PERSISTENT DATABASES THAT PERFORM?

F. A. HENSKENS & M. G. ASHTON

Information Systems Group
Department of Management
University of Newcastle
N.S.W. 2308
email: mgfah@alinga.newcastle.edu.au, mashton@ozemail.com.au

KEYWORDS:

Persistence, database, object stores, Grasshopper, Congeries

1. INTRODUCTION

Conventional computer systems implement a dichotomy of storage mechanisms: a *file store* which provides a durable repository allowing data to exist after the creating program has ceased execution (such data is termed long-term data), and *virtual memory* which provides a repository for data during program execution (such data is termed short-term data). In typical conventional systems it is the responsibility of the programmer to manage the conversion of data between the transient form suitable for virtual memory and the permanent form suitable for the file store. This conversion uses up CPU cycles, and can result in data misinterpretation leading to reduced data protection.

Persistent systems, on the other hand, remove the distinction between long and short-term data by providing a single set of mechanisms for the management of data regardless of its lifetime. Work on persistence to date has largely concentrated on implementation of stores which adhere to the properties of orthogonal persistence defined in [1]. A much smaller body of work has investigated applications which exploit the benefits of persistence. Persistent stores have long been touted as alternatives to conventional database systems, but to our knowledge no-one, with the possible exception of IBM with the AS/400 [2], has actually implemented a database-style application interface to a persistent store. Such an interface would rely on the underlying persistent store providing a durable repository for data rather than having to provide this durability itself, as occurs with conventional database systems.

On the surface it would seem that a persistent database system could be implemented by provision of appropriate software which provides the required interface to the persistent store. This approach ignores the performance issues which have long been recognised by the database community [3]. In the quest for performance, conventional database systems provide a durable store by building on or duplicating some of the features of the underlying operating system. For example conventional database systems typically implement their own buffer management.

In this paper we present the results of research which verifies the performance of the Grasshopper persistent operating system compared to Unix executing on the same hardware, and then presents techniques which draw on the large body of database experience to improve the efficiency of database operations when utilising features of an underlying persistent store.

2. PERSISTENCE

Approaches to implementation of persistent stores have been varied, ranging from purpose-built hardware and operating systems (for example Monads [4]), through persistent operating systems executing above conventional hardware (for example Choices [5], Clouds [6] and Grasshopper [7]) to layered architectures above conventional hardware and operating systems (for example Napier [8] and Casper [9]). Others have built persistent classes into programming languages (for example E [10] and Texas [11]), but such implementations do not fully satisfy the requirements of orthogonal persistence.

The Grasshopper persistent operating system [7] provides explicit support for orthogonal persistence utilising conventional hardware. It provides three important abstractions:

- Containers: Containers provide an abstraction for both storage and access to data objects.
- Loci: The locus provides an abstraction for execution.
- Capabilities: Capabilities provide an abstraction for naming and protection.

These abstractions have been described elsewhere [7] and provide useful support for database management. In particular:

- Container - a single abstraction for both storage of and access to data. Containers may be larger than the virtual address space supported by the underlying hardware. The current implementation supports containers up to 2^{64} bytes in size. Mechanisms, such as container mapping, provide the ability to access even larger data sets.
- Associated with each container is a manager which is responsible for rendering the container's data available to application programs. These managers are situated at user level, allowing container management to be tailored to the kind of data stored in the container and the patterns of access to this data.
- Data is durable and may outlive the program which created it. Persistence is a property of the operating system and is entirely transparent to the user. Consequently programmers do not have to think in terms of both storage and access models.
- Support for the notion of stability. Grasshopper supports the recovery of the system to a consistent state in the event of a crash. This is potentially useful in the development of support for database transactions.

Published in: Proceedings, IASTED International Conference on Software Engineering SE '97, IASTED/ACTA Press, ISBN 0-88986-244-3, San Francisco, U.S.A., pp. 168-172, November 1997.

- Capabilities - capabilities control access to and manipulation of objects. Database data is typically shared between multiple applications acting on behalf of multiple concurrent users. Such access must be strictly controlled from both the security and concurrency view-points. Capabilities achieve this using a single mechanism which provides security and access control in contrast with two mechanisms used by conventional systems.

Persistent systems have long been criticised for poor performance, and since our research is aimed at providing database systems which exhibit similar performance to conventional database systems, we thought it important to perform comparative benchmark testing of the underlying operating system. We used a simple relational database implementation [12] as the basis for a close approximation of the OO1 [13] database benchmark. Using this we compared the performance of the benchmark operations using an underlying Unix file store with its performance using the store provided by the Grasshopper persistent operating system.

3. PERFORMANCE EVALUATION OF GRASSHOPPER

The OO1 database benchmark exercises a database over a number of operations such as random lookup, directed search and insertion of new records. It was chosen for this experiment because it was relatively straightforward to implement in a wide range of systems, yet provided a useful set of operations on the database.

The OO1 database is defined as two logical records:

Part: RECORD[id: INT, type: STRING[10], x, y: INT, build: DATE]
Connection: RECORD[from: Part-id, to: Part-id, type: STRING[10], length: INT]

The small database has 20,000 parts with unique IDs 1 through 20,000. There are 60,000 connections with exactly three connections from each part to other randomly selected parts. The x, y and length fields each contain values randomly distributed in the range [0..99999], the type fields have values randomly selected from the strings {"part-type0" . . . "part-type9"}, and the build date is randomly distributed over a 10-year range. The random connections are generated so that 90% of the connections are randomly selected from the 1% of parts that are closest to the original part, with the remainder selected from any random part.

The following operations are carried out on the database:

Lookup	Generate 1,000 random part IDs and fetch the corresponding part from the database. For each part, call a null procedure written in a host programming language, passing the x, y position and type of the part.
Traversal	Find all the parts connected to a randomly selected part, or to a part connected to it, and so on, up to seven hops (total of 3280 parts, with possible duplicates). For each part, call a null programming language procedure with the value of the a and y fields and the part type. Also measure time for reverse traversal, swapping "from" and "to" directions to compare the results obtained.
Insert	Enter 100 parts and three connections from each to other randomly selected parts. Time must be included to update indices, or other access structures used in the execution of *lookup* and *traversal*. Call a null programming language procedure to obtain the x, y position for each insert. Commit the changes to disk.

The OO1 benchmark specifies two sizes of database; small as described above with 20,000 parts and 60,000 connections, and large, scaled by a factor of 10 with 200,000 parts and 600,000 records. A third size, huge, with 2,000,000 parts and 6,000,000 connections is specified, but not used for this experiment.

The database was set up as a simple physical storage model based on B+-tree indexes and implemented at the application level using the C language. It consisted of four levels:

- The physical storage level. The tables and their indexes were each implemented as separate files.
- The file and index management level. Access and management of the files were supported through a data file management module and a B-tree management module.
- The database management level. The database management module abstracts over details of file and index management.
- The application level. Operations on the database such as database creation, lookup, traversal, and insertion were implemented at the application level.

The datafile, B-tree and database management software was based on a database management project written by Stephens [12] for a MS-DOS system, and adapted by us for the Unix and Grasshopper environments.

Three implementations were developed making only those changes which were absolutely necessary for the implementation to run in its particular environment, thus reducing the effect of coding differences on the comparative results. Changes were required to the open, close, read, write, and seek operations, but algorithms were not changed and no effort was made to take advantage of features of any environment. All testing was performed on a 133 MHz DEC Alpha with 64 Megabytes of memory using OSF Unix or Grasshopper as appropriate. Tests performed were:

- A file-based model where each table and index was implemented as a separate file. This was run using OSF Unix.
- A memory-based model where a file was modelled in persistent memory. All the operations on the database remained identical to those used in other tests. This was run using the Grasshopper system.
- A memory-based model mapped onto memory-mapped files. This was run using OSF Unix.

The application layer, implementing the OO1 database was developed in C. The specification of the database was written in the schema language provided with Cdata project and then converted to a C program and header files.

The OO1 database for this experiment consisted of the following:

PART	The file of parts as specified in the original specification, together with a B-tree index on the primary key *id*.
CONNECT	The file of connections as specified in the original specification together with three indexes. A B-tree index on the primary key (the concatenation of the *from* and *to* fields), together with separate indexes on the *from* and *to* fields.

The results of these experiments are summarised in Table 1.

The results for the 20,000 part database presented in Table 1 indicate that the Grasshopper and memory-mapped OSF implementations perform the test an order of magnitude faster than the file-based OSF version. This is understandable because the Grasshopper and memory-mapped systems do not depend on the file system calls with their associated overheads. Results for the 200,000 part database were 33% higher for OSF files, and 18% higher for the memory mapped and Grasshopper systems. Grasshopper could not be tested for the 2,000,000 part database because the state of system development did not allow such large data sets. While the results for Grasshopper and the memory-mapped file systems were consistently similar, it should be noted that memory mapped files do not provide any resilience and would be unsuitable for implementing real-world database systems.

These performance figures were achieved using the persistent store as a *replacement* for a file system with no attempt to exploit features uniquely provided by the store. In particular no attempt was made to optimise the positioning of data to suit the access patterns used in the benchmark tests.

4. OBJECT PLACEMENT AND PERFORMANCE

Contemporary computers, when used for data intensive applications, provide insufficient primary (virtual or computational) memory to contain the entire data set. This limitation, together with the requirement for durable storage, results in the need for constant movement of data between primary and secondary memory. Secondary memory is generally four orders of magnitude slower than primary memory. Thus program performance is adversely affected by operations which require movement of data between memory levels. Clearly it is desirable to maximise the presence of soon-to-be-accessed data in primary memory, while simultaneously minimising data transfer operations. This is achieved at the file-system level in conventional database systems by a combination of judicious data placement and appropriate buffer management [3], and at the virtual memory level by advanced memory management protocols [14, 15]. Such placement clusters data according to, for instance, its existence within relational tables or expected access patterns defined by the database administrator.

In conventional database systems, placement of newly created data is achieved when the data is moved from (temporary) computational memory to (durable) database memory. Until now such user control over placement has not been available for orthogonally persistent stores - in these systems data is created, its placement is transparent to application programs, and its durability is achieved through its being referenced by other persistent data.

Observation of the data objects created in our earlier experiments revealed that:

- Object placement was according to order of creation.
- Multiple objects typically exist within the disk I/O transfer block.
- Subsequent object accesses were often not related to order of object creation. The neighbourhood established during object creation was thus not necessarily the neighbourhood required for minimisation of disk I/O during subsequent access to the objects.

Clearly control over the physical location of data objects is necessary to achieve the performance benefits derived from data clustering. In the persistence context such control requires the ability to specify placement at the time of object creation.

Commonly used programming languages provide instructions (eg *malloc*() in C and *new* in C++) for object creation. These allocate object storage sequentially from a single heap, and thus do not provide the level of support required for optimal placement. (Multiple heaps are supported by some commercial libraries [16].)

5. OBJECT PLACEMENT AND *CONGERIES*

In a typical database environment, the database designer must set parameters which affect the performance of the system. These include buffer properties, file locations (possibly over different physical disks), clustering of data into and within files, use of indexes, and so on. In performing this task the designer must take into account issues such as expected data access patterns, size of data, frequency of use, size of memory, number of disk drives available and so on. Many of these are equally relevant to databases implemented in persistent systems though, of course, persistent systems do not have a direct equivalent to files.

To achieve optimal placement of objects within the persistent store we need the ability to:

- Allocate new blocks of storage with the same size and alignment as those used for disk I/O.
- Organise these blocks to form storage regions which we call *congeries* (a congeries is a gathered mass).
- Nominate the congeries in which a new object is created.

Operation	OSF: Files	OSF: Memory Mapped	Grasshopper
Lookup	1.45	0.11	0.11
Traversal	7.68	0.47	0.47
Insert	1.70	0.17	0.18
Total	10.01	0.75	0.76

TABLE 1: RESULTS OF THE OO1 BENCHMARK - 20,000 PART DATABASE (TIME IN SECONDS)

Congeries perform a management function for persistent databases similar to that of the file in conventional databases. At the time of persistent database design, elements of the schema are allocated to congeries. This is analogous to the mapping of, for example, tables onto files in conventional databases.

We note that there is no requirement that congeries comprise contiguous disk or virtual memory blocks. Contiguity of the storage provided by congeries is achieved by appropriate management rather than by the physical location of its component blocks (in the same way as non-contiguous physical memory pages may be used to implement contiguous virtual memory).

The addition of an extra parameter to the call used to create objects allows identification of the congeries to which the object belongs. Congeries management requires facilities to:

- Expand the congeries size by adding new blocks.
- Add a new object to the congeries.
- Delete an object from the congeries.
- Garbage collect the congeries.
- Release blocks from the congeries.

In practice congeries may be implemented using library functions, even where there is no support for multiple heaps. The allocation of a newly created object to a congeries may either be automatic (based, for instance, on database designer specification combined with the relational table to which the object belongs), or through the provision of dedicated programming language instructions.

6. CONGERIES AND GRASSHOPPER

The above abstractions provide the fundamentals on which to build more fine-grained control over object management, in particular in the context of this paper, object placement.

Currently, because of the state of development of the Grasshopper Operating System, we are forced to implement database systems within a single container. Of necessity, then, congeries form subsets of container blocks. Congeries management is implemented as a set of library functions which extend the program code used to implement the database interface.

As the functionality of the Grasshopper system evolves to reliably support multiple containers and their managers at user level, we intend to:

- Integrate congeries management with container management, thus creating a generic class of container designed to host congeries.
- Implement and benchmark database schema which initially place each congeries into a discrete container.
- Investigate the impact of various combinations of congeries and containers on database performance.

It is expected that congeries will provide advantages in other areas of data management (apart from performance), similarly to those provided by large objects in a system which supports large and small grained objects as described in [17].

7. CONCLUSION

In this paper we present results which demonstrate the viability, from the program performance point-of-view, of the Grasshopper persistent store as an alternative to conventional operating systems such as Unix. Grasshopper is thus suitable as a basis for the implementation of conventional-style database systems which can exploit features of persistent stores not present in conventional programming environments.

As previously recognised by the developers of conventional database systems, the placement of data objects has a significant impact on database performance. To date such control has not been available to programmers working in persistent environments. User-level control over placement of persistent objects may be provided through the use of a new abstraction called a *congeries.* Congeries provide support for object placement, with consequent improvements in I/O performance for persistent databases. The Grasshopper persistent operating system is being used as a test-bed for evaluation of this new abstraction, and experimental results will be the subject of future reports.

REFERENCES

1. Atkinson, M. and Morrison, R. "Persistent System Architectures", *Proceedings of the Third International Workshop on Persistent Object Systems*, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 73-97, 1989.
2. Soltis, F. "Inside the AS/400", Duke Press, Loveland, Colorado, 1995.
3. Stonebraker, M. "Operating System Support for Database Management", *Communications of the ACM*, 24(7), ACM, pp. 412-418, 1981.
4. Keedy, J. L. and Rosenberg, J. "Support for Objects in the MONADS Architecture", *Proceedings of the International Workshop on Persistent Object Systems*, ed J. Rosenberg and D. M. Koch, Springer-Verlag, 1989.

5. Campbell, R. H., Johnston, G. M. and Russo, V. F. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)", *ACM Operating Systems Review*, 21(3), pp. 9-17, 1987.
6. Dasgupta, P., Chen, R. C., Menon, S., Pearson, M., Ananthanarayanan, R., Ramachandran, U., Ahamad, M., LeBlanc Jr., R., Applebe, W., Bernabeu-Auban, J. M., Hutto, P. W., Khalidi, M. Y. A. and Wileknloh, C. J. "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems Journal*, vol 3, 1990.
7. Dearle, A., di Bona, R., Farrow, J. M., Henskens, F. A., Lindström, A., Rosenberg, J. and Vaughan, F. "Grasshopper: An Orthogonally Persistent Operating System", *Computing Systems Journal*, 1994.
8. Brown, A. L., Dearle, A., Morrison, R., Munro, D. and Rosenberg, J. "A Layered Persistent Architecture for Napier88", *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 155-172, 1990.
9. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings, The Fourth International Workshop on Persistent Object Systems*, pp. 99-109, 1990.
10. Richardson, J. E. and Carey, M. J. "Implementing Persistence in E", *Proceedings of the Third International Workshop on Persistent Object Systems*, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 175-199, 1989.
11. Singhal, V., Kakkad, S. and Wilson, P. "Texas: An Efficient, Portable Persistent Store", *Persistent Object Systems, Proceedings of the 5th International Workshop on Persistent Object Systems*, San Miniato, Italy, pp. 11-33, 1992.
12. Stephens, A. "C Database Development", MIS Press, New York, 1991.
13. Cattell, R. G. G. and Skeen, J. "Object Operations Benchmark", *ACM Transactions on Database Systems*, 17(1), ACM, pp. 1-31, 1992.
14. Pang, H., Carey, M. and Livney, M. "Managing Memory for Real-Time Queries", *SIGMOD*, 5/94, ACM, Minneapolis, Minnesota, USA, pp. 221-232, 1994.
15. Ulusoy, O. and Buchmann, A. "Exploiting Main Memory DBMS Features to Improve Real-Time Concurrency Control Protocols", *SIGMOD*, 25(1), ACM, pp. 23-25, 1996.
16. Applegate, A. D. "Rethinking Memory Management", *Dr Dobb's Journal*, 19(6), 1994.
17. Henskens, F. A., Brössler, P., Keedy, J. L. and Rosenberg, J. "Coarse and Fine Grain Objects in a Distributed Persistent Store", *Proceedings, Third International Workshop on Object Orientation in Operating Systems*, IEEE, Ashville, North Carolina, pp. 116-123, 1993.