Congeries, Mapping and Grasshopper

M. G. Ashton & F. A. Henskens

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

1. INTRODUCTION

Conventional computer systems implement a dichotomy of storage mechanisms: a *file store* providing a durable repository allowing data to exist after the creating program has ceased execution (such data is termed long-term data), and *virtual memory* providing 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 is particularly true for modern applications where computational structures have significant semantic content, and the rebuilding of such structures is not only time-consuming, they may be impossible to reproduce. In addition the 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 adhering to the properties of orthogonal persistence defined in [2]. A much smaller body of work has investigated applications that 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 [15], 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 providing the required interface to the persistent store. This approach ignores the performance issues that have long been recognised by the database community [17]. 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 describe the current state and future direction of research which has already verified the performance of the Grasshopper persistent operating system compared to Unix executing on the same hardware. This research is now drawing on techniques described in the large body of database literature 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, and range from purpose-built hardware and operating systems (for example Monads [10]), through persistent operating systems executing above conventional hardware (for example Choices [4], Clouds [6] and Grasshopper [7]) to layered architectures above conventional hardware and operating systems (for example Napier [3] and

Published in: Proceedings of the Fifth IDEA (Integrated Data Environments Australia) International Workshop, ed A L Brown, Fremantle, Western Australia, February 1998.

Casper [11]). Others have built persistent classes into programming languages (for example E [13] and Texas [14]), 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:

- Containers provide 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
- Parts of containers may be mapped into other containers. This mechanism provides a powerful technique for building a database as a single address space out of a number of separately managed components. Grasshopper provides for recursive mapping with the only restriction being that circular mapping is not allowed. Separate management allows each mapped region to have its own paging policy, and security restrictions.
- Associated with each container is a manager 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 persistent and may outlive the program that 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.
- 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 providing both security and access control, in contrast with the 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 (or even superior) 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 [16] as the basis for a close approximation of the OO1 [5] 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.

The results of these experiments were reported in [8]. While the results for Grasshopper and the memory-mapped Unix file systems were consistently similar, it should be noted that the files as they are

currently implemented, do not provide any resilience and would be unsuitable in their present state 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.

3. 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 [17], and at the virtual memory level by advanced memory management protocols [12, 18]. 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. The authors recognise that this becomes quite complex when the data set is active, and data is being constantly added to and removed from the data set.

Commonly used programming languages provide instructions (eg *malloc*() in C and *new* in C++) for object creation. These typically allocate object storage from a single heap based on a policy that is generally suitable for the placement of temporary, rather than persistent data. Such placement policies do not take into account the need to manage data beyond the invocation of the current program (multiple heaps are supported by some commercial libraries [1]).

4. OBJECT PLACEMENT AND *CONGERIES*

In a typical database environment, the database designer must set parameters affecting 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.

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.

5. 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.

The container abstraction provided by Grasshopper appears to be well suited as a basis for implementation of congeries. Of particular interest is container mapping, which allows a contiguous region (including all) of a (mapped) container to be mapped into another (host) container. The mapped region(s) thus becoming part of the addressing environment provided by the host container. Mapping is recursive or transitive with the only restriction being that circular mapping is not allowed. Mapping thus provides the possibility of separate management for groups of data objects within a host container while simultaneously providing a single addressing environment in which application modules execute.

Mapping itself is not a new idea, though its has not yet been used in commercial database systems. One of the problems with mapping to date has been the expense of extensibility, in that if a mapped area became full it had to be unmapped and remapped to a larger area (in much the same way as early fixed-size files were mapped onto contiguous disk blocks). This introduced the possibility that there may not be a sufficiently large free area, either because of fragmentation or because the space had simply all

been used. The advent of 64-bit architectures has largely eliminated this problem. If containers were limited, for instance, to 48-bit addresses, then each of the 65,536 containers would be capable of storing $2x10^{14}$ bytes of data (significantly more than the entire addressing environment provided by the previous generation of 32-bit computers). In addition each container would only need to be mapped once in its lifetime.

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.

Essentially, we will create a "wrapping" or "database" container which implements the database through the component containers mapped into it. This database container will define the address space used by the DBMS software modules. According to the Grasshopper mapping facility, each of the mapped containers' managers will be responsible for providing its container data as required during DBMS operation. This structure leads to some interesting possibilities, including, for instance, the provision of a range of encryption and/or distribution strategies implemented independently for the mapped regions.

The authors expect that there will be a need for extensions to the current system calls provided by the Grasshopper persistent operating system. These extensions would, for instance, allow the database container manager to provide hints, based on observed access behaviour, to managers of mapped containers resulting in improved responses to system page discard requests. It is also 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 [9].

6. CONCLUSION

We have previously presented results demonstrating 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 capable of exploiting 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 support 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.

As the next stage of their research the authors intend to experiment with the mapping facility provided by Grasshopper in the expectation that independent management of sections of the database will result in database performance and utility improvements.

REFERENCES

1. Applegate, A. D. "Rethinking Memory Management", *Dr Dobb's Journal*, 19(6), 1994.
2. 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.
3. 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.
4. 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.
5. Cattell, R. G. G. and Skeen, J. "Object Operations Benchmark", *ACM Transactions on Database Systems*, 17(1), ACM, pp. 1-31, 1992.
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. Henskens, F. A. and Ashton, M. G. "Persistent Databases That Perform?", *Proceedings, IASTED International Conference on Software Engineering*, San Francisco, U.S.A, IASTED, 1997.
9. 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.
10. 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.
11. 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.
12. Pang, H., Carey, M. and Livney, M. "Managing Memory for Real-Time Queries", *SIGMOD*, 5/94, ACM, Minneapolis, Minnesota, USA, pp. 221-232, 1994.
13. 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.
14. 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.
15. Soltis, F. "Inside the AS/400", Duke Press, Loveland, Colorada, 1995.
16. Stephens, A. "C Database Development", MIS Press, New York, 1991.

17. Stonebraker, M. "Operating System Support for Database Management", *Communications of the ACM*, 24(7), ACM, pp. 412-418, 1981.
18. 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.