Distributed Paged Hash Tables

José Rufino[1] *, António Pina[2], Albano Alves[1], and José Exposto[1]

ta, citation and similar papers at core.ac.uk

[2] University of Minho, 4710-057 Braga, Portugal
pina@di.uminho.pt

Abstract. In this paper we present the design and implementation of DPH, a storage layer for cluster environments. DPH is a Distributed Data Structure (DDS) based on the distribution of a paged hash table. It combines main memory with file system resources across the cluster in order to implement a distributed *dictionary* that can be used for the storage of very large data sets with key based addressing techniques. The DPH storage layer is supported by a collection of cluster–aware utilities and services. Access to the DPH interface is provided by a user–level API. A preliminary performance evaluation shows promising results.

1 Introduction

Today commodity hardware and message passing standards such as PVM [1] and MPI [2] are making possible to assemble clusters that exploit distributed storage and computing power, allowing for the deployment of data-intensive computer applications at an affordable cost. These applications may deal with massive amounts of data both at the main and secondary memory levels. As such, traditional data structures and algorithms may no longer be able to cope with the new challenges specific to cluster computing.

Several techniques have thus been devised to distribute data among a set of nodes. Traditional data structures have evolved towards Distributed Data Structures (DDSs) [3, 4, 5, 6, 7, 8, 9] . At the file system level, cluster aware file systems [10, 11] already provide resilience to distributed applications. More recently a new research trend has emerged: online data structures for external memory that bypass the virtual memory system and explicitly manage their own I/O [12].

Distributed Paged Hashing (DPH[1]) is a cluster aware storage layer that implements a hash based Distributed Data Structure (DDS). DPH has been designed to support a Scalable Information Retrieval environment (SIRe), an ongoing research project with a primary focus on information retrieval and cataloging techniques suited to the World Wide Web.

* Supported by PRODEP III, through the grant 5.3/N/199.006/00, and SAPIENS, through the grant 41739/CHS/2001.

[1] A preliminary presentation of our work took place at the PADDA2001 workshop [13]; here we present a more in-depth and updated description of DPH.

J.M.L.M. Palma et al. (Eds.): VECPAR 2002, LNCS 2565, pp. 679–693, 2003.
© Springer-Verlag Berlin Heidelberg 2003

The main idea behind DPH is the distribution of a paged hash table over a set of networked *page servers*. Pages are contiguous bucket sets[2], all with the same number of buckets. Because the amount of pages is initially set our strategy appears to be static. However, pages are created on–demand so the hash table grows dynamically.

A *page broker* is responsible for the mapping of pages to *page servers*. The mapping takes place just once for the lifetime of a page (page migration is not yet supported) and so the use of local caches at the service clients alleviates the *page broker*. In a typical scenario, the *page broker* is mainly active during the first requests to the DPH structure when pages are mapped to the available *page servers*. Because the local caches are incrementally updated the *page broker* will be relieved from further mapping requests.

The system doesn't rely only on the available main memory at each node. When performance is not the primary concern, a threshold based swap mechanism may also be used to take advantage of the file system. It is even possible to operate the system solely based on the file system, achieving the maximum level of resilience. The selection of the swap-out bucket victims is based on a Least–Recently–Used (LRU) policy.

The paper is organized as follows: section 2 covers related work, section 3 presents the system architecture, section 4 shows preliminary performance results and section 5 concludes and points directions for future work.

2 Related Work

Hash tables are well known data structures [14] mainly used as a fast key based addressing technique. Hashing has been intensively exploited because retrieval times are $O(1)$ when compared with $O(log\ n)$ for tree-structured schemes or $O(n)$ for sequential schemes. Hashing is classically *static* meaning that, once set, the bit–length of the hash index never changes and so the complete hash table must be initially allocated.

In dynamic environments, with no regular patterns of utilization, the use of static hash tables results on storage space waste if only a small bucket subset is used. Static hashing may not also be able to guarantee $O(1)$ retrieval times when buckets overflow. To counterwork these limitations several dynamic hashing [15] techniques have been proposed, such as Linear Hashing (LH) [16] and Extendible Hashing (EH) [17], along with some variants.

Meanwhile, with the advent of cluster computing, traditional data structures have evolved towards distributed versions. The issues involved aren't trivial because, in a distributed environment, scalability is a primary concern and new problems arise (consistency, timing, order, security, fault tolerance, hot–spots, etc.). In the hashing domain, LH* [3] extended LH [16] techniques for file and table addressing and coined the term *Scalable Distributed Data Structure* (SDDS). Distributed Dynamic Hashing (DDH) [4] offered an alternative

[2] In the DPH context, a bucket is a hash table entry where collisions are allowed and self–contained, that is, collisions don't overflow into other buckets.

approach to LH* while EH* [5] provided a distributed version of EH [17]. Although in a very specific application context, [18] have exploited a very similar concept to DPH, named *two–level hashing*. Distributed versions of several other classical data structures, such as trees [7, 8] and even hybrid structures, such as hash–trees [19], have also been designed. More recently, work has been done on hash based distributed data structures to support Internet services [9].

3 Distributed Paged Hashing

Our proposal shows that for certain classes of problems, an hybrid approach, that mixes static and dynamic techniques, may achieve good performance and scalability without the complexity of purely dynamic schemes.

When the dimension of the key space is unknown *a priori*, a pure dynamic hashing approach would incrementally use more bits from the hash index when buckets overflow and split. Only then storage consumption would expand to make room for the new buckets. Typically, the expansion takes place at another server, as distributed dynamic hashing schemes tend to move one of the splits to another server.

Although providing maximum flexibility, a dynamic approach increases the load on the network, not only during bucket splitting, but also when a server forwards requests from clients with an outdated view of the <bucket, server> mapping. Once we know in advance that the application domain (SIRe) will include a distributed web crawler, designed to extract and manage millions of URLs, then it doesn't make much sense not to start, from the beginning, using the maximum bit–length of the hash index. As such, DPH is a kind of hybrid approach that includes both static and dynamic features: it uses a fixed bit–length hash table, but pages (and buckets) are created on–demand and distributed across the cluster.

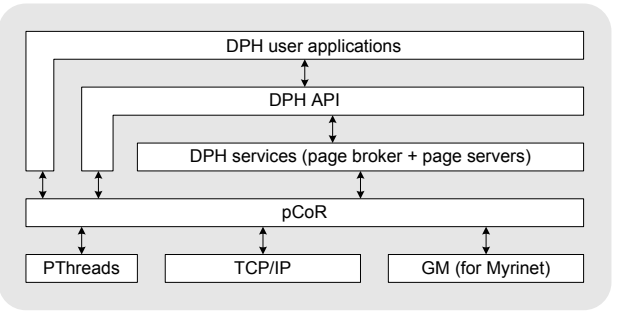


Fig. 1. The DPH architecture

3.1 Architecture

Figure 1 presents the architecture of DPH. User applications interface with the DPH core (the *page broker* and the *page servers*) through a proper API. The runtime system is provided by pCoR [20], a prototype of CoR [21]. CoR paradigm extends the process abstraction to achieve structured fine grained computation using a combination of message passing, shared memory and POSIX Threads. pCoR is both multithreaded and thread safe and already provides some very useful features, namely message passing (by using GM [22] over Myrinet) between threads across the cluster. This is fully exploited by the DPH API and services, which are also multithreaded and thread safe.

3.2 Addressing

The DPH addressing scheme is based on *one–level paging* of the hash table:

1. a static hash function H is used to compute an index i for a key k: $H(k) = i$;
2. the index i may be split into a page field p and an offset field o: $i =< p, o >$;
3. the hash table may be viewed as a set of $2^{\#p}$ pages, with $2^{\#o}$ buckets per page, where $\#p$ and $\#o$ are the (fixed) bit–length of the page and offset fields, respectively;
4. the page table pt will have $2^{\#p}$ entries, such that $pt[j] = ps_j$, where ps_j is a reference to the page server for page j.

H is a 32 bit hash function[3], but smaller bit subsets from the hash index may be used, with the remaining bits being simply discarded. The definition of the page and offset bit–lengths are the main decisions to take prior to the usage of the DPH data structure. The more bits the page field uses, the more pages will be created, leading to a very sparse hash table (if enough page servers are provided), with a small number of buckets per page. Of course, the reverse will happen when the offset field consumes more bits: fewer, larger pages, handled by a small number of page servers. The later scenario will less likely take advantage of the distribution. Thus, defining the index bit–length is a decision dependent on the key domain. We want to minimize collisions and so large indexes may seem reasonable but that should be an option only if we presume that the key space will be uniformly used. Otherwise storage space will be mostly wasted on control data structures.

3.3 Page Broker

The *page broker* is responsible for the mapping of pages into *page servers*. As such, the *page broker* maintains a page table, pt, with $2^{\#p}$ entries, one for each page. When it receives a mapping request for page p, the *page broker* returns

[3] H has been chosen from [23]. A comparison was made with other general hash functions from [24], [14] and [25], but no significant differences have been found, both in terms of performance and collision avoidance.

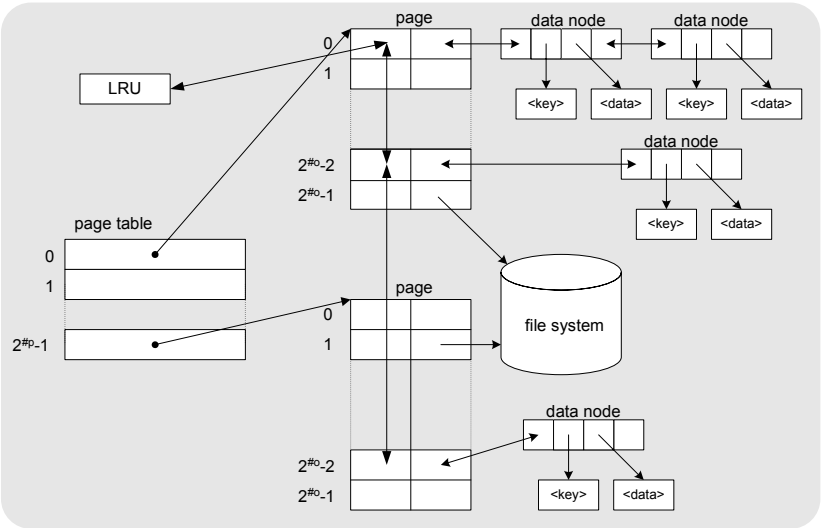


Fig. 2. Main data structures for a *page server*

$pt[p]$, which is a reference to the *page server* responsible for the page p. It may happen, however, that this is the first mapping request for the page p. If so, the *page broker* will have to choose a *page server* to handle that page. A Round Robin (RR) policy is currently used over the available *page servers*, assuring that each handles an equal share of the hash table, but we plan to add the choice for more adaptive policies, such as weighted RR (proportional to the available node memory and/or current load, for instance) or others.

3.4 Page Servers

A *page server* hosts a page subset of the distributed hash table (as requested by the *page broker*, during the mapping process), and answers most of the DPH user level API requests (insertions, searches, deletions, etc.).

Figure 2 presents the main data structures for a *page server*. A *page table* with $2^{\#p}$ entries is used to keep track of the locally managed pages. A page is a bucket set with $2^{\#o}$ entries. A bucket is an entry point to a set of data nodes which are <key, data> pairs. Collisions are self contained in a bucket (*chaining*). Other techniques, like using available empty buckets on other pages (*probing*), wouldn't be compatible with the swapping mechanism[4].

Presently, buckets are doubly–linked lists. These rather inefficient data structures, with $O(n)$ access times, were used just to rapidly develop the prototype. In the future we plan to use more efficient structures, such as trees, skip–lists [26] or even dynamic hashing.

[4] This mechanism uses the bucket as the swap unit and depends on information kept therein to optimize the process.

One of the most valuable features of a *page server* is the ability to use the file system as a complementary online storage resource. Whenever the current user data memory usage surpasses a certain configurable threshold, a swapping mechanism is activated. A bucket victim is chosen, from the buckets currently hold in memory. The victim, chosen from a Least–Recently–Used (LRU) list, is the oldest possible referenced bucket that still frees enough memory to lower the current usage bellow the threshold.

The LRU list links every bucket currently in main memory, crossing local page boundaries, and so a bucket may be elected as a victim in order to release storage to a bucket from another local page. The LRU list may also be exploited in other ways. For instance, besides being a natural queue, quickly browsing every bucket in a *page server* is possible, without the need to hash any key.

Buckets that have been swapped-out to the file system are still viewed as online and will be swapped-in as they are needed. The swapping granularity is currently at the bucket level and not at the data node level. This may be unfair to some data nodes in the bucket but prevents too many small files (often thousands), one for each data node, at the file system level, which would degrade performance. The swapping mechanism is further ooptimizedthrough the use of a *dirty bit* per bucket, preventing unmodified buckets to be unnecessarily saved.

A *page server* may work with a zero threshold thus using the main memory just to keep control data structures and as an intermediate pool to perform the user request, after which the bucket is immediately saved to the file system and the temporary instance removed from main memory.

If a DPH instance has been terminated graciously (thus ssynchronizingits state with the file system), then it may be loaded again, on–demand: whenever a *page server* is asked to perform an operation on a bucket that is empty, it first tries to load a possible instance from the file system because an instance may be there from a previous shutdown of the DPH hash table. In fact, even after an unclean shutdown, partial recovery may be possible because uunsynchronized bucket instances are still loaded.

3.5 User Applications

User applications may be built on top of the DPH API and runtime environment. From a user application perspective, insertions, retrievals and removals are the main interactions with the DPH storage layer. These operations must have a key hashed and then mapped into the correct *page server*. This mapping is primarily done through a local cache of the *page broker* page table. A user application starts with an empty page table cache and so many cache misses will take place, forcing mapping requests to the *page broker*. This is done automatically, in a transparent way to the user application. Further mappings of the same page will benefit from a cache hit and so the user application will readily contact the relevant *page server*.

Presently, mapping never changes for the lifetime of a DPH instance[5] and so the cache will be valid during the execution of the user application. This way, a *page broker* will be a hot–spot (if ever) for a very limited amount of time. Our preliminary tests show no significant impact on performance during cache fills.

3.6 Client–Server Interactions

Our system operates with a relatively small number of exchanged messages[6]:

1. mapping a page into a *page server* may use zero, two, four or (very seldom) more messages: if the local cache gives a hit, zero messages were needed; otherwise the *page broker* must be contacted; if the page table gives a hit, only the reply to the user application is needed, summing up two messages; otherwise a *page server* must be contacted and so two more messages are needed (request and reply); of course, if the *page server* replied with a negative acknowledgement, the Round Robin search for another *page server* will add two more messages per *page server*;
2. insertions, retrievals and removals typically use two messages (provided a cache hit); however, insertions and retrievals may be asynchronous, using only one message (provided, once again, a cache hit); the later means that no acknowledge is requested from the *page server*, which translates into better performance, though the operation may have not be successfully performed and the user application won't be aware of it.

Once local caches become updated, and assuming the vast majority of the requests to be synchronous insertions, retrievals and deletions, we may set two messages as the upper bound for each interaction of a client with a DPH instance.

4 Performance Evaluation

4.1 Test–Bed

The performance evaluation took place in a cluster of five nodes, all running Linux Red Hat 7.2 with the stock kernel (2.4.7-10smp) and GM 1.5.1 [22]. The nodes were interconnected using a 1.28+1.28 Gbits/s Myrinet switch. Four of the nodes (A,B,C,D) have the following hardware specifics: two Pentium III processors at 733 Mhz, 512 Mb SDRAM/100 MHz, i840 chipset, 9Gb UDMA 66 hard disks, Myrinet SAN LANai 9 network adapter. The fifth node (E) has four Pentium III Xeon processors running at 700 Mhz, 1 Gb ECC SDRAM/100 MHz, ServerWorks HE chipset, 36 Gb Ultra SCSI 160 hard disk and a Myrinet SAN LANai 9 network adapter.

[5] We are referring to a live instance, on top of a DPH runtime system.
[6] We have restricted the following analysis to the most relevant interactions.

4.2 Hash Bit–Length

Because DPH uses static hashing, the hash bit–length must be preset. This should be done in such a way that overhead from control data structures and collisions are both minimized. However, those are conflicting requisites. For instance, to minimize collisions we should increase the bit–length, thus increasing the hash table height; in turn, a larger hash table will have more empty buckets and will consume more control data structures. We thus need a metric for the choice of the right hash bit–length.

Metric Definition Let B_j be the number of buckets with j data nodes, after the hash table has been built. If k keys have been inserted, then $P_j = (B_j \times j)/k$ is the probability of any given key to have been inserted in a B_j bucket. Also, let N_j be the average number of nodes visited to find a key in a B_j bucket. Once we have used linked lists to handle collisions, $N_j = (j+1)/2$. Then, given an arbitrary key, the average number of nodes to be searched for the key is $N = \sum_j (P_j \times N_j)$. The overhead from control data structures is $O = C/(U+C)$, where C is the storage consumed in control data structures and U is the storage consumed in user data (keys and other possible attached data). Finally, our metric is defined by the ranking $R = nN \times oO$, where n and o are the percentual weights given to N and O, respectively. For a specific scenario, the hash bit–length to choose will be the one that minimizes R.

Application Scenario The tests were performed, in a single cluster node (A), for a varying number of keys, using hash bit–lengths from 15 to 20. The page field of our addressing scheme used half of the hash; the other half was used as an offset in the page (for odd bit–lengths, the page field was favored). Keys were random unique sequences, 128 bytes wide; user data measured 256 bytes[7].

Figure 3 presents the rankings obtained. If an ideal general hash function (one that uniformly spspreads the hashes across the hash space, regardless of the randomness and nature of the keys) was used, we would expect the optimum hash bit–length to be approximately $log_2 k$, for each number of keys k. However, not only our general hash function [23] isn't ideal, but also the overhead factor must be taken into account. We thus observe that our metric is minimized when the bit–length is $log_2 k - 1$, regardless of k[8].

In order to determine if the variation of the key size would interfere with the optimum hash bit–length we ran another test, this time by varying the key size across {4, 128, 256}. Figure 4 shows the results for 125000 keys. It may be observed that $log_2 k - 1$ still is the recommended hash bit–length, independently of the key size[9]. The ranking is preserved because regardless of the key size, the hash function provides similar distributions of the keys; therefore, N is approximately the same, while the overhead O is the varying factor.

[7] Typical sizes used in the web crawler being developed under the SIRe project.

[8] For instance, 17 bits for the hash bit–length seems reasonable when dealing with a maximum of 125000 keys, but our metric gives 16 bits as the recommended value.

[9] This was also observed with 250000, 500000 and 1000000 keys.

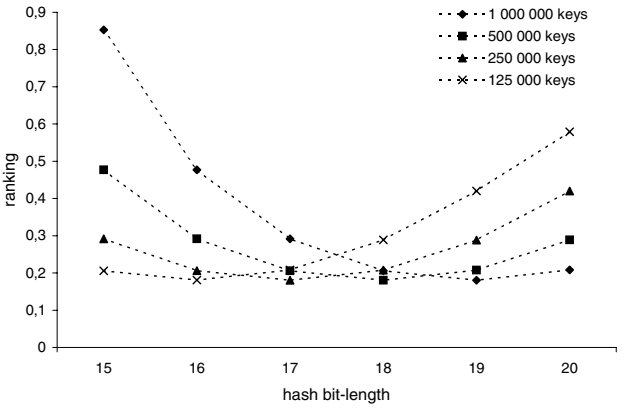


Fig. 3. R for $n = 50\%$ and $o = 50\%$

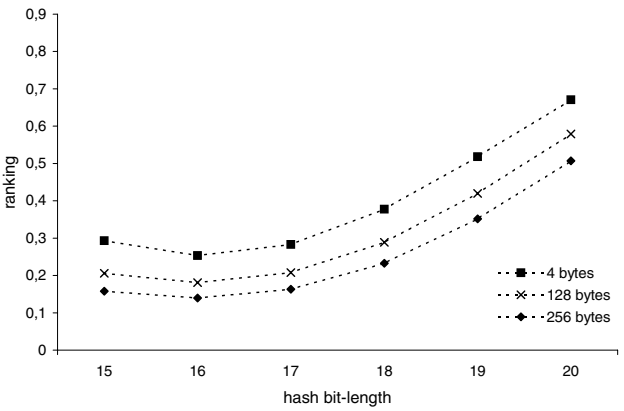


Fig. 4. Effect of the key size on R

4.3 Scalability

To evaluate the scalability of our system we produced another type of experiments using k=1500000 as the maximum number of keys. Accordingly with the metric R defined in the previous experiment, the hash bit–length was set to $log_2 k - 1 = 19$ bits. Also, as previously, keys were random unique sequences, 128 bytes wide, with 256 bytes of attached user data. Each client thread was responsible for the handling of 125000 keys.

We measured insertions and retrievals. Insertions were done in newly created DPH instances and thus the measured times ("build times") accounted for cache misses and *page broker* mappings. The retrieval times and retrieval key rates are not presented, because they were observed to be only marginally better. The memory threshold was set high enough to prevent any DPH swapping.

One Page Server, Multiple Clients The first test was made to investigate how far the system would scale by having a single *page server* to attend simultaneous requests from several multithreaded clients. Our cluster is relatively small and so, to minimize the influence of hardware differences between nodes, we used the following configuration: nodes A,B and C hosted clients, node D hosted the *page server* and node E hosted the *page broker*.

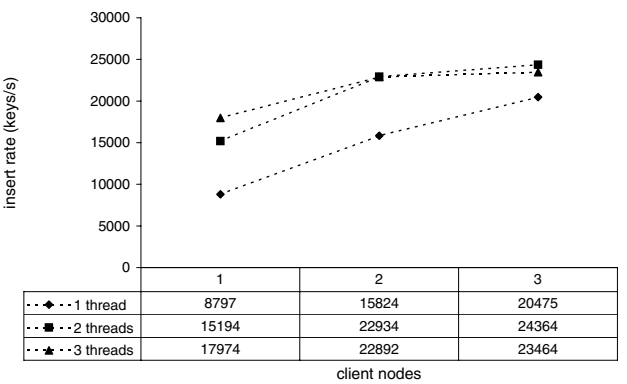
Figure 5 shows the throughput obtained when 1, 2 or 3 clients make simultaneous key insertions by using, successively, 1, 2 or 3 threads: 1 active client, with 1 thread, will insert 125000 keys; ...; 3 active clients, with 3 threads each, will insert $3 \times 3 \times 125000 = 1125000$ keys.

It may be observed that, as expected, we need to add more working nodes to increment the throughput, when using 1 thread per client. Of course, this trend will stop as soon as the communication medium or the *page server* get saturated.

With 2 threads per client, the keyrate still increases; in fact, with just 1 client and 2 threads the throughput achieved is the same as with 2 clients with 1 thread each but, when 3 simultaneous clients are active (in a total of 6 client threads), the speedup from 2 clients is minimum, thus indicating that the saturation point may be near.

When using 3 threads per client and just 1 active client, the speedup from 2 threads is still positive but, when increasing the number of active clients, no advantage is taken from the use of 3 threads. With 2 active clients, 6 threads are used, which equals the number of working threads when 3 clients are active, with 2 threads each; as we already have seen, this later scenario produces very poor speedup; nevertheless it still produces better results than 2 clients with 3 threads (the more threads per client, the more time will be consumed in thread scheduling and I/O contention).

The values presented allow us to conclude that 6 working threads are pushing the system to the limit, but they are unclear about the origin of that behavior: the communication medium or the *page server*?



	1	2	3
1 thread	8797	15824	20475
2 threads	15194	22934	24364
3 threads	17974	22892	23464

Fig. 5. Insert keyrate with one page server and multiple clients

Two Page Servers, Multiple Clients To answer the last question we added one more *page server* to the crew and repeated the tests. But, with just four nodes (the fifth hosted the *page broker* solely), we couldn't perform tests with more than 2 clients. Still, with a maximum of 3 threads per client, we were able to obtain results using a total of 6 threads.

Figure 6 sums up the test results by showing the improving on the insert rate when using one more *page server*. For 1 active client the gains are relatively modest. For 2 active clients the speedup is much more evident, specially when 3 threads per client are used, summing up 6 threads on overall.

The results presented allow us to conclude that by adding *page servers* to our system important performance gains may be obtained. However it remains to be done a quantitative study of the performance scaling in a cluster environment with much more nodes to assign both to clients and *page servers*.

Multiple <Page Server, Client> Pairs So far, we have decoupled clients and *page servers* on every scenario we have tested. It may happen, however, that both must share the same cluster node (as is the case for our small cluster). Thus, it is convenient to evaluate how the system scales in such circumstances.

As previously, the *page broker* was always kept at the node E and measurements were made with a different number of working threads in the client (1, 2 and 3). We started with a single node, hosting a client and a *page server*. We then increased the number of nodes, always pairing a client and a *page server*. The last scenario had four of these pairs, one per node, summing up to 12 active threads and accounting for a maximum of $12 \times 125000 = 1500000$ keys inserted.

Figure 7 shows the insert key rate. The 1–node scenario shows very low key rates with 2 and 3 threads. This is due to high I/O contention between the client threads and the *page server* threads. When the number of nodes is augmented, the key space, although larger, is also more scattered across the nodes, which

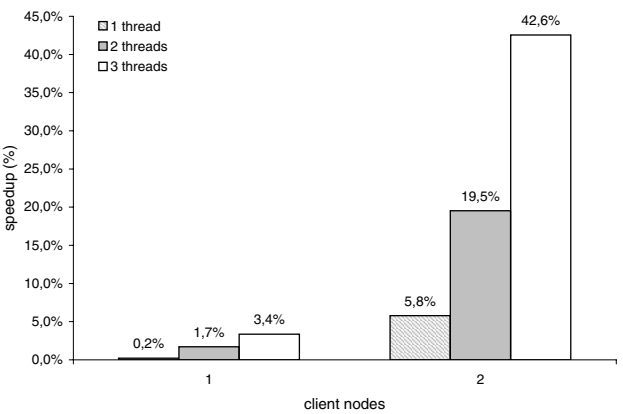


Fig. 6. Speedup with two *page servers*

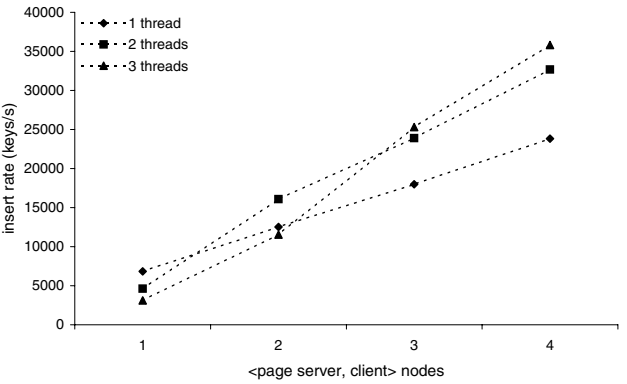


Fig. 7. Insert keyrate with multiple <page server, client> pairs

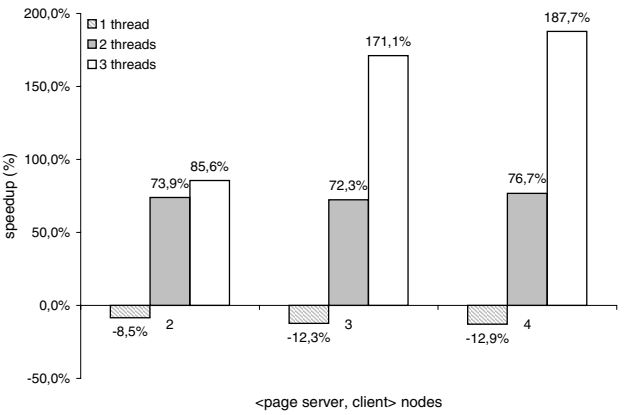


Fig. 8. Insert speedup with multiple <page server, client> pairs

alleviates the contention on each node and makes the use of more threads much more profitable.

Figure 8 shows the speedup with multiple nodes. The speedup refers to the increasing of the measured rates over the rates that could be predicted by linear extrapolation from the 1–node scenario.

5 Conclusions

DPH is a Distributed Data Structure (DDS) based on a simple yet very effective principle: the paging of a hash table and the mapping of the pages among a set of networked *page servers*.

Conceptually, DPH uses *static hashing*, because the hash index bit–length is set in advance. Also, the usage of a *page table* to preserve mappings between

sections (pages) of the table and their locations (*page servers*) makes DPH a *directory* based [15] approach.

However, the hash table is not created at once, because it is virtually paged and pages are dynamically created, on–demand, being scattered across the cluster, thus achieving data balancing. Local caches at user applications prevent the *page broker* to become a *hot–spot* and provide some immunity to *page broker* failures (once established, mappings do not change and so the *page broker* can almost be dismissed).

Another important feature available in the DPH DDS is the capability to exploit the file system as a complementary on–line storage area, which is made possible through the use of a LRU/threshold based swapping mechanism. In this regard, DPH is very flexible in the way it consumes available storage resources, whether they are memory or disk based.

Finally, the performance evaluation we have presented shows that it is possible to define practical metrics to set the hash bit–length and that our selected hash function [23] preserves the (relative) rankings regardless of the key size. We have also investigated the scalability of our system and although we have observed promising results, further investigation is needed with many more nodes.

Much of the research work on hash based DDSs has been focused on dynamic hashing schemes. With this work we wanted to show that the increasing performance and storage capacity of modern clusters may also be exploited with great benefits using an hybrid approach.

In the future we plan to pursue our work in several directions: elimination of the *page broker* by using *directoryless* schemes, inspired by hash routing techniques, such as *consistent hashing* [27]; usage of efficient data structures to handle collisions and near zero–memory–copy techniques to improve performance; exploitation of cluster aware file systems (delayed due to the lack of choice on quality open–source implementations) and external memory techniques [12].

References

[1] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. MIT Press, 1994. 679

[2] M. Snir, S. Otto, S. Huss-Lederman, David Walker, and J. Dongarra. *MPI - The Complete Reference.* Scientific and Engineering Computation. MIT Press, 1998. 679

[3] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH*: Linear Hashing for Distributed Files. In *Proceedings of the ACM SIGMOD - International Conference on Management of Data*, pages 327–336, 1993. 679, 680

[4] R. Devine. Design and implementation of DDH: a distributed dynamic hashing algorithm. In *Proceedings of the 4th Int. Conf. on Foundations of Data Organization and Algorithms*, pages 101–114, 1993. 679, 680

[5] V. Hilford, F. B. Bastani, and B. Cukic. EH* – Extendible Hashing in a Distributed Environment. In *Proceedings of the COMPSAC '97 - 21st International Computer Software and Applications Conference*, 1997. 679, 681

[6] R. Vingralek, Y. Breitbart, and G. Weikum. Distributed File Organization with Scalable Cost/Performance. In *Proceedings of the ACM SIGMOD - International Conference on Management of Data*, 1994. 679
[7] B. Kroll and P. Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In *Proceedings of the ACM SIGMOD – International Conference on Management of Data*, pages 265–276, 1994. 679, 681
[8] T. Johnson and A. Colbrook. A Distributed, Replicated, Data–Balanced Search Structure. Technical Report TR03-028, Dept. of CISE, University of Florida, 1995. 679, 681
[9] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000. 679, 681
[10] W. K. Preslan et all. A 64–bit, Shared Disk File System for Linux. In *Proceedings of the 7h NASA Goddard Conference on Mass Storage Systems and Tech. in cooperation with the Sixteenth IEEE Symposium on Mass Storage Systems*, 1999. 679
[11] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000. 679
[12] J. S. Vitter. Online Data Structures in External Memory. In *Proceedings of the 26th Annual Intern. Colloquium on Automata, Languages, and Programming*, 1999. 679, 691
[13] J. Rufino, A. Pina, A. Alves, and J. Exposto. Distributed Hash Tables. International Workshop on Performance-oriented Application Development for Distributed Architectures (PADDA 2001), 2001. 679
[14] D. E. Knuth. *The Art of Computer Programming – Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. 680, 682
[15] R. J. Enbody and H. C. Du. Dynamic Hashing Schemes. *ACM Computing Surveys*, (20):85–113, 1988. 680, 691
[16] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Databases*, pages 212–223, 1980. 680
[17] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems*, (315-344), 1979. 680, 681
[18] T. Stornetta and F. Brewer. Implementation of an Efficient Parallel BDD Package. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, 1996. 681
[19] P. Bagwell. Ideal Hash Trees. Technical report, Computer Science Department, Ecole Polytechnique Federale de Lausanne, 2000. 681
[20] A. Pina, V. Oliveira, C. Moreira, and A. Alves. pCoR - a Prototype for Resource Oriented Computing. (to appear in HPC 2002), 2002. 682
[21] A. Pina. *MC^2 - Modelo de Computação Celular. Origem e Evolução*. PhD thesis, Dep. de Informática, Univ. do Minho, Braga, Portugal, 1997. 682
[22] Myricom. *The GM Message Passing System*, 2000. 682, 685
[23] B. Jenkins. A Hash Function for Hash Table Lookup. *Dr. Doob's*, 1997. 682, 686, 691
[24] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison–Wesley, 1985. 682
[25] R. C. Uzgalis. General Hash Functions. Technical Report TR 91-01, University of Hong Kong, 1991. 682
[26] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990. 683

[27] D. Kargeer, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *Proceedings of the 8th International WWW Conference*, 1999. 691