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.

1. 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. This may result 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. This results in a significant amount of programmer effort and execution overhead.

(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 second inadequacy mentioned above, lack of support for resilience and recovery. Systems such as Unix provide no guarantees about the state of files following a crash. Instead, utilities are provided to check and repair file system structures; however, these do not guarantee the integrity of the data stored in the files. It is not unusual for files, or parts of files to be lost following a crash, leaving data in related files in an inconsistent state. However, in addition to passive data, operating systems also support active data (processes) and these systems provide no support for recovery of computations, which must be manually restarted.

Although resilience of data and computations is not provided by conventional operating systems, both of these 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 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 [9, 10] 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 attribute 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 [25] and may outlive a single invocation of a system.

A number of approaches to the construction of persistent systems have been adopted. These include the design of programming languages with integrated support for persistence [8, 10, 30], operating systems [13, 14, 17, 19] and new hardware architectures [26, 34]. Of these approaches, we favour the development of an operating system. 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.

It is our contention that 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, a persistent operating system being developed at the Universities of Stirling and Sydney. The paper is organised as follows. 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.

2. Data Management

2.1 Ad hoc solutions

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. Another common example is in compilers where parse trees are saved in some flat format as they are passed between different phases of the compilation.

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. Applications, such as remote file system servers, effectively simulate this restartability by maintaining enough state information in files and having the operating system restart them each time the system is re-booted. The results of this approach for systems such as NFS locking are notorious.

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 need 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.

2.2 Integrated solutions – persistence

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.

3. An operating system supporting orthogonal persistence

3.1 Why?

Since 1978 a large number of researchers have constructed systems which support orthogonal persistence [1, 2, 3, 4, 5, 6, 11, 29], most of which are programming language systems. A number of persistent languages have been developed either from scratch [8, 10, 30] or as an extension to an existing language [31, 32, 33, 35, 36]. These usually provide a large *store* within which concurrent processes manipulate persistent data. In some of these systems, the stores contain all data including procedures, graphics objects, processes and their associated state [12].

However, implementing the abstraction of persistent data at the programming language level suffers from two drawbacks. The first of these is that 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 [38, 41].

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.

We therefore believe that the abstraction of orthogonal persistence should be implemented by the operating system. 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. In the following section we briefly outline the requirements of a persistent operating system.

3.2 Requirements

Tanenbaum [40] 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 [21]:

i. The major requirement is support for persistent objects as the basic abstraction. Persistent objects consist of data (including code) and relationships with other persistent objects; the system must therefore provide a mechanism for supporting the creation and maintenance of these objects and relationships. This mechanism should be based upon a uniform addressing scheme used by all processes to access objects. That is, all processes should (potentially) have access to all data using the same addressing scheme. This is essential for orthogonal persistence.

ii. A further requirement is that these objects must be both stable and resilient. The system must reliably manage the transition between long and short term memory transparently to the programmer.

iii. Processes must be integrated with the object space in such a way that process state is itself contained within persistent objects. The importance of this is that processes themselves become resilient.

iv. Although the persistent store is uniform, there is still a requirement to be able to restrict access to objects for the same reasons that file systems contain access control mechanisms. Any operating system supporting persistence must therefore provide some protection mechanism.

We call an operating system that provides these facilities a *persistent operating system*.

4 Grasshopper

Grasshopper 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*. 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.

4.1 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. Unlike the memory object mechanism provided by other systems [15, 16], containers may be arbitrarily (possibly recursively) composed which provides considerably enhanced flexibility and performance [28].

Since any container can have another mapped into it, it is possible to construct a hierarchy of container mappings as shown in Figure 1. The hierarchy of container mappings forms a directed acyclic graph. 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*.

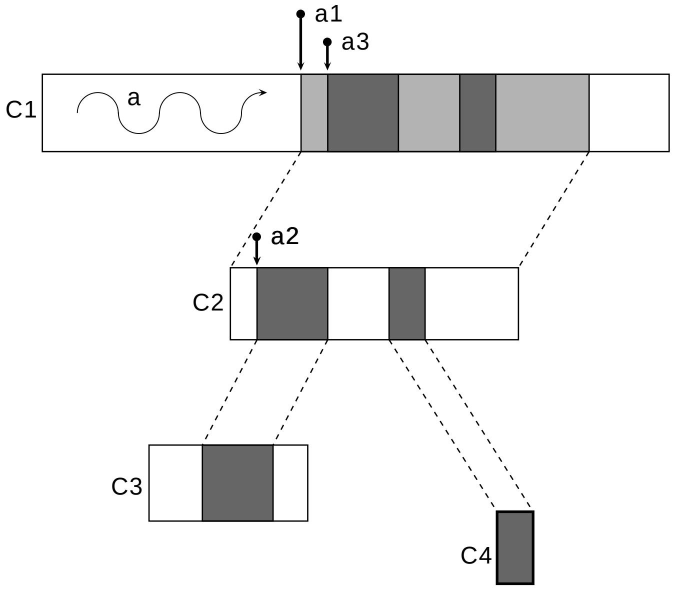

Figure 1: A container mapping hierarchy

Loci perceive the address space of their host container. Therefore, all loci executing within a container share the same address space. However, a locus may require private data, which is visible to it, yet invisible to other loci that inhabit the same container. To satisfy this need, Grasshopper provides the notion of *locus private mappings* which are visible only to the locus which created them and take precedence over host container mappings. For example, this permits each locus to have its own stack with all stacks occupying the same address range within the host container [28].

4.2 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.

4.3 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 procedure calls and threads (loci) move between entities [27]. Thus a locus may *invoke* a container thereby changing its host container and may later return to the original container.

Any container may include, as one of its attributes, a single entry point known as an *invocation point*. When a locus invokes a container, it begins executing code at the invocation point. The single invocation point is important for security since it is the invoked container that controls the execution of the invoking locus by providing the code that will be executed.

The kernel appears to user-level applications as a set of containers. Thus, access to operating system functions is also achieved by invocation. This provides a uniform interface for applications and blurs the distinction between system and user functions.

4.4 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), some access control mechanisms are required.

In a conventional operating system many of these controls are provided by the file system which maintains access lists, usually on a hierarchical basis. This is not appropriate in Grasshopper since there is no file system. Some persistent systems use the type system to provide control over access, however, as we have stated earlier, we propose to support multiple languages with different type systems, and so this is not an alternative. For these reasons we believe that it is essential for Grasshopper to support a third abstraction, a naming and protection mechanism, in the form of *capabilities* [22, 43].

In Grasshopper, every container and locus has an associated list of capabilities [18]. 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. At any time, a locus has access to:

i. all the capabilities in its own list,

ii. all capabilities in its host container's list,

Programs can refer to capabilities by specifying a capability list (locus or host container) and a key. Grasshopper checks that an entry with the given key exists in the specified list. An appropriate capability must be presented for operations involving the manipulation of entities, such as invocation, mapping, and blocking and unblocking of loci. Since kernel services are requested by invocation, access to these may also be controlled by capabilities.

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.

4.5 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 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 [7, 23, 44], 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 [20], i.e. maintenance of the integrity and resilience of data, and
- maintenance of coherence in the case of distributed access to the container.

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 recent operating systems.

4.6 Consistency

Containers and their associated managers provide the abstraction of persistent data. Managers are responsible for maintaining a consistent and recoverable stable copy of the data represented by the container. As part of its interface, each manager must provide a *stabilise* operation. Stabilisation involves creating a consistent copy of the data on a stable medium.

Managers alone are not able to maintain a system-wide consistent state since there may be dependencies between the loci executing within different containers. The state of a locus is defined by a set of registers, its kernel state and the set of modified pages that the locus has accessed since its last snapshot. In Grasshopper, a locus may snapshot this state upon which the kernel requests the managers of the containers which hold these pages to perform a snapshot, thereby incorporating the changed state of the modified pages into the recoverable state.

Since loci may be dependent on the results of computation performed by other loci, it is necessary to detect these dependencies and ensure that they are preserved across failure of the system; this guarantees global consistency which is the responsibility of the Grasshopper kernel. 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 [24, 39, 42]. Details of these techniques are beyond the scope of this paper.

Kernel data must also be made persistent and recoverable; in this way, the kernel is part of the persistent environment, thereby extending the concept of an operating system instance. A Grasshopper kernel persists even when the host machine is not operating. Conventional operating systems rebuild the operating system from scratch each time they are bootstrapped. In Grasshopper, the entire kernel, operating system and user state persists. After an initial bootstrap, an entire self-consistent state is loaded and continues execution.

5 Recoverability, Resilience and 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 in section 2 and show how the Grasshopper mechanisms provide a coordinated and simple solution to the problems described.

5.1 Generic Applications

Generic applications are programs which operate on data to achieve a particular purpose. Different sets of data are operated on at different times. Examples include word processors, spreadsheet programs, CAD systems, editors, etc. In conventional systems the permanent state of the 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.

5.2 Intermediate Representations

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 [37]. 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.

5.3 Long-lived Applications

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.

5.4 Environments

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.

6 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 Maurice Ashton, Ewa Bem, Soren Brandt, Matty Farrow and Aoife Kavennagh for their constructive comments on earlier drafts of this paper.

References

[1] "Proceedings of the International Workshop on Database Programming Languages", 1987.

[2] "Datatypes and Persistence", *Proceedings of Data Types and Persistence Workshop Aug. 1985*, (ed M. P. Atkinson, P. Bunerman and R. Morrison), Springer-Verlag, 1988.

[3] "Persistent Object Systems", *Proceedings of the 3rd International Workshop on Persistent Object Systems*, (ed J. Rosenberg and D. M. Koch), Springer-Verlag, 1989.

[4] "Proceedings of the International Workshop on Database Programming Languages", Morgan Kaufmann, 1989.

[5] "Proceedings of the 4th International Conference on Persistent Object Systems", (ed A. Dearle, G. Shaw and S. Zdonik), Morgan-Kauffman, 1990.

[6] "Security and Persistence", *Proceedings of the International Workshop on Architectures to Support Security and Persistence of Information*, (ed J. Rosenberg and J. L. Keedy), Springer-Verlag, 1990.

[7] Abrossimov, V., Rozier, M. and Shapiro, M. "Generic Virtual Memory Management for Operating System Kernels", *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, The Wigwam, Litchfield Park, Arizona, ACM, pp. 123-136, 1989.

[8] Albano, A., Cardelli, L. and Orsini, R. "Galileo: A Strongly Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems*, 10(2), pp. 230-260, 1985.

[9] 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.

[10] 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.

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

[12] 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.

[13] Campbell, R. H., Johnston, G. M. and Russo, V. F. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)", *ACM Operating Systems Review*, vol 21, 3, pp. 9-17, 1987.

[14] 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.

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

[16] Chorus Systems "Overview of the CHORUS Distributed Operating Systems", *Computer Systems - The Journal of the Usenix Association*, vol 1, 4, 1990.

[17] Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. "The Clouds Distributed Operating System", *Proc. 8th International Conference on Distributed Computing Systems*, 1988.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] Johnson, D. B. and Zwaenepoel, W. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing", *Proc. 7th Symposium on Principles of Distributed Computing*, ACM, pp. 171-181, 1988.

[25] 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.

[26] Koch, D. M. and Rosenberg, J. "A Secure RISC-based Architecture Supporting Data Persistence", *Proceedings of the International Workshop on Architecture to*

Support Security and Persistence of Information, Workshops in Computing, Springer-Verlag, Bremen, Germany, pp. 188-201, 1990.

[27] Lauer, H. C. and Needham, R. M. "On the Duality of Operating System Structures", *Operating Systems Review*, 13(2), pp. 3-19, 1979.

[28] 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., 1995.

[29] McLellan, P. and Chisholm, K. "Implementation of a POMS: Shadow Paging via VAX VMS Memory Mapping", Unpublished Report, 1982.

[30] 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.

[31] Moss, E. and Sinofsky, A. "Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface", *Advances in Object-Oriented Database Systems*, Springer-Verlag, pp. 298-316, 1988.

[32] 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.

[33] Rosenberg, J. "Pascal/M - A Pascal Extension Supporting Orthogonal Persistence", Department of Computer Science, University of Newcastle, Technical Report 89/1, 1989.

[34] Rosenberg, J. and Abramson, D. A. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", *Proc. 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

[35] Schmidt, J. W. "Some High Level Language Constructs for Data of Type Relation", *ACM Transactions on Database Systems*, 2(3), pp. 247-261, 1977.

[36] Shapiro, M., Gautron, P. and Mosseri, L. "Persistence and Migration for C++ Objects", *Proceedings, European Conference on Object-Oriented Programming (ECOOP)*, 1989.

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

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

[39] Strom, R. E. and Yemini, S. A. "Optimistic Recovery in Distributed Systems", *ACM Transactions on Computer Systems*, vol 3, 3, pp. 204-226, 1985.

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

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

[42] Vaughan, F., Dearle, A., Cao, J., di Bona, R., Farrow, M., Henskens, F., Lindström, A. and Rosenberg, J. "Causality Considerations in Distributed Persistent Operating Systems", *Proc. 17th Annual Computer Science Conference (ACSC-17)*, pp. 409-420, 1994.

[43] Wulf, W. A., Levin, R. and Harbison, S. P. "HYDRA/C.mmp: An Experimental Computer System", McGraw-Hill, New York, 1981.

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