LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers

Jonas S Karlsson
Email: jonka@ida.liu.se

Witold Litwin
Email: litwin@cidmac.dauphine.fr

Tore Risch
Email: torri@ida.liu.se

Abstract

LH*LH is a new data structure for scalable high-performance hash files on the increasingly popular switched multicomputers, i.e., MIMD multiprocessor machines with distributed RAM memory and without shared memory. An LH*LH file scales up gracefully over available processors and the distributed memory, easily reaching Gbytes. Address calculus does not require any centralized component that could lead to a hot- spot. Access times to the file can be under a millisecond and the file can be used in parallel by several client processors. We show the LH*LH design, and report on the performance analysis. This includes experiments on the Parsytec GC/PowerPlus multicomputer with up to 128 Power PCs and 32 MB of distributed RAM per node. We prove the efficiency of the method and justify various algorithmic choices that were made. LH*LH opens a new perspective for high-performance applications, especially for the database management of new types of data and in real-time environments.

This work has been supported by the Swedish National Board for Industrial and Technical Development (NUTEK) and Center for Industrial Information Technology (CENIIT).

This report has been accepted to EDBT-96 (Extended Database Technology), in Avinon, France, March 1996.

IDA Technical Report 1995
LiTH-IDA-R-95-25
ISSN-0281-4250

Department of Computer
and Information Science,
Linköping University,
S-581 83 Linkping, Sweden

LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers

Jonas S Karlsson
Email: jonka@ida.liu.se

Witold Litwin
Email: litwin@cidmac.dauphine.fr

Tore Risch
Email: torri@ida.liu.se

September 27, 1995

Abstract

LH*LH is a new data structure for scalable high-performance hash files on the increasingly popular switched multicomputers, i.e., MIMD multiprocessor machines with distributed RAM memory and without shared memory. An LH*LH file scales up gracefully over available processors and the distributed memory, easily reaching Gbytes. Address calculus does not require any centralized component that could lead to a hot- spot. Access times to the file can be under a millisecond and the file can be used in parallel by several client processors. We show the LH*LH design, and report on the performance analysis. This includes experiments on the Parsytec GC/PowerPlus multicomputer with up to 128 Power PCs and 32 MB of distributed RAM per node. We prove the efficiency of the method and justify various algorithmic choices that were made. LH*LH opens a new perspective for high-performance applications, especially for the database management of new types of data and in real-time environments.

1 Introduction

New applications of databases require increased performance. One way is to use parallel and distributed architectures [17][2]. The *multicomputers*, i.e., networks of multiple CPUs with local storage become a popular hardware platform for this purpose[17][2][21]. In particular, multicomputer files need to be able to scale to large sizes over the distributed storage, especially the RAM. The *Scalable Distributed Data Structures* (SDDSs)[15] is an approach towards this goal. An SDDS file can gracefully expand with the inserts from a single storage site to as many as needed, e.g., thousands, appended dynamically to the file. The data sites termed *servers* can be used from any number of autonomous sites termed *clients*. To avoid a hot-spot, there is no central directory for the addressing accross the current structure of the file. Each client has its own *image* of this structure. An image can become outdated when the file expands. The client may then send a request to an incorrect server. The servers forward such requests, possible in several steps, towards the correct address. The correct server appends to the reply a special message to the client, called *Image Adjustment Message* (IAM). The client adjusts its image, avoiding to repeat the error. A well designed SDDS should make addressing errors occasional and forwards few, and should provide for the scalability of the access performance when the file grows.

Up to now, the design of SDDSs was aimed at *network* multicomputers constituted of autonomous PCs and WSs linked through a local network. A promising type of multicomputer is also *shared-nothing multiprocessor multicomputers*, also called *switched multicomputers* (SM) [21]. Both types of multicomputers share the idea of cooperating autonomous CPUs communicating through message passing. This suggests that an SDDS could be useful for an SM as well. We have developed and implemented a variant of LH*, which we call LH*LH, designed specifically for this purpose. Performance analysis showed that LH*LH should be an attractive data structure for CPU and RAM intensive multiprocessor applications.

LH*LH allows for scalable RAM files spanning over several CPUs of an SM and its RAMs. On our testbed machine, a Parsytec GC/PowerPlus with 64 nodes of 32 MB RAM each, a RAM file can scale up to almost 2 GB with an average load factor of 70%. A file may be created and searched by several (client) CPUs concurrently. The access times may be about as fast as the communication network allows it to be. On our testbed, the average time per insert is as low as 1.2 ms per client. Eight clients building a file concurrently reach a throughput of 2500 inserts/second i.e., 400 μs/insert. These access times are more than an order of magnitude better than the best ones with the current disk file technology and will probably never be reached by mechanical devices.

Below we present the LH*LH design and performance. With respect to LH* [15], LH*LH is characterized by several original features. Its overall architecture is geared towards an SM while that of LH* was designed for a network multicomputer. Then, the design of LH*LH involves local bucket management while in [15] this aspect of LH* design was left for further study. In LH*LH one uses for this purpose a modified version of main-memory Linear Hashing defined in [19] on the basis of [11]. An interesting interaction between LH and LH* appears, allowing for much more efficient LH* bucket splitting. The reason is that LH*LH allows the splitting of LH*-buckets without visiting individual keys.

The average access time is of primary importance for any SDDS on a network computer or SM. Minimizing the worst case is, however, probably more important for an SM where processers work more tightly connected than in a network computer. The worst case for LH* occurs when a client accesses a bucket undergoing a split. LH* splits should be infrequent in practice since buckets should be rather large. In the basic LH* schema, a client's request simply waits at the server till the split ends. In the Parsytec context, performance measurements show that this approach may easily lead to several seconds per split, e.g. three to seven seconds in our experiences (as compared to $1-2$ msec per request on the average). Such a variance would be detrimental to many SM applications.

LH*LH is therefore provided with an enhanced splitting schema, termed *concurrent splitting*. It is based on ideas sketched in [14] allowing for the client's request to be dealt with while the split is in progress. Several concurrent splitting schemes were designed and experimented with. Our performance studies shows superiority of one of these schemes, termed concurrent splitting with bulk shipping. The maximal response time of an insert while a split occurs decreases by a factor of three hundred to a thousand times. As we report in what follows, it becomes about 7 msec for one active client in our experiences and 25 msec for a file in use by eight clients. The latter value is due to interference among clients requesting simultaneous access to the server splitting.

Given the space limitations, in what follows we assume basic knowledge of LH* as in [15], and of LH as defined in [13]. Section 2 discusses related work. Section 3 presents the Parsytec machine. Section 4 describes LH*LH. Section 5 shows performance study. Section 6 concludes the paper.

2 Related work

In traditional distributed files systems, in implementations like NFS or AFS, a file resides entirely at one specific site. This gives obvious limitations not only on the size of the file but also on the access performance scalability. To overcome these limitations distributions over multiple sites have been used. One example of such a scheme is *round-robin* [1] where records of a file are evenly distributed by rotating through the nodes when records are inserted. The *hash-declustering* [8] assigns records to nodes on basis of a hashing function. The *range-partitioning* [4] divides key values into ranges and different ranges are assigned to different nodes. All these schemes are *static* which means that the declustering criterion does not change over time. Hence, updating a directory or declustering function is not required. The price to pay is that the file cannot expand over more sites than initially allocated.

To overcome this limitation of static schemes, the dynamic partitioning started appearing. The first such scheme is DLH [20]. This scheme was designed for a shared memory system. In DLH, the file is in RAM and the file parameters are cached in the local memory of each processor. The caches are refreshed selectively when addressing errors occur and through atomic updates to all the local memories at some points. DLH shows impressively efficient for high insert rates.

SDDSs were proposed for distributing files in the network multicomputer environment, hence without a shared memory. The first scheme was LH* [15]. Distributed Dynamic Hashing (DDH) [3] is another SDDS, it is based on Dynamic Hashing [10]. The idea with respect to LH* is that DDH allows greater splitting autonomy by immediately splitting overflowing buckets. One drawback is that while LH* limits the number of forwardings to two[1] when the client makes an addressing error, DDH may use $O(\log_2 N)$ forwardings, where N is the number of buckets in the DDH file.

Another SDDS has been defined in [22]. It extends LH* and DDH to more efficiently control the load of a file. The main idea is to manage several buckets of a file per server while LH* and DDH have basically only one bucket per server. One also controls the server load as opposed to bucket load for LH*.

Finally, in [9] and in [16] SDDSs for (primary key) ordered files are proposed. In [9] the access computations on the clients and servers use a distributed binary search tree. The SDDSs in [16], collectively termed RP*, use broadcast or distributed n-ary trees. It is shown that both kinds of SDDSs allow for much larger and faster files than the traditional ones.

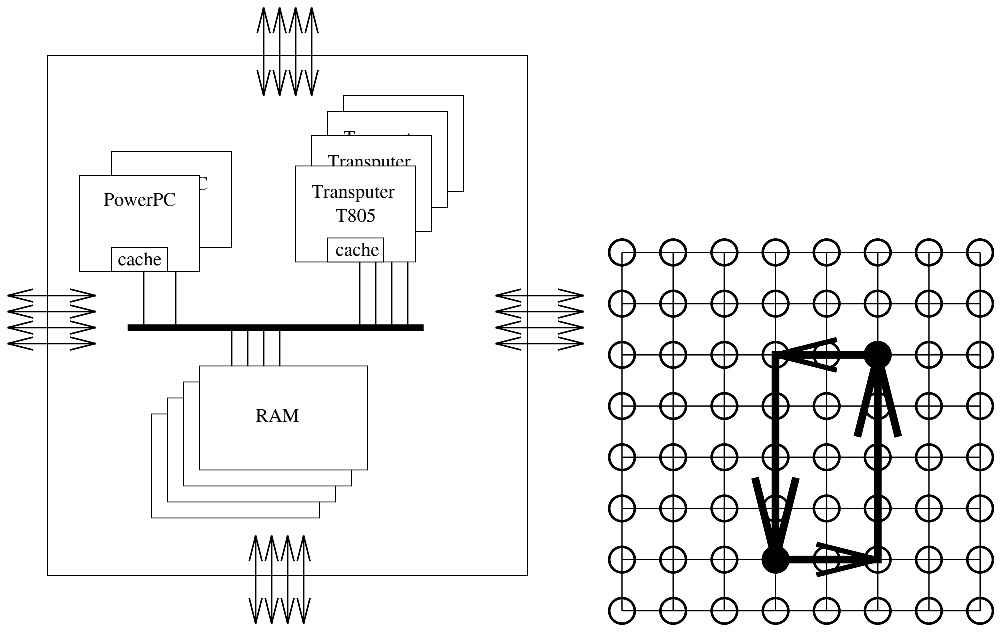
3 The Parsytec multicomputer

The Parsytec GC/PowerPlus architecture (Figure 1) is massively parallel with distributed memory, also know as MIMD (Multiple Instruction Multiple Data). The machine used for the LH*$_{\text{LH}}$ implementation has 128 PowerPC-601 RISC-processors, constituting 64 nodes. One node is shown in Figure 1a. Each node has 32 MB of memory shared between two PowerPC processors and four T805 Transputer processors. The latter are used for communication. Each Transputer has four bidirectional communication links. The nodes are connected through a bidirectional fat (multiple) grid network with packet message routing.

The communication is point-to-point. The software libraries [18] support both synchronous and asynchronous communication and some other types of communication, e.g. mailboxes.

The response time of a communication depends on the actual machine topology. The closer the communicating nodes are the faster is the response. Routing is done

[1] In theory, communication delays could trigger more forwarding [22].



(a) One node. (b) 64 nodes interconnection grid.

Figure 1: The Parsytec architecture.

statically by the hardware as in Figure 1b with the packages first routed in the horizontal direction.

4 LH*LH Overview

4.1 Overall Architecture

An LH*LH-*client* is a process that accesses an LH*LH file on the behalf of the application. An LH*LH-*server* at a node stores data of LH*LH files. An application can use several clients to explore a file. This way of processing increases the throughput, as will be shown in Section 5. Both clients and servers are created dynamically. The allocation of clients start from the higher numbered nodes. The servers are allocated from the lower nodes, as in Figure 2a.

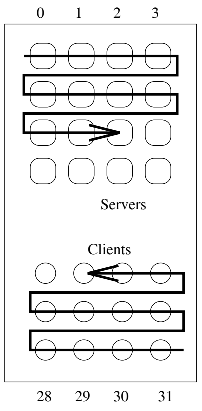
At a server, one *bucket* per LH* file contains the stored data. The bucket management is described in Section 4.5. The file starts at one server and expands to others when it overloads the buckets already used.

4.2 LH* addressing scheme

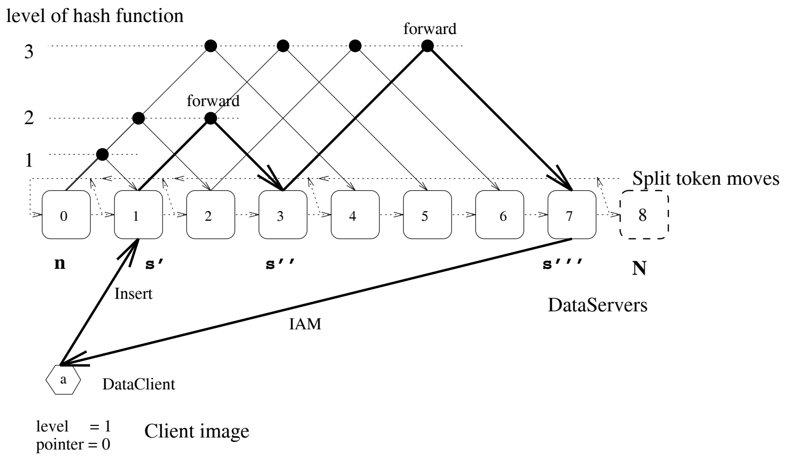
The global addressing rule in LH*LH file is that every key C is inserted to the server s_C whose address $s = 0, 1, ...N-1$ is given by the following LH addressing algorithm [13]:

$$s_C := h_i(C)$$

$$\text{if } s_C < \text{ then } s_C := h_{i+1}(C),$$



(a) Allocation of clients and servers.



(b) LH* File Expansion Scheme

Figure 2: Clients and Servers.

where i (LH* file level) and n (split pointer address) are file parameters evolving with splits. The h_i functions are basically:

$$h_i(C) = C \bmod (2^i \times K), K = 1, 2, ..$$

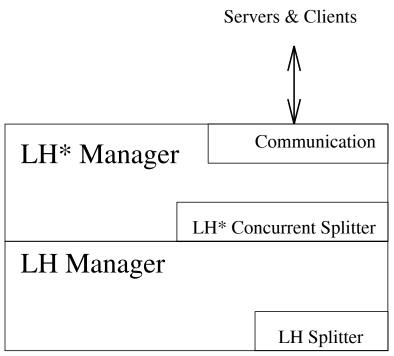
and $K = 1$ in what follows. No client of an LH* file knows the current i and n of the file. Every client has its own *image* of these values, let it be i' and n'; typically $i' \leq i$ [15]. The client sends the query, e.g. the insert of key C, to the address $s'_C(i', n')$.

The server s'_C verifies upon query reception whether its own address s'_C is $s'_C = s_C$ using a short algorithm stated in [15]. If so the server processes the query. Otherwise, it calculates a forwarding address s''_C using the forwarding algorithm in [15] and sends the query to server s''_C. Server s''_C acts as s'_C and perhaps resends the query to server s'''_C as shown for Server 1 in Figure 2b. It is proven in [15] that then s'''_C must be the correct server. In every case, of forwarding, the correct server sends to the client an Image Adjustment Message (IAM) containing the level i of the correct server. Knowing the i and the s_C address, the client adjusts its i' and n' (see [15]) and from now on will send C directly to s_C.

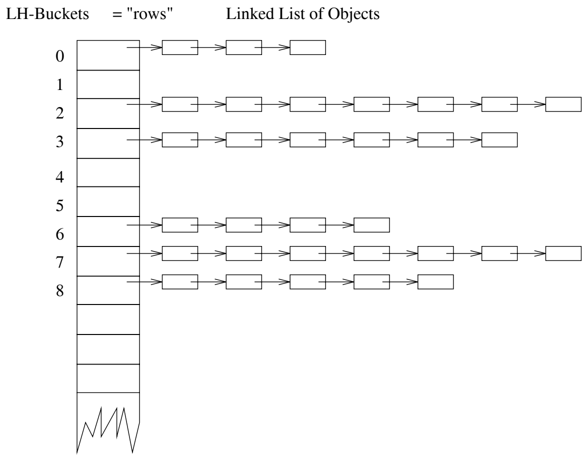
4.3 LH* File Expansion

LH* file expands through bucket splits as in Figure 2. The bucket next to split is generally noted bucket n, $n = 0$ in the figure. Each bucket keeps the value of i used (called LH*-bucket level) in its header starting from $i = 0$ for bucket 0 when the file is created. Bucket n splits through the replacement of h_i with h_{i+1} for every C it contains. As result, typically half of its records move to a new bucket N, appended to the file with address $n + 2^i$. In Figure 2, one has $N = 8$. After the split, n is set to $(n + 1) \bmod 2^i$. The successive values of n can thus be seen as a linear move of a *split token* through the addresses $0, 0, 1, 0, 1, 2, 3, 0, ..., 2^i - 1, 0, ...$. The arrows of Figure 2 show both the token moves and a new bucket address for every split, as resulting from this scheme.

There are many strategies, called *split control* strategies, that one can use to decide when a bucket should split [14] [13] [22]. The overall goal is to avoid the file



(a) Data Server.



(b) LH-structure.

Figure 3: The Data Server and the LH-structure.

to overload. As no LH* bucket can know the global load, one way to proceed is to fix some threshold S on a bucket load [14]. Bucket n splits when it gets an insert and the actual number of objects it stores is at least S. S can be fixed as a file parameter. A potentially more performant strategy for an SM environment is to calculate S dynamically through the following formula:

$$S = M \times V \times \frac{2^i + n}{2^i},$$

where i is the LH*-bucket level, M is a file parameter, and V is the bucket capacity in number of objects. Typically one sets M to some value between 0.7 and 0.9.

The performance analysis in Section 5.1 shows indeed that the dynamic strategy should be preferred in our context. This is the strategy adopted for LH*LH.

4.4 Communication Mode

In the LH*LH implementation on the Parsytec machine a server receiving a request must have issued the *receive* call before the client can do any further processing. This well known *rendezvous* technique enforces entry flow control on the servers, preventing the clients from working much faster than the server could accept requests[2]. Insert operations do not give any specific acknowledge messages by the LH* manager since communication is "safe" on the Parsytec machine (if send returns ok the message is guaranteed to be received). IAMs, split messages with the split token, and general service messages use the asynchronous type of communication.

4.5 Server architecture

The server consists of two layers, as shown in Figure 3a. The LH*-Manager handles communications and concurrent splits. The LH-Manager manages the objects in the bucket. It uses the Linear Hashing algorithm [12].

[2]The overloaded server could run out of memory space and could send outdated IAMs [6].

4.5.1 The LH Manager

LH creates files able to grow and shrink gracefully on a site. In our implementation, the LH-manager stores all data in the main memory. The LH variant used is a modified implementation of Main Memory Linear Hashing [19].

The LH file in an LH*-bucket (Figure 3b) essentially contains (i) a header with the *LH-level*, an *LH-splitting pointer*, and the count x of objects stored, and (ii) a dynamic array of pointers to LH-buckets, and (iii) LH-buckets with records. An LH-bucket is implemented as a linked list of the records. Each record contains the calculated hash value, called a *pseudo-key*. Both the pointer to the actual key, and the pointer to the object are stored as bitstrings. Pseudo-keys make the rehashing faster. An LH-bucket split occurs when $L = 1$, with:

$$L = \frac{x}{b \times m},$$

where b is the number of buckets in the LH file, and m is a file parameter to control the required mean number of objects in an LH-bucket (linked list).

4.5.2 LH* partitioning of an LH file

The use of LH allows the LH* splitting in a particularly efficient way. The reason is that individual keys are not visited for rehashing. Figure 4 and Figure 5 illustrates the ideas.

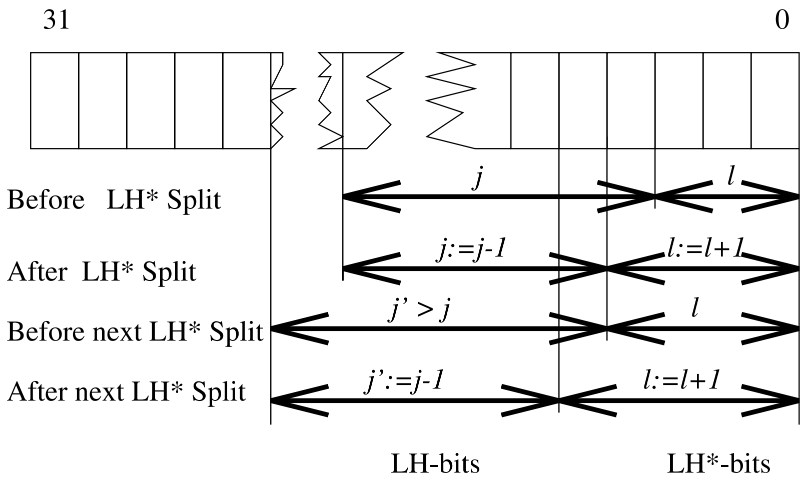


Figure 4: Pseudo-key usage by LH and LH*.

LH and LH* share the pseudo-key. The pseudo-key has J bits, in Figure 4; $J = 32$ at every bucket. LH* uses the lower l bits $(b_{l-1}, b_{l-2}, ...b_0)$. LH uses j bits $(b_{j+l-2}, b_{j+l-3}, ...b_l)$, where $j + l \leq J$. During an LH*-split l increases by one whereas j decreases by one. The value of the new lth bit determines whether an LH-bucket is to be shipped. Only the odd LH-buckets i.e. with $b_l = 1$ are shipped to the new LH*-bucket N. The array of the remaining LH-buckets is compacted, the count of objects is adjusted, the LH-bucket level is decreased by one (LH uses one bit less), and the split pointer is halved. Figure 5 illustrates this process.

Further inserts to the bucket may lead to any number of new LH splits, increasing j in Figure 4 to some j'. Next LH* split of the bucket will then decrease j' to $j' := j' - 1$, and set $l := l + 1$ again.

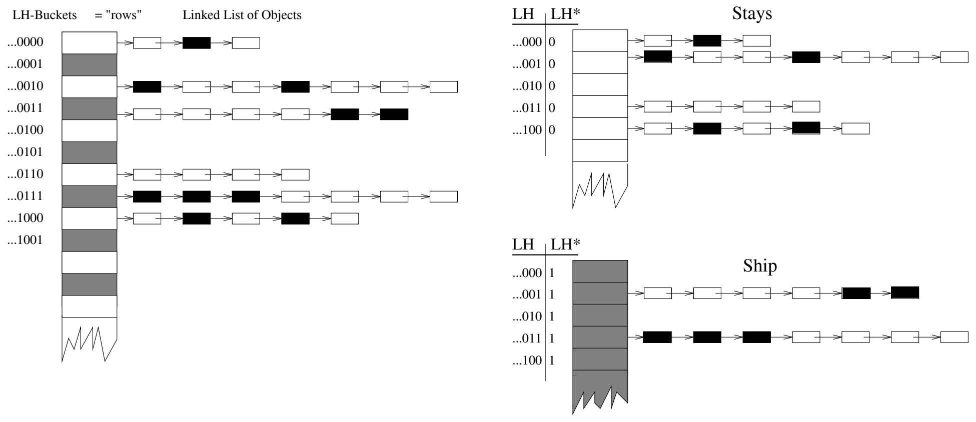


Figure 5: Partitioning of an LH-file by LH* splitting.

4.5.3 Concurrent Request Processing and Splitting

A split is a much longer operation than a search or an insert. The split should also be atomic for the clients. Basic LH* [15] simply requires the client to wait till the split finishes. For the high-performance applications on an SM multicomputer it is fundamental that the server processes a split concurrently with searches and inserts. This is achieved as follows in LH*LH.

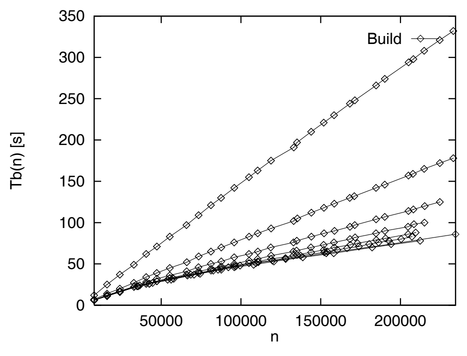
Requests received by the server undergoing a split are processed as if the server had not started splitting, with one exception: a request that concerns parts of the local LH structure processed by the Splitter is queued to be processed by the Splitter.

The Splitter processes the queue of requests since these requests concern LH-buckets of objects that have been or are being shipped. If the request concerns an LH-bucket that has already been shipped the request is forwarded, since the data is guaranteed to arrive at the destination. If the request, concerns an LH-bucket not yet shipped it is processed in the local LH table as usual. The requests that concerns the current LH-bucket being shipped is first searched among the remaining objects in that LH-bucket. If not found there it is forwarded by the Splitter. All forwardings are serialized within the Splitter task.

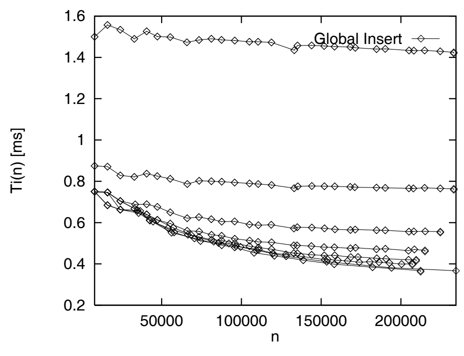
4.5.4 Shipping

Shipping means transferring the objects selected during the LH*-bucket split to the newly appended bucket N. In LH* [14] the shipping was assumed basically to be of the *bulk* type with all the objects packed into a single message. After shipping has been completed, bucket N sends back a *commit message*. In LH*LH there is no need for the commit message. The communcation is safe, and the sender's data cannot be updated before the shipping is entirely received. In particular, no client can directly access bucket N before the split is complete.

In the LH*LH environment there are several resons for not shipping too many objects in a message, especially all the objects in a single message. Packing and unpacking objects into a message requires CPU time and memory transfers, as objects are not stored contiguously in the memory. One also needs buffers of sizes at least proportional to the message size, and a longer occupation of the communication subsystem. Sending objects individually simplifies these aspects but generates more messages and more overhead time in the dialog with the communication subsystem. It does not seem that one can decide easily which strategy is finally more effective in practice.



(a) Build time



(b) Global insert time

Figure 6: Build and insert time for LH*LH for different number of clients (1-8).

The performance analysis in Section 5.2 motivated the corresponding design choice for LH*LH. The approach is that of bulk shipping but with a limited message size. At least one object is shipped per message and at most one LH-bucket. The message size is a parameter allowing for an application dependent packing factor. For the test data using bulks of a dozen of records per shipment showed to be much more effective than the individual shipping.

5 Performance evaluation

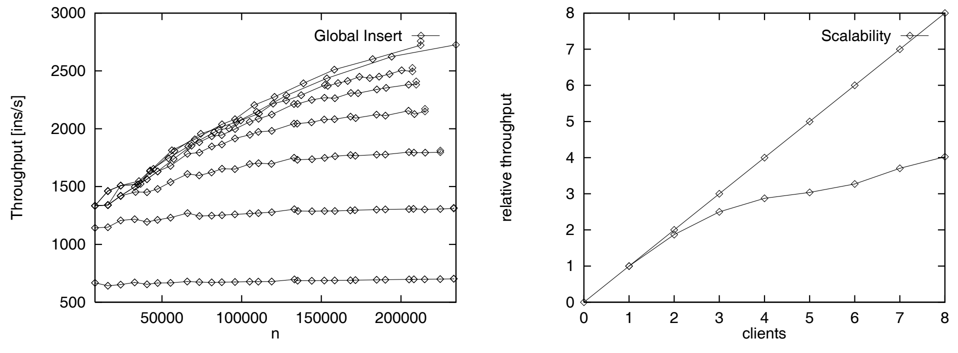
The access performance of our implementation was studied experimentally. The measurements below show elapsed times of various operations and their scalability. Each experiment consists of a series of inserts creating an LH* file. The number of clients, the file parameters M and m, and the size of the objects are LH*LH parameters.

At the time when the tests were performed only 32 nodes were available at our site. The clients are allocated downwards from node 31 and downwards and servers from node 0 and upwards. The clients read the test data (a random list of words) from the file system in advance to avoid that the I/O disturbs the measurements. Then the clients start inserting their data, creating the example LH*LH-file. When a client sends a request to the server it continues with the next item only when the request has been accepted by the server (rendezvous). Each time before the LH* file is split measures are collected by the splitting server. Some measurements are also collected at some client, especially timing values for each of that client's requests.

5.1 Scalability

Figure 6a plots the elapsed time to constitute the example LH*LH file through n inserts; $n = 1, 2..N$ and $N = 235.000$; performed simultaneously by k clients $k = 1, 2..8$. This time is called *build time* and is noted $Tb(n)$, or $Tb^k(N)$ with k as a parameter. In Figure 6a, $Tb(N)$ is measured in seconds. Each point in a curve corresponds to a split. The splits were performed using the concurrent splitting with the dynamic control and the bulk shipping. The upper curve is $Tb^1(n)$. Next lower curve is $Tb^2(n)$, etc., until $Tb^8(n)$.

The curves show that each $Tb^k(n)$ scales-up about linearly with the file size n. This is close to the ideal result. Also, using more clients to build the file, uniformly



(a) Throughput (b) Scalability

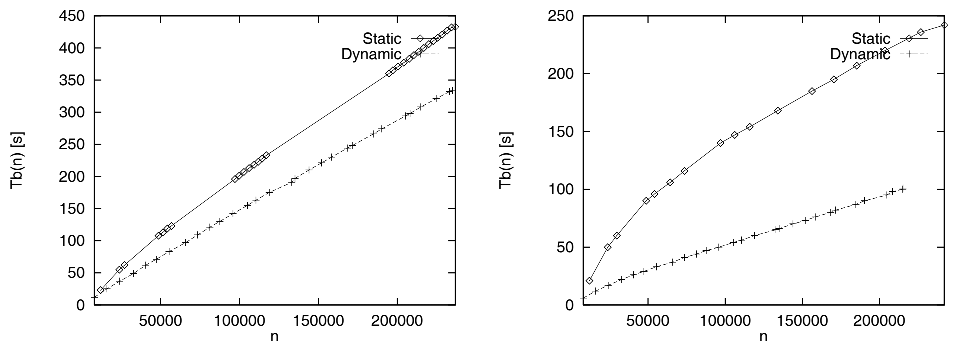
Figure 7: Throughput scale-up for different number of clients.

decreases Tb^k, i.e., $k' > k'' -> Tb^{k'}(n) \leq Tb\ k''(n)$ for every n. Using two clients almost halves Tb, especially $Tb(N)$, from $Tb^1(N) = 330$ sec to $Tb^2(N) = 170$ sec. Building the file through eight clients decreases Tb further, by a factor of four. $Tb(N)$ becomes only $Tb^8(N) = 80$ sec. While this is in practice an excellent performance, the ideal scale-up could reach k times, i.e., the build time $Tb^8(N) = 40$ sec only. The difference results from various communication and processing delays at a server shared by several clients, discussed in the previous sections and in what follows.

Figure 6b plots the curves of the global insert time $Ti^k(n) = Tb^k(n)/n$ [msec]. Ti measures the average time of an insert from the perspective of the application building the file on the multicomputer. The internal mechanics of LH*LH file is transparent at this level including the distribution of the inserts among the k clients and several servers, the corresponding parallelism of some inserts, the splits etc. The values of n, N and k are those from Figure 6a. To increase k improves Ti in the same way as for Tb. The curves are also about as linear, constant in fact, as they should be. Higly interestingly, and perhaps unexpectedly, each $Tb^k(n)$ even decreases when n grows, the gradient increasing with k. One reason is the increasing number of servers of a growing file, leading to fewer requests per server. Also, our allocation schema decreases the mean distance through the net between the servers and the clients of the file.

The overall result is that Ti always is under 1.6 msec. Increasing k uniformly decreases Ti, until $Ti^8(n) < 0.8$ msec, and $Ti^8(N) < 0.4$ msec. These values are about ten to twenty times smaller than access times to a disk file, typically over 10 msec per insert or search. They are likely to remain forever beyond the reach of any storage on a mechanical device. On the other hand, a faster net and more efficient communication subsystem than the one used should allow for even much smaller Ti's, in the order of dozens of μsecs [14] [16].

Figure 7a plots the global throughput $T^k(n)$ defined as $T^k(n) = 1/Ti(n)[i/sec]$ (inserts per second). The curves express again an almost linear scalability with n. For the reasons above discussed, T^k even increases for larger files, up to 2700 i/sec. An increase of k also uniformly increases T for every n. To see the throughput scalability more clearly, Figure 7b plots the relative throughput $Tr(k) = T^k(n)/T^1(n)$ for a large n; $n = N$. One compares Tr to the plot of the ideal scale-up that is simply $T'r(k) = k$. The communication and service delays we spoke about clearly play an increasing role when k increases. Although Tr monotonically increases with k, it diverges more and more from $T'r$. For $k = 8$, one has $Tr = 4$ which is only the half of the ideal scale-up. It means that the actual throughput per client,



(a) One client (b) Four clients

Figure 8: Static and dynamic split control.

$Tc^k(n) = T^k(n)/k$, comparatively also decreases until the half of the throughput T^1 of a single client.

Figure 8 shows the comparative study of the dynamic and the static split control strategies. The plots show build times, let it be $Tb'(n)$ for the static control and $Tb(n)$ for the dynamic one. The curves correspond to the constitution of our example file, with $k = 1$ in Figure 8a and $k = 4$ in Figure 8b. The plots Tb are the same as in Figure 6a. Figure 8 clearly justifies our choice of the dynamic control strategy. Static control uniformly leads to the longer build time, i.e., for every n and k one has $Tb'(n) > Tb(n)$. The relative difference $(Tb' - Tb)/Tb$ reaches 30% for $k = 1$, e.g. $Tb'(N) = 440$ and $Tb(N) = 340$. For $k = 4$ the dynamic strategy more than halves the build time, e.g from 230 to 100 sec.

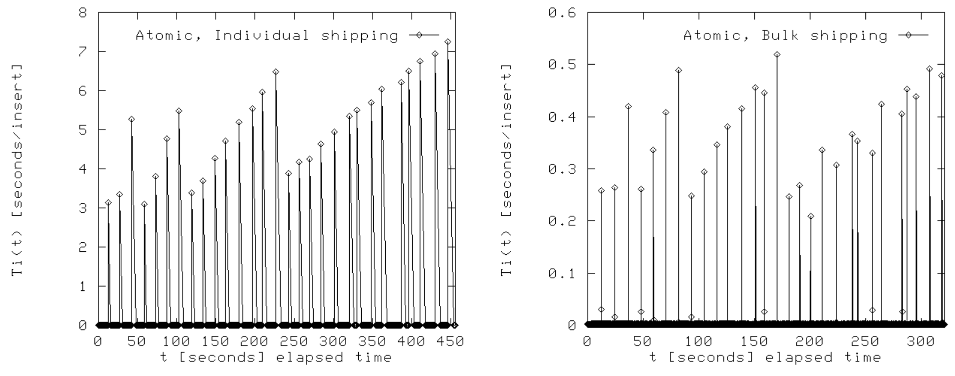
Note that the dynamic strategy also generates splits generally more uniformly over the inserts, particularly for $k = 1$. The static strategy leads to short periods when a few inserts generate splits of about every bucket. This creates a heavier load on the communication system and increases the insert and search times during that period.

5.2 Efficiency of Concurrent Splitting

Figure 9 shows the study of comparative efficiency of individual and bulk shipping for LH* atomic splitting (non-concurrent), as described earlier. The curves plot the insert time $Ti^1(t)$ measured at t seconds during the constitution of the example file by a single client. A bulk message contains at most all the records constituting an LH-bucket to ship. In this experiment there are 14 records per bulk on the average. A peak corresponds to a split in progress, when an insert gets blocked till the split ends.

The average insert time beyond the peaks is 1.3 msec. The corresponding Ti's are barely visible at the bottom of the plots. The individual shipping, Figure 9a, leads to a peak of $Ti = 7.3$ sec. The bulk shipping plot, Figure 9b, shows the highest peak of $Ti = 0.52$ sec, i.e., 14 times smaller. The overall build time $Tb(N)$ decreases also by about 1/3, from 450 sec in Figure 9a, to 320 sec in Figure 9b. The figures clearly prove the utility of the bulk shipping.

Observe that the maximal peak size got reduced accordingly to the bulk size. It means that larger bulks improve the access performance. However, such bulks require also more storage for themselves as well as for the intermediate communication buffers and more CPU for the bulk assembly and disassembly. To choose



(a) Individual Shipping

(b) Bulk Shipping

Figure 9: Efficiency of (a) individual and (b) bulk shipping.

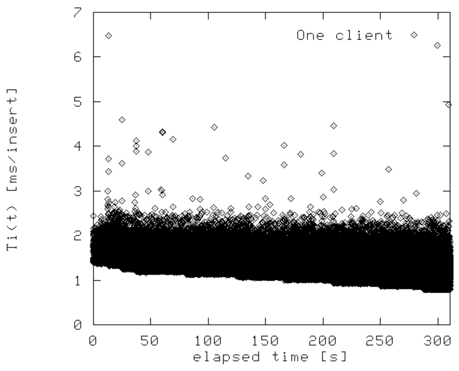
the best bulk size in practice, one has to weight all these factors depending on the application and the hardware used.

Figure 10 shows the results of the study where the bulk shipping from Figure 9 is finally combined with the concurrent splitting. Each plot $Ti(t)$ shows the evolution of the insert time at one selected client among k clients; $k = 1..4, 8$; concurrently building the example file with the same insert rate per client. The peaks at the figures correspond again to the splits in progress but they are much lower. For $k = 1$, they are under 7 msec, and for $k = 8$ they reach 25 msec. The worst insert time with respect to Figure 9 improves thus by a factor of 70 for $k = 1$ and of 20 for $k = 4$. This result clearly justifies the utility of the concurrent splitting and our overall design of the splitting algorithm of LH*LH.

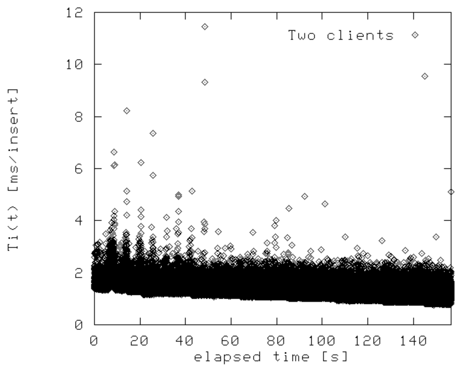
The plots in Figures 10a to 10e show the tendency towards higher peaks of Ti, as well as towards higher global average and variances of Ti over $Ti(t)$, when more clients build the file. The plot in Figure 10f confirms this tendency for the average and the variance. Figures 10d and 10e show also that the insert times become especially affected when the file is still small, as one can see for $t < 10$ in these figures. All these phenomena are due to more clients per server for a larger k. A client has then to wait more for the service. A greater k is nevertheless advantageous for the global throughput as it was shown earlier.

Figure 10 hardly allows to see the tendency of the insert time when the file scales up, as non-peak values are buried in the black areas. Figure 11 plots therefore the evolution of the corresponding *marginal client insert time* Tm^k. Tm^k is computed as an average over a sliding window of 500 inserts plotted in Figure 10. The averaging smoothes the variability of successive values giving the black areas in Figure 10. The plots $Tm^k(t)$ show that the insert times not only do not deteriorate when the file grows, but even improve. Tm^1 decreases from 1.65 msec to under 1.2 msec, and Tm^8 from 8 msec to 1.5 msec. This nice behavior is due again to the increase in the number of servers and to the decreasing distance between the clients and the servers.

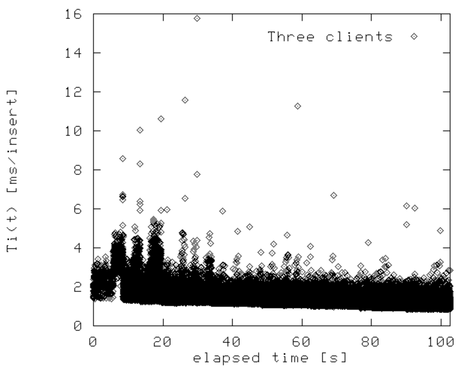
The plots show also that $Tm^k(t)$ uniformly increases with k, i.e. $k'' > k' \rightarrow Tm^{k''}(t) > Tm^{k'}(t)$, for every t. This phenomena is due to the increased load of each server. Also interestingly, the shape of Tm^k becomes stepwise, for greater k's, with insert times about halving at each new step. A step corresponds to a split token trip at some level i. The drop occurs when the last bucket of level i splits and the split token comes back to bucket 0. This tendency seems to show that the serialization



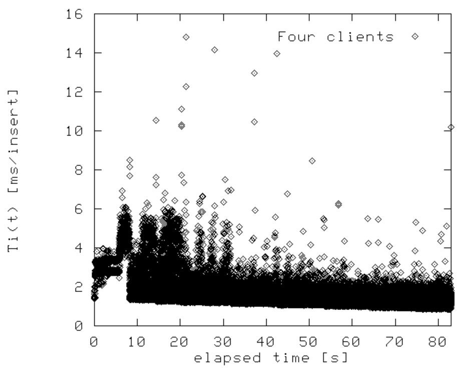
(a) One active client



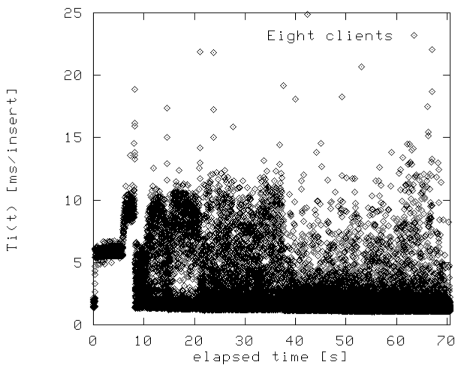
(b) Two active clients



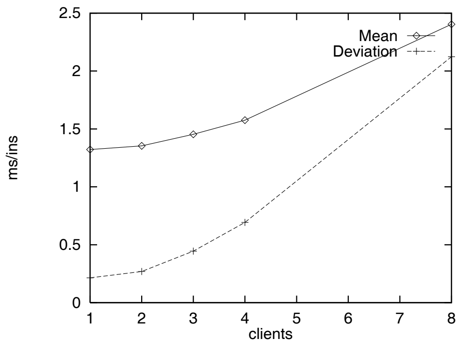
(c) Three active clients



(d) Four active clients

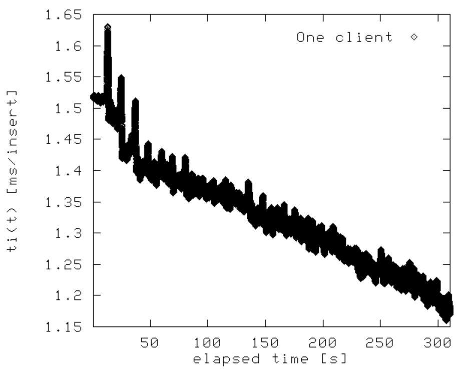


(e) Eight active clients

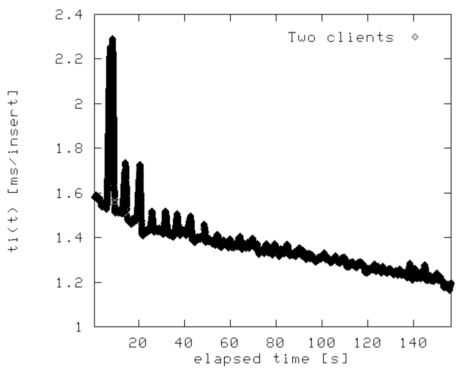


(f) Average, std. deviation

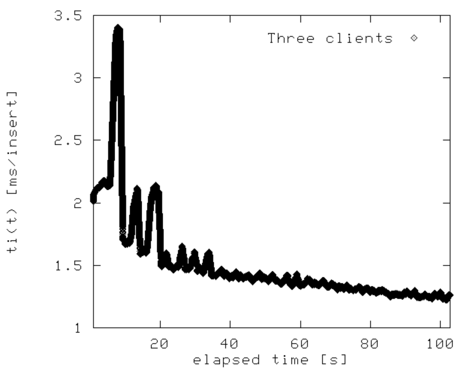
Figure 10: Efficiency of the concurrent splitting.



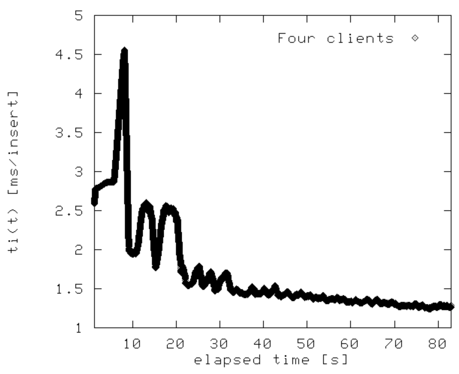
(a) One active client



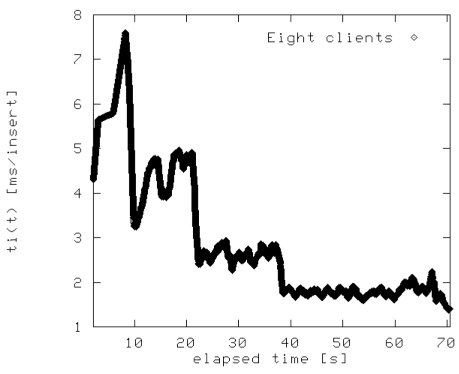
(b) Two active clients



(c) Three active clients



(d) Four active clients



(e) Eight active clients

Figure 11: LH*LH client insert time scalability.

of inserts contributing most to a Tm^k value occurs mainly at the buckets that are not yet split.

The overall conclusion from Figure 11 is that the insert times at a client of a file equally shared among k clients, is basically either always under 2 msec, for $k = 1$, or tends to be under this time when the file enlarges. Again this performance shows excellent scale-up behavior of LH*LH. The performance is in particular largely superior to the one of a typical disk file used in a similar way. For $k = 8$ clients, for example, the speed-up factor could reach 40 times, i.e., 2 msec versus $8 * 10$ msec.

6 Conclusions

Switched multicomputers such as the Parsytec GC/PowerPlus are powerful tools for high-performance applications. LH*LH was shown an efficient new data structure for such multicomputers. Performance analysis showed that access times may be in general of the order of a milisecond, reaching 0.4 msec per insert in our experiences, and that the throughput may reach thousands of operations per second, over 2700 in our study, regardless of the file scale-up. An LH*LH file can scale-up over as much of distributed RAM as available, e.g., 2 Gbytes on the Parsytec, without any access performance deterioration. The access times are in particular an order of magnitude faster than one could attain using disk files.

Performance analysis confirmed also various design choices made for LH*LH. In particular, the use of LH for the bucket management, as well as of the concurrent splitting with the dynamic split control and the bulk shipping, effectively reduced the peaks of response time. The improvement reached a thousand times in our experiences, from over $7sec$ that would characterize LH*, to under 7 msec for LH*LH. Without this reduction, LH*LH would likely to be inadequate for many high-performance applications.

Future work should concern a deeper performance analysis of LH*LH under various conditions. More experiments with actual data should be performed. A formal performance model is also needed. Such models yet lack in general for the SDDSs. The task seems of even greater complexity than for more traditional data structures.

The ideas put into the LH*LH design should apply also to other known SDDSs. They should allow for the corresponding variants for switched multicomputers. One benefit would be scalable high performance ordered files. SDDSs in [16], or [9] should be a promising basis towards this goal.

A particularly promising direction should be the integration of LH*LH as a component of a DBMS. One may expect important performance gain, opening to DBMSs new application perspectives. Video servers seem one promising axis, as it is well known that major DBMS manufacturers look already upon switched multicomputers for this purpose. The complex real-time switching data management in telephone networks seems another interesting domain.

To approach these goals, we plan to make use of the implementation of LH*LH for high-performance databases. We will interface it with our research platform AMOS [5], which is an extensible object-relational database management system with a complete query language [7]. AMOS would then reside on an ordinary workstation, whereas some datatypes/relations/functions would be stored and searched by the MIMD machine. AMOS will then act as a front-end system to the parallel stored data. The query optimization of AMOS will have to be extended to also take into account the communication time and possible speed-up gained by using distributed parallel processing. Other SDDSs than LH* are also of interest for evaluation, a new candidate is the RP* [16] that handles ordered data sets.

Acknowledgment

This project was supported by NUTEK (The Swedish National Board for Industrial and Technical Development), and CENIIT (The Center for Industrial Information Technology).

References

[1] Teradata Corporation. DBC/1012 data base computer concepts and facilities. Technical Report Teradata Document C02-001-05, Teradata Corporation, 1988.

[2] D. Culler. NOW: Towards Everyday Supercomputing on a Network of Workstations. Technical report, EECS Tech. Rep. UC Berkeley, 1994.

[3] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proc. of the 4th Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, 1993.

[4] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA: A high performance dataflow database machine. In *Proc of VLDB*, August 1986.

[5] G. Fahl, T. Risch, and M. Sköld. AMOS - An Architecture for Active Mediators. In *IEEE Transactions on Knowledge and Data Engineering*, Haifa, Israel, June 1993.

[6] J. S. Karlsson. LH*LH: Architecture and Implementation. Technical report, IDA, Linkping University, Sweden, 1995.

[7] J. S. Karlsson, S. Larsson, T. Risch, M. Sköld, and M. Werner. *AMOS User's Guide.* CAELAB, IDA, IDA, Dept. of CS and IS, Linköping University, Sweden, memo 94-01 edition, Mars 1994. URL: http://www.ida.liu.se/labs/edslab/amos/amosdoc.html.

[8] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Architecture and performance of relational algebra machine GRACE. In *Proc. of the Intl. Conf. on Parallel Processing*, Chicago, 1984.

[9] B. Kroll and P. Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In *ACM-SIGMOD Int. Conf. On Management of Data*, 1994.

[10] P.Á. Larson. Dynamic hashing. *BIT*, 18(2):184–201, 1978.

[11] P.Á. Larson. Dynamic hash tables. In *Communications of the ACM*, volume 31(4), pages 446–57. April 1988.

[12] W. Litwin. Linear Hashing: A new tool for file and table addressing. Montreal, Canada, 1980. Proc. of VLDB.

[13] W. Litwin. Linear Hashing: A new tool for file and table addressing. In Michael Stonebraker, editor, *Readings in DATABASE SYSTEMS, 2nd edition*, pages 96–107. 1994.

[14] W. Litwin, M-A. Neimat, and D. Schneider. LH*: A Scalable Distributed Data Structure. submitted for journal publication, Nov 1993.

[15] W. Litwin, M-A Neimat, and D. Schneider. LH*: Linear hashing for distributed files. ACM SIGMOD International Conference on Management of Data, May 1993.

[16] W. Litwin, M-A Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. VLDB Conference, 1994.

[17] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Number ISBN 0-13-715681-2. Prentice Hall, 1991.

[18] Parsytec Computer GmbH. *Programmers Guide, Parix 1.2-PowerPC*, 1994.

[19] M. Pettersson. Main-Memory Linear Hashing - Some Enhancements of Larson's Algorithm. Technical Report LiTH-IDA-R-93-04, ISSN-0281-4250, IDA, 1993.

[20] C. Severance, S. Pramanik, and P. Wolberg. Distributed linear hashing and parallel projection in main memory databases. In *Proceedings of the 16th International Conference on VLDB*, Brisbane, Australia, 1990.

[21] Andrew S. Tanenbaum. *Distributed Operating Systems*. 1995.

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