Operating System Support for Persistent and Recoverable Computations

[*]J. Rosenberg, [†]A. Dearle, [†]D. Hulse, [*]A. Lindström and [*]S. Norris

[†]Department of Computing Science
University of Stirling
Scotland
{al,dave}@cs.stir.ac.uk

[*]Basser Department of Computer Science
University of Sydney
Australia
{johnr,anders,srn}@cs.usyd.edu.au

Abstract

The principal tasks of an operating system are to manage the resources of the system, maintain the permanent data of the system and to provide an efficient environment for the execution of user programs. In conventional operating systems these tasks are centred around the file system as the repository of permanent data and virtual memory as the execution environment. Persistent systems offer an alternative view in which the lifetime of data is separated from the access mechanism. In a persistent system all data, regardless of its lifetime, is created and manipulated in a uniform manner. When persistence is included as the basic abstraction of an operating system, many of the inadequacies of existing operating systems are eliminated and the tasks of an application developer are greatly simplified. This results in major improvements both in terms of program development time and execution efficiency. Grasshopper, a persistent operating system being developed by the authors, provides a testbed for the demonstration of these claims.

Introduction

The principal tasks of an operating system are to manage the resources of the system, maintain the permanent data of the system and to provide an efficient environment for the execution of user tasks. In addition, users expect that the operating system will provide a level of resilience to failure and appropriate facilities to recover from failure with a minimum of interruption to computations and minimum loss of data. Most existing operating systems provide the resource management, permanent data maintenance and execution environment. However, there are two common inadequacies: the discontinuity between permanent and temporary data and the lack of resilience to failure.

The model of permanent data (a file system) is fundamentally different from the model of data supported in the execution environment (virtual memory). Consequently, permanent data must be accessed indirectly via the file system interface making it difficult to maintain complex data structures such as graphs. By contrast, arbitrary data structures may be created and manipulated in virtual memory, but these cannot persist longer than the lifetime of the creating program.

The support of two different data models results in a number of difficulties and potential inefficiencies:

(i) Programmers must determine the lifetime of their data early in the design process and write their program accordingly, potentially resulting in duplication of effort.

(ii) If the data embedded within a complex data structure is to be stored permanently, the programmer must write code to flatten the structure and copy it to a file, and corresponding code to reload it.

(iii) The programmer must deal with two different protection models.

Memory-mapped files are an attempt to blur this distinction. However, they are limited in their application, partly due to the lack of support for resilience and recovery.

Resilience of data and computations are essential for many applications. For example, a user editing a file expects that the file will not be lost if the system crashes. Indeed, they would prefer that all of the changes up to the time of the crash are included. Similarly, users with long running applications (e.g. simulations) would prefer it if these were automatically restarted from the point at which the crash occurred. Since the operating system does not include such services, they are added to each application on an ad hoc basis as discussed in the next section.

In 1981, Atkinson [1, 2] proposed that all data in a system should be able to survive for as long as that data is required; he called the attribute of longevity *persistence*. He also proposed that all data should be treated in a uniform manner regardless of the length of time for which it persists. That is, the persistence of data is orthogonal to its other attributes such as size, type, ownership etc. Systems that provide this abstraction are said to support *orthogonal persistence*. In this sense orthogonally persistent systems provide a uniform abstraction over all data storage. Furthermore, since the state of a process is just data, processes themselves may be made persistent [15] and may outlive a single invocation of a system.

Although persistence can be implemented by a programming language runtime system [2] it is our contention that the provision of support for persistence at the operating system level ensures the overall integrity of the data without restricting the system to a single language, and that such a persistent operating system provides a solution to the problems outlined above. Although virtually all of the examples cited in this paper can be implemented using a conventional operating system, the result is usually a somewhat contorted design and the programmer is forced to wrestle with the operating system in order to achieve the desired result. A persistent operating system provides a natural and elegant solution, whilst maintaining efficiency.

This paper describes the approach to persistence and resilience taken in Grasshopper [9], a persistent operating system being developed at the Universities of Stirling and Sydney. We first describe the various approaches to data management and demonstrate that the approach used in persistent systems removes the need for ad hoc techniques. This is followed by a discussion of the requirements of a persistent operating system. We then describe the persistence model provided by Grasshopper and show how it provides a uniform model of persistence and resilience. We conclude with some initial performance figures.

Data management

Almost all computer systems are concerned with the saving and recovery of dynamic state. In the light of this, a variety of ad hoc mechanisms have evolved to maintain dynamic state. Perhaps the most common example of this is the saving of documents in word processors and editors. In these applications the saved data is relatively simple consisting of linear strings of text. In other application areas, such as computer aided design, the data is much more complex, consisting of large pointer-based data structures containing objects of a variety of types. Such structures are considerably more difficult to save in either a file system or database.

As the complexity of the data that is saved and restored increases, so does the time taken to save and recreate the data set each time an application is run. In many cases application users have compromises forced upon them due to the complexity and cost of having programmers make the appropriate encodings. A good example of this is *core* files where we are forced to examine a flat data representation (a core file) of an extremely complex collection of data structures such as register sets, stacks and heaps.

In all the above cases, the data that is saved is separate from the computation that transforms the data. In other computations, it is the actual state of the computation that we wish to preserve. Consider a long lived computer simulation; we may wish to *snapshot* the state of the computation so that it may be recovered after a system failure. In this case there is extremely close coupling between the data that is saved: register values, stacks, memory state etc. and the application itself.

The technique of saving the state of an active application may also be applied to arbitrary application programs such as word processors and editors. For example, an entire window manager session could be saved and subsequently recovered at some later time. Such an approach would have many advantages; for example we would no longer require a plethora of ad hoc mechanisms such as .xsession, .Xrdb, .login, .cshrc and autoexec.bat to recreate *some* of the state of the user's environment since *all* of the dynamic state would be captured in the snapshot.

Persistent systems have no need for the ad hoc techniques described above. Since all data may persist for an arbitrary length of time, the original data structures used by applications may be maintained in their original form. Subsequent work on saved documents simply involves the application re-attaching itself to the persistent data structures. Similarly, the data and process driving a simulation will persist across system invocations. Startup files such as .cshrc, etc. are no longer needed since the environment they attempt to recreate is persistent.

An operating system supporting orthogonal persistence

Most existing persistent systems have implemented the abstraction of persistent data at the programming language level [3, 18, 20] and, as a result, have suffered from two drawbacks. First, the host operating system was not designed to support persistence; therefore the operating system interface does not usually provide abstractions sympathetic to a persistent language implementation. The consequence of this is that the language designer is usually forced to implement a persistent abstract machine above the operating system abstractions, with a corresponding loss of efficiency. A similar problem is reported by the designers of database systems [22, 24].

The second problem with this approach is that every persistent language implements its own persistent abstract machine duplicating much of the functionality found inside the operating system and other language implementations. Often these different implementations are entirely incompatible with each other, prohibiting interactions between programs written in different languages. This would appear to be a retrograde step compared to the mixed language environments supported by conventional systems.

The implementation of the orthogonal persistence abstraction by the operating system avoids these problems. We believe that such an approach to operating system design could be as revolutionary as virtual memory in terms of the advantages for user-level applications.

The requirements of a persistent operating system are quite different from those of conventional systems. Tanenbaum [23] lists the four major components of an operating system as being memory management, the file system, the input-output subsystem and process management. The nature of these four components is different in persistent systems. In a persistent system, the functionality of the file system and memory management are replaced by the persistent store. In many operating systems, notably Unix, input-output is presented using the same abstractions as the file system; clearly this is not appropriate in a persistent environment since there is no file system, and much of the input-output is eliminated by the single store abstraction. In most operating systems, processes are ephemeral entities; we have already argued the virtue of making processes persistent. It is therefore to be expected that an operating system designed to support persistence will have a different structure from a conventional operating system and will provide a different set of facilities.

We can summarise the principal requirements of such an operating system as follows [11]:

- support for persistent objects as the basic abstraction,
- consistent recoverability of data,
- support for persistent and resilient processes, and
- a uniform protection mechanism.

We call an operating system that provides all of these facilities a *persistent operating system*. It is important to distinguish such an operating system from a system which provides only limited support for the implementation of persistence. For example, both Spring [19] and Opal [6] have some notion of persistence, but this is not the central theme of these systems and they do not meet all of the requirements listed above. In particular they do not support resilience of data or persistent processes. Choices [4] takes a different approach in which the basic kernel abstraction of storage is the memory object. Inheritance is then used to special these into both file objects and persistent objects. Again, persistence is not ubiquitous and processes are transient.

Grasshopper

Grasshopper [9] is an example of a persistent operating system. In this section we describe the three basic abstractions provided by Grasshopper. The abstraction over storage is the *container* and the abstraction over execution is the *locus* (plural *loci*). The container in which a locus is currently executing is called its *host container.* Containers are repositories of data and may be of any size. The third basic abstraction is *capabilities* which provide control over access to Grasshopper entities.

Containers

Containers are the only storage abstraction provided by Grasshopper; they are persistent entities which replace both address spaces and file systems. In most operating systems, the notion of a virtual address space is associated with an ephemeral entity, a process, which accesses data within that address space. In contrast, containers and loci are orthogonal concepts. A Grasshopper system consists of a number of containers which may have loci executing within them. At any time, a locus can only address the data visible in the container in which it is executing. Grasshopper provides two facilities, *mapping* and *invocation*, which allow the transfer of data between containers.

The purpose of container mapping is to allow data to be shared between containers. This is achieved by allowing data in a region of one container to be viewed within a region of another container. Mappings may be either global, or private to a particular locus while executing in a particular container. Unlike the memory object mechanism provided by other systems [7], containers may be arbitrarily (possibly recursively) composed which provides considerably enhanced flexibility and performance [17].

Since any container can have another mapped into it, it is possible to construct a hierarchy of container mappings which forms a directed acyclic graph as shown in figure 1. The restriction that mappings cannot contain circular dependencies is imposed to ensure that one container is always ultimately responsible for data. In Figure 1, container *C2* is mapped into container *C1* at location *a1*. In turn, *C2* has regions of containers *C3* and *C4* mapped into it. The data from *C3* is visible in *C1* at address *a3,* which is equal to *a1* + *a2*.

Loci

In its simplest form, a locus is simply the contents of the registers of the machine on which it is executing. Like containers, loci are maintained by the Grasshopper kernel and are inherently persistent.

A locus is associated with a *host* container. The locus perceives the address space of the host container plus any privately mapped containers. Virtual addresses generated by the locus map directly onto addresses within the host container and the privately mapped containers. A container comprising program code, mutable data and a locus forms a basic running program. Loci are an orthogonal abstraction to containers; any number of loci may execute simultaneously within a given container.

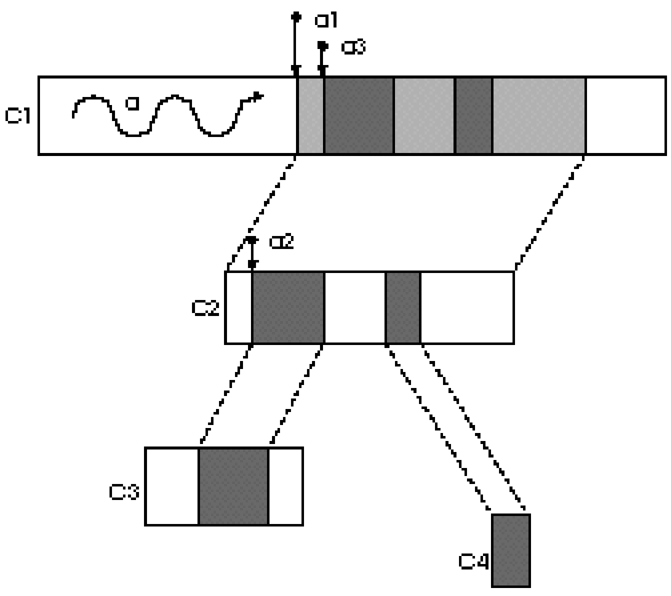

Figure 1: A container mapping hierarchy

Inter-Container Communication

An operating system is largely responsible for the control and management of two entities: objects, which contain data (containers); and processes (loci), the active elements which manipulate these objects. One of the most important considerations in the design of an operating system is the model of interaction between these entities. Grasshopper uses the object-thread model in which communication is achieved via a mechanism similar to procedure calls in which threads (loci) move between entities. Thus a locus may *invoke* a container thereby changing its host container and may later return to the original container.

Naming and Protection

In the previous sections we have described the basic abstractions in Grasshopper and the operations over these abstractions. Given that containers are the *only* abstraction over storage (i.e. there is no file system), a naming and access control mechanism is required and this is provided by *capabilities* which are protected by segregation [12].

In Grasshopper, every container and locus has an associated list of capabilities [8]. A capability list is constructed from tuples containing a unique fixed length key and a capability. Operations are provided for copying capabilities and for adding and removing them to and from lists. The capability mechanism is deliberately simple and low-level for reasons of efficiency and flexibility. Higher level naming mechanisms, e.g. name servers, are implemented as user-level containers using the operations described above.

Managers

Thus far we have described how all data storage in Grasshopper is provided by containers. However, we have not described how containers are populated with data. This is the responsibility of *managers* which are also user-level entities. The use of managers is motivated by the desire, as far as practicable, to leave all *policy* decisions out of the kernel. The kernel provides *mechanisms* which can be used by higher level software to implement required policies. This provides maximum flexibility and avoids the kernel making decisions which impact upon performance. For example, the memory management policy can have major effects on the performance of garbage collection. User-level virtual memory management, supported on a number of recent operating systems [14], has a similar motivation.

Each container has an associated manager, which is an ordinary user-level program, held within a container. The manager is responsible for:

- provision of the pages of data stored in the container,
- responding to access faults,
- operation within a limited amount of physical memory (page discard),
- implementation of a stability algorithm for the container [10], i.e. maintenance of the integrity and resilience of data, and

The kernel provides a standard framework in which managers may operate. This includes automatic invocation of the appropriate manager on an access fault, and a set of interfaces which allow managers to arrange the hardware translation tables in such a way that the required data is visible at an appropriate address in the container. Thus managers provide user-level virtual memory management in common with several other operating systems [25].

Persistence, Recoverability and Concurrency Control

The Grasshopper kernel treats loci and the data accessed by them during computation as the unit of recovery. Loci are able to snapshot the state of their computation at any time, a task which is coordinated by the kernel and draws on services provided by the managers to snapshot user level data. A snapshot consists of all the data related to the computation of a locus and includes:

1. any modified container data seen by the locus,
2. any data maintained within the kernel to represent the state of the locus (including the registers) and the containers in which it has executed.

Since a locus can move between containers during the course of its computation, a snapshot typically involves recording the state of pages within a number of different containers. In contrast to other persistent systems in which a snapshot involves making the entire persistent store stable, the snapshot mechanism in Grasshopper only affects the stability of the portions of containers seen during the computation of a particular locus. Since loci are free to use shared memory as a means of inter-process communication, the actions of one locus can be influenced by the actions of another. This interaction creates causal dependencies between loci. During the normal operation of the system it is possible to ignore these causal dependencies because they are automatically preserved. However, if the system needs to be restarted after a shutdown or crash, locus snapshots must be used to rebuild a consistent system state.

It is therefore necessary to detect causal dependencies and ensure that they are preserved across failure of the system, thus guaranteeing global consistency. Detection of causal dependencies is performed by the kernel and managers which monitor read and write faults to compile modified page lists containing an entry for every modified page seen by a locus since its last snapshot. In addition, the kernel also maintains a list of containers in which a locus has seen modified data. The kernel uses this list to determine which managers it must request to snapshot data modified by the snapshotting locus.

The kernel coordinates the processing of locus snapshots and maintains dependency information such that it is possible to recover the state of the system from a causally consistent set of locus snapshots following a failure. Causal dependencies between loci are represented using vector time [13]. Each locus has an associated vector time which is lazily updated whenever a snapshot is performed [10]. The vector time contains a list of pairs representing the state of each computation on which the snapshotting locus is dependent. Each pair contains the identity of a locus and a timestamp derived from a Lamport clock [16] associated with the locus which is used to identify points in time during its execution. This information is sufficient to characterise the causal dependencies of loci and their snapshots [10].

The above mechanism guarantees that a Grasshopper system will always recover data and processes to a self consistent state. It does not guarantee that the snapshotted state of the system was semantically consistent when a snapshot was made. Such guarantees require either cooperation or exclusion at the application level. Grasshopper supports these activities in two ways. First mechanisms that allow concurrent loci to cooperate are provided. These mechanisms include semaphores and conditional locks. Secondly the Grasshopper kernel and managers cooperate to provide those loci that require it with transactional semantics.

Applications

The Grasshopper model effectively provides the programmer with resilient and recoverable data and processes. In this section we return to the examples of ad hoc data management discussed earlier and show how the Grasshopper mechanisms provide a coordinated and simple solution to the problems described.

Applications such as word processors, spreadsheet programs, CAD systems, editors, etc are required to operate on different sets of data at different times. In conventional systems the permanent state of these documents is held in files. In Grasshopper each document is held in its original form in a container. The application code is also held in a container. Each document container has its corresponding application container mapped into itself with the invocation point set to the start of the application code. Thus the application may be initiated to operate on any document simply by invoking that document container. The application has direct access to the data structures representing the document and the costs associated with converting the document to and from its file format are totally avoided; no recovery code need be written by the application developer. In addition the capability system can guarantee that the internal representation of the documents cannot be accessed by other programs. It should be stressed that this is only one approach and several other techniques using the Grasshopper mechanisms are possible.

Compilation systems present an excellent example of simplifications introduced by persistence. Typically compilers provide an option to embed symbolic information which can be used by the debugger within the generated executable code. The information is essentially a flattened copy of the symbol table produced and used during the compilation process. In Grasshopper, the symbol table can outlive the compilation in its original form. This could either be stored in the generated code container or in a separate container for use by tools such as the debugger. Similarly, intermediate representations of the code can be maintained to improve error reporting or to enable automatic re-generation of machine code if the application is moved to a different architecture. This latter approach is used on the AS/400 [21]. It is important to note that these facilities can be provided with very little change to the compilation system. The data structures already exist and Grasshopper automatically makes them persistent. The use of strongly-typed programming languages [18] can ensure the integrity of such stored data structures.

The third example is long-lived applications such as simulations. In Grasshopper these applications and the corresponding loci are automatically resilient. The application programmer need not write any special recovery code. The system guarantees that, following a crash, the loci executing these applications automatically restart from the last consistent state. This results in considerable savings in terms of programmer productivity and program development time.

The cost of construction of a typical Unix/X-windows environment can be significant, both in terms of execution time and user time. Users typically create several windows and establish particular environments within those windows. In conventional systems this environment must be recreated each time the user logs on, a process which may involve the execution of a number of scripts as well as gestures by the user in order to re-establish the state within windows. In Grasshopper this cost is eliminated. Since the environment (including open windows) is represented by data structures in containers, it is automatically persistent. Furthermore, any loci (processes) associated with the environment are also persistent. Thus, login simply corresponds to reconnecting to the environment and no user provided start-up code or initial gestures need be performed. In addition, users may create many different environments and connect to the one most appropriate for the task at hand. For example, there may be a program development environment, a word processing environment, etc. Again, no special user-level code need be written to achieve this flexibility; it is all a direct result of persistence.

Performance

Performance of an operating system is extremely difficult to evaluate because of the close dependency on the particular applications being executed. The major differences between Grasshopper and conventional operating systems are in two areas: first, the model of address spaces and processes and second, the support of orthogonal persistence.

Initial measurements of the invocation mechanism indicate that it is significant improvement over similar mechanisms in other systems. For example, the Opal group reports that cross address space communication on the Mach microkernel takes about 88 μs whereas a Grasshopper invocation takes around 20 μs on identical hardware. This result corroborates other studies that show that allowing a single computation to cross address space boundaries provides superior performance to message-based RPC. It is interesting to note that a variant of this facility is being considered as an extension to Mach.

The cost of persistence is harder to determine. Here we present some initial results from the OO1 benchmark developed by Cattell and Skeen [5]. A simple physical storage model based on B-tree indexes was used for the experiment. The small OO1 database consisting of 20,000 parts and occupying about 7 megabytes was measured. The operations include random lookup, directed search (traversal) and insertion of new records. Although the database is quite small it does provide useful figures since most of the cost is related to loading the data from the store and address translation.

Three different versions were measured. In all cases the programs were run on a 133 Mhz DEC Alpha with 64 Megabytes of memory, using both OSF Unix and Grasshopper. The first version used the Unix file system to access the database. The second version used memory-mapped files under Unix and the third version used persistent containers under Grasshopper. The results for each of the operations are shown in Table 1. All figures are in seconds and the experiments were repeated one hundred times and the results averaged.

Operation	OSF: Files	OSF: Memory-mapped	Grasshopper
Lookup	1.45	0.20	0.11
Traversal	7.86	0.85	0.47
Insert	1.70	0.27	0.18
Total	10.01	1.32	0.76

Table 1: Performance on the OO1 Benchmark

The figures indicate that Grasshopper is approximately twice as fast as OSF using memory-mapped files. This is extremely promising when it is considered that Grasshopper provides full recoverability whereas no such support is provided by OSF. In addition it should be noted that no effort has been expended in optimising the Grasshopper structures or algorithms.

There are two points to be made about these figures. First, Grasshopper performs the benchmark approximately fifteen times faster than the file-based version. This is the sensible comparison since conventional systems access file data via the file system. Second, although OSF supports memory-mapped files which can run the benchmark at a speed closer to that of Grasshopper, these are totally non-resilient and therefore could not be used for "real" applications.

The above figures give us confidence that Grasshopper can compete with conventional systems and provide an efficient computing platform.

Conclusions

A significant proportion of the effort spent developing an application is devoted to dealing with issues of storage and recoverability. This is because most existing operating systems provide a severely limited long-term storage data model and little support for recoverability, resilience and consistency of recovered data. Grasshopper provides persistent containers and loci as its base abstractions and guarantees their recoverability; following a system failure, they will be recovered to an globally self-consistent state.

The provision of these guarantees by the operating system, coupled with the ability to create arbitrarily long-lived data structures, considerably simplifies the development of application programs and encourages the construction of integrated systems. The programmer is not required to write any code in order to save a data structure and all programs are automatically recoverable and resilient. This results in major improvements in terms of both program development time and execution efficiency.

A first implementation of the Grasshopper operating system is nearing completion. This operates on the DEC Alpha range of machines. Initial experiments with development of applications in (persistent) C confirm our expectations.

Acknowledgements

This work is supported by grant number A49530086 from the Australian Research Council and by an equipment donation from Digital Equipment Corporation under the Alpha Innovators Program. We would also like to thank Ewa Bem, Soren Brandt, Matty Farrow and Aoife Kavanagh for their constructive comments on earlier drafts of this paper and Maurice Ashton for his work on the OO1 benchmark.

References

[1] Atkinson, M. P., Bailey, P., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, vol 26, 4, pp. 360-365, 1983.

[2] Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17(7), pp. 24-31, 1981.

[3] Brown, A. L. and Cockshott, W. P. "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews, PPRR-13, 1985.

[4] Campbell, R. H., Islam, N. and Madany, P. "Choices, Frameworks and Refinement", *Computing Systems*, vol 5, 3, 1992.

[5] Cattell, R. G. G. and Skeen, J. "Object Operations Benchmark", *ACM Transactions on Database Systems*, vol 17, 1, pp. 1-31, 1992.

[6] Chase, J. S., Levy, H. M., Feeley, M. J. and Lazowska, E. D. "Sharing and Protection in a Single Address Space Operating System", *ACM Transactions on Computer Systems*, vol 12, 4, 1994.

[7] Cheriton, D. R. "The V Kernel: A Software Base for Distributed Systems", *Software*, vol 1, 2, pp. 9-42, 1984.

[8] Dearle, A., di Bona, R., Farrow, J. M., Henskens, F. A., Hulse, D., Lindström, A., Norris, S., Rosenberg, J. and Vaughan, F. "Protection in the Grasshopper Operating System", *Proceedings of Sixth International Workshop on Persistent Object Systems*, Springer-Verlag, Tarascon, France, pp. 60-78, 1994.

[9] Dearle, A., di Bona, R. M., Farrow, J. M., Henskens, F. A., Lindström, A. G., Rosenberg, J. and Vaughan, F. "Grasshopper: An Orthogonally Persistent Operating System", *Computer Systems*, vol 7, 3, pp. 289-312, 1994.

[10] Dearle, A. and Hulse, D. "On Page-based Optimistic Process Checkpointing", *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, pp. 24-32, 1995.

[11] Dearle, A., Rosenberg, J., Henskens, F. A., Vaughan, F. and Maciunas, K. "An Examination of Operating System Support for Persistent Object Systems", *Proceedings of the 25th Hawaii International Conference on System Sciences*, vol 1, (ed V. Milutinovic and B. D. Shriver), IEEE Computer Society Press, Hawaii, U. S. A., pp. 779-789, 1992.

[12] Dennis, J. B. and Van Horn, E. C. "Programming Semantics for Multiprogrammed Computations", *Communications of the A.C.M.*, 9(3), pp. 143-145, 1966.

[13] Fidge, L. J. "Timestamps in Message Passing Systems that Preserve Partial Ordering", *Proc. 11th Australian Computer Science Conference*, pp. 56-66, 1988.

[14] Harty, K. and Cheriton, D. R. "Application-Controlled Physical Memory using External Page-Cache Management", *ASPLOS V*, ACM, Boston, 1992.

[15] Keedy, J. L. and Vosseberg, K. "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System", *Proceedings of the 25th Hawaii International Conference on Systems Sciences*, vol 1, IEEE, Hawaii, USA, pp. 747-756, 1992.

[16] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol 21, 7, pp. 558-565, 1978.

[17] Lindström, A., Rosenberg, J. and Dearle, A. "The Grand Unified Theory of Address Spaces", *Proceedings of Fifth Workshop on Hot Topics in Operating Systems*, I.E.E.E. Press, Orcas Island, U.S.A., pp. 66-71, 1995.

[18] Morrison, R., Brown, A. L., Carrick, R., Connor, R., Dearle, A. and Atkinson, M. P. "The Napier Type System", *Persistent Object Systems - Proceedings of the Third International Workshop*, (ed J. Rosenberg and D. Koch), Springer-Verlag, pp. 3-18, 1989.

[19] Radia, S., Madany, P. and Powell, M. L. "Persistence in the Spring System", *Proceedings 3rd International Workshop on Object Orientation in Operating Systems*, IEEE Press, pp. 12-23, 1993.

[20] 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.

[21] Soltis, F. "Inside the AS/400", Duke Press, Loveland, Colorado, 1995.

[22] Stonebraker, M. "Virtual Memory Transaction Management", *Operating Systems Review*, vol 18, 2, pp. 8-16, 1984.

[23] Tanenbaum, A. S. "Modern Operating Systems", Prentice-Hall International, pp. 385-387, 1992.

[24] Traiger, I. "Virtual Memory Management for Database Systems", *Operating Systems Review*, vol 16, 4, pp. 26-48, 1982.

[25] Young, M. W. "Exporting a User Interface to Memory Management from a Communications-Oriented Operating System", PhD Thesis, Carnegie-Mellon University, 1989.