*LH prototype implementation, we used an application which builds the SDDS by inserting a large number of data elements. The application uses a Dutch dictionary of roughly 180,000 words for the data keys. Attached to each key is a data element which has a size of 1 Kb (unless stated otherwise). If multiple clients are used to build the SDDS, then they each insert a different part of the dictionary in parallel. Due to space restrictions, this section only presents a selection of our experimental results. More results, including for example data retrieval performance, can be found in [6].

The platform used for our experiments is a cluster of workstations consisting of eight nodes with each two 300Mhz Pentium II processors and 512 Mb main memory. This makes a total of 4 Gb of memory which we regard as the upper limit for LH*LH's size in our experiments. The workstations are connected to both 100 Mb/s Fast Ethernet and 1.28 Gb/s Myrinet. Our prototype implementation is capable of selecting between either one of these two networks but it uses Fast Ethernet by default. For our experiments, the clients are mapped onto nodes 1 up to 8 while the servers are mapped onto nodes 8 down to 1. So, if the application is executed with 8 clients and the LH*LH SDDS has grown to 8 servers, then each node effectively is handling both a client and a server process.

Figure 1(a) shows the time it takes to build the LH*LH SDDS for $1, 2, ..., 8$ clients. In this experiment, the SDDS starts with one server. The data points in Figure 1(a) correspond to the points in time where global LH*LH splits take place. Clearly, the splits stop after about 80K insertions because the SDDS has grown to the maximum number of eight servers of our experimental setup. Note that this limit is no way a restriction imposed by the data structure.

LH*LH's performance behavior on a COW, as shown in Figure 1(a), closely corresponds to its behavior on multicomputer architectures [2, 7]. Figure 1(a) demonstrates that LH*LH is truly scalable since the build time scales linearly with the number of insertions. Hence, the insertion latency does not deteriorate for large data structure sizes and is dominated entirely by the message roundtrip time. In addition, Figure 1(a) shows that the build-time decreases when more clients are used. This is due to the increased parallelism as each client concurrently operates on the LH*LH SDDS. Figure 1(b) depicts how the obtained insertion throughput relates to the ideal behavior when increasing the number of clients. The results show that the performance scales quite well up to about 5 clients, after which

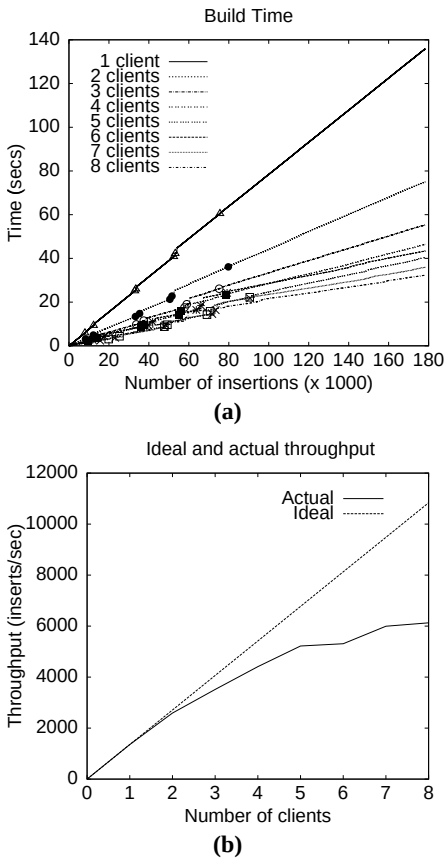

Figure 1: SDDS build time when using 1 starting server (a) and speedup for multiple clients (b).

network and server contention halts the performance improvement.

In Figure 2, the average insertion time as experienced by the application is plotted. Again, the data points refer to the moments in time where splits occur. The different curves correspond to the performance when varying the number of clients. From the top downwards, the first curve depicts the average insertion time for a single client, the next one shows the performance for two clients and so on. As expected (see Figure 1), the average insertion time improves when increasing the number of clients. The average insertion time for one client (sequential access to the data structure) stays below 0.85ms, while 8 clients yield an average insertion time of about 0.2ms. These numbers are an order of magnitude faster than the typical time to access a disk. This clearly justifies the use of a data structure like LH*LH.

At the occurrence of global splits, Figure 2 shows small increases of the average insertion time. This is due to the contention between the server and splitter threads for the local LH data structure. Another phenomenon we observed is that below 30K insertions the configuration with 8 clients performs slightly worse than when using 6 or 7 clients. This can be explained by the fact that a larger number of clients requires the creation of enough servers before the clients are able to effectively exploit parallelism, i.e. allowing them to concurrently access the SDDS with low server contention. Evidently, after about 30K insertions, the SDDS has grown to enough servers to support 8 clients.

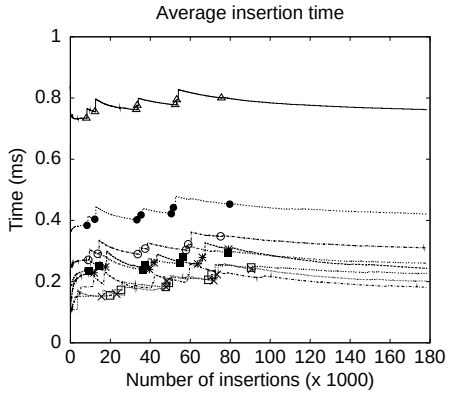

Figure 2: Average insertion time measured from the application's point of perspective (1 starting server).

In Figure 3, a histogram is plotted of the insertion times measured at a single client. The results are for 8 clients and 3 starting servers. Most accesses (more than 70%) lie in a narrow band between 0.6ms and 0.8ms which once again highlights the scalability and uniform performance of the SDDS. The small cluster near 0.25ms is caused by the way we mapped the server and client processes onto the available nodes. When a server and client get mapped to the same node, accesses by the client to the local server are much faster as opposed to remote accesses since they do not require network communication. The small number of relatively slow insertions (up to about 2.5ms) are again caused by server contention (both due to multiple clients accessing the same server and the two server threads contending for the shared LH structure).

So far, all experiments used data elements of 1 Kb, thereby building a data structure of about 175 Mb. In Figure 4, the results are shown for an experiment in which we increased the data element size to respectively 2, 4 and 8 Kb. Using data elements of 8 Kb, a data structure of more than 1.4 Gb is created. In this experiment, we used 3 starting servers. Note that the x-axis in Figure 4 has a logarithmic scale. The curves indicate that the average insertion time is linear to the data size. Of course, this is not surprising, knowing that the insertion performance is dominated by the message roundtrip time. The deterioration of the insertion performance due to larger data elements becomes slightly worse for an increasing number of clients.

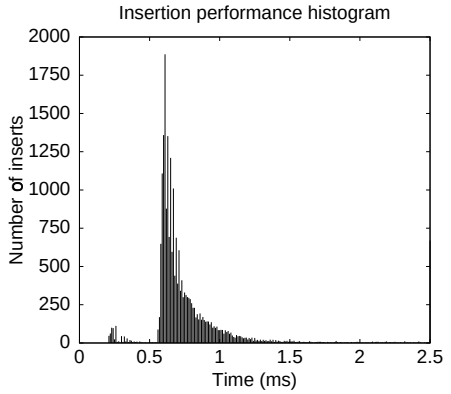

Figure 3: Histogram of insertion latencies for a single client. In this experiment, we used 8 clients and 3 starting servers.

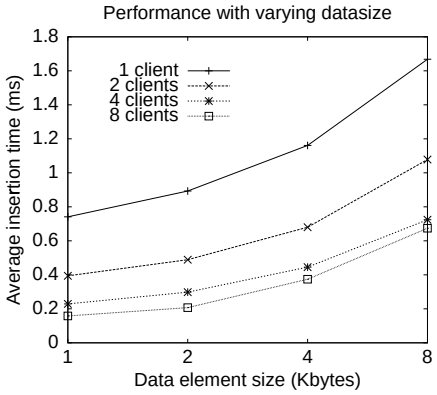

Figure 4: Insertion performance when varying the data size (with 3 starting servers).

Evidently, more clients generate higher network contention which reduces the available parallelism.

The previous experiments allowed LH*LH to grow up to 8 servers. Figure 5 depicts the SDDS build times when varying the maximum number of servers to 1, 2, 4 or 8. The results are shown for two experiments, using 4 clients (Figure 5a) and 8 clients (Figure 5b).

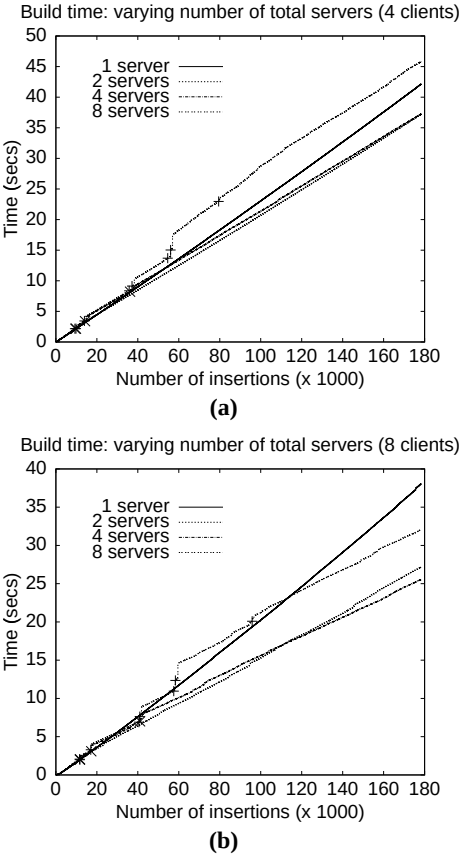

Figure 5: Insertion performance when varying the maximum number of servers (with one starting server), using 4 clients (a) or 8 clients (b).

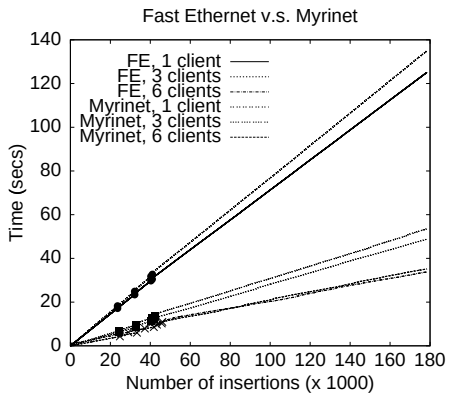

Figure 6: Comparing the build times for Fast Ethernet (FE) and Myrinet for several client configurations (with 3 starting servers).

Both experiments use one starting server, and the data points on the curves again refer to global splits. The obtained performance for 4 clients can be ordered in decreasing order as 2, 4, 1 and 8 servers (where the results of 2 and 4 servers are nearly identical). Interesting in this respect is that allowing the SDDS to grow up to 8 servers gives the worst performance. This is because the overhead of the global splits overwhelms the improvement in insertion time due to enhanced parallelism. Note that improving our data shipping scheme for global splits (using bulk shipments instead of shipping single data elements) will affect the performance trade-off between splitting penalties and enhanced parallelism.

When increasing the number of clients (see Figure 5b), the performance order after 180K insertions for the same experiment becomes 4, 2, 8 and 1 (again in decreasing order). We observe that the performance ranking of the configurations with a higher number of maximum servers (4 and 8) has improved. From this we can conclude that the larger number of clients shifts the balance of the trade-off between global split penalties and the increase of parallelism as they tend to favor a higher degree of parallelism. More research is needed to understand how to optimize this performance trade-off for different numbers of clients.

In Figure 6, the LH*LH build times are shown for both Fast Ethernet and Myrinet with the default 1 Kb data elements. The results are for three starting servers and a maximum of eight servers. We were unable to perform the experiment with 8 clients due to a faulty Myrinet network interface, which explains the chosen client configurations (1, 3 and 6 clients). Comparing Fast Ethernet and Myrinet, we observed slightly better performance for Fast Ethernet. At first sight, this may be surprising because of Myrinet's higher bandwidth, but Myrinet also suffers from a higher communication latency than Fast Ethernet. The results therefore suggest that the message roundtrip time of insertion requests is dominated by the network latency rather than the network bandwidth. Experiments with larger data sizes (not shown here) indicate that only for very large data elements (in the order of megabytes) Myrinet starts to outperform Fast Ethernet. Of course, our shipping scheme in which every single data element separately incurs communication latency, does not contribute to a good performance in the case of Myrinet.

5. CONCLUSIONS

In this paper, we presented a performance evaluation of a prototype implementation of the LH*LH distributed data structure for a cluster of workstations. Our prototype has demonstrated to be truly scalable as the time to insert data elements is independent of the size of the data structure. In the case of sequential access to the data structure, the average insertion time has been found to be an order of magnitude faster than a typical disk access. Insertion times can even be reduced by using multiple clients which concurrently insert data into the distributed data structure. Our results show that for a maximum of 8 servers the speedup of insertions scales reasonably well up to about 5 clients.

We also found that the insertion performance of our prototype implementation is dominated by the message roundtrip time and in particular by the network latency. This explains, for example, why our experiments on a Myrinet network only outperform the experiments on a Fast Ethernet network for very large data sizes (as Myrinet provides high bandwidth but also suffers from higher network latencies).

Future work will extend our experiments to include more aspects of LH*LH's functionality. An example is the optimization of the global-split threshold function by studying the incorporation of parameters like the available memory of the hosts (as clusters often are heterogeneous) and the number of clients accessing the SDDS. The latter has shown to affect the performance trade-off between global split penalties and the increase of parallelism by adding servers.

6. REFERENCES

[1] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proc. of the 4th Int. Conference on Foundations of Data Organization and Algorithms*, 1993.

[2] J. S. Karlsson. A scalable data structure for a parallel data server. Master's thesis, Dept. of Comp. and Inf. Science, Linköping University, Feb. 1997.

[3] J. S. Karlsson, W. Litwin, and T. Risch. LH*lh: A scalable high performance data structure for switched multicomputers. In *Advances in Database Technology*, pages 573–591, March 1996.

[4] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of VLDB*, 1980.

[5] W. Litwin, M.-A. Neimat, and D. Schneider. LH*: A scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–526, Dec. 1996.

[6] M. Modi, V. Gupta, and A. D. Pimentel. LH*lh: Implementation on a cluster of workstations. Technical report, Dept. of Computer Science, University of Amsterdam, July 2000.

[7] A. D. Pimentel and L. O. Hertzberger. Evaluation of LH*lh for a multicomputer architecture. In *Proc. of the EuroPar Conference*, pages 217–229, Sept. 1999.

[8] R. Vingralek, Y. Breitbart, and G. Weikum. Distributed file organisation with scalable cost/performance. In *Proc. of ACM-SIGMOD*, May 1994.