

# Operating System Support for Persistent Systems: Past, Present and Future

ALAN DEARLE

*School of Mathematical and Computational Sciences, University of St Andrews, North Haugh,  
St Andrews KY16 9SS, Scotland  
email: al@dcs.st-and.ac.uk*

DAVID HULSE

*Department of Computing Science and Mathematics  
University of Stirling  
Stirling, FK9 4LA, Scotland  
email: dave@cs.stir.ac.uk*

## SUMMARY

Since the 1980s, various groups have been constructing systems that support a concept known as *orthogonal persistence*. These systems support objects whose lifetime is independent of the context in which they were created. The benefits of such systems include greater run-time efficiency, strong semantic guarantees about the existence of data and its type, early error checking, and lower construction costs. However, the implementation of persistent systems has been hindered by the lack of support provided by the operating system. This paper examines the implementation of persistent systems on traditional operating systems and on operating systems that directly support persistence, and looks at current attempts to provide flexible architectures that permit persistence to be provided efficiently above the operating system.

**KEY WORDS:** persistent systems, operating systems, persistent operating systems, micro-kernels, exokernels

## 1. INTRODUCTION

The first systems supporting *orthogonal persistence* [5] were programming language systems constructed in the early 1980s. These systems realised two ideals: that all data should be able to persist (survive) for as long as that data was required and that all data was manipulated in a uniform manner regardless of the length of time for which it persists.

Over the past fifteen years, much research effort has been expended in attempting to build systems that support orthogonal persistence. Many of these have been language systems implemented above traditional operating systems such as VMS, VME and Unix. The literature contains many reports of problems associated with the construction of persistent systems using such operating systems<sup>1</sup>. This has led some, including the authors, to attempt to provide explicit operating system support for persistent systems. This paper surveys approaches to implementing orthogonal persistence on conventional operating systems and examines attempts to provide explicit operating system support for persistent systems. It also describes a new operating system designed to support persistent systems based on current trends in operating system design and on the lessons learned from previous attempts to provide operating system support for persistence.

---

<sup>1</sup> The interested reader is referred to the Persistent Object Systems Workshop series published by Morgan Kaufmann and Springer-Verlag.

The paper is structured as follows. Section 2 describes the requirements for implementing persistence and looks at common techniques for meeting those requirements. By examining these techniques it attempts to identify deficiencies in traditional operating system support for the implementation of persistent systems. These deficiencies have much in common with those reported by implementers of database systems [61] [13].

Section 3 surveys some operating systems that have attempted to provide direct operating system support for orthogonal persistence. In each case, only those operating system features related to persistence are discussed. Section 4 analyses the lessons learned from these attempts. In particular, it examines in detail the Grasshopper Operating System [25] using the specific domain knowledge of the authors who were among the designers and implementers of this system. We attempt to extrapolate from our experiences with Grasshopper to draw some conclusions about operating system support for persistence in general. In particular we argue the inappropriateness of (all) operating system abstractions for persistence.

Section 5 surveys general trends in operating system architectures and seeks to identify appropriate architectures that may be used to support persistent systems. Based on lessons learned in Section 4 and from our observations of the systems described in Section 5, we embarked on the construction of a new operating system architecture. Our new operating system, called Charm, is described in Section 6.

Section 7 shows how a persistent application environment similar to that provided by Grasshopper may be constructed using Charm. Space precludes showing how all the persistent models discussed in Section 2 may be implemented. However, we believe that Section 6 contains enough information to convince the reader that his or her favourite persistence mechanism could be efficiently implemented using the Charm infrastructure.

## 2. Techniques used to implement persistence

In this section we attempt to identify the techniques used to implement persistence. This knowledge may be used to identify requirements made of any operating system whose aim is to support orthogonally persistent systems efficiently.

### **Support for persistent objects and relationships**

Conventional operating systems<sup>2</sup> expect to support processes that access only short-lived data held in directly addressable physical or virtual memory. Long-lived data is held in secondary storage and cannot be directly addressed. This has led to the establishment of separate operating system APIs for each class of data creating a

dichotomy between long- and short-lived data. For example, in Unix one set of system calls is responsible for the allocation of virtual memory (e.g. `brk`) and others that deal with files (e.g. `open`, `close`, etc.). In contrast, systems that provide orthogonal persistence treat all data identically as persistent objects and a single API is provided to manipulate these objects. This leads to a requirement that objects must be uniformly addressable and moved between long- and short-term storage in a manner that is transparent to the application programmer. The engineering solutions that have evolved to satisfy this requirement may be partitioned into two categories:

1. Those that utilise software address translation as typified by Brown's stable store [10], and
2. Those that utilise hardware address translation.

In Brown's stable store [10], which is typical of systems that utilise software address translation, two address spaces are managed: a local process address space and a persistent address space. The process address space contains objects that may be directly accessed by machine instructions. Long-lived objects are stored in the persistent address space which are stored on disk. The storage system moves objects transparently from one address space to the other on demand. This requires software address translation between the local and persistent address spaces. The impact of the cost of address translation in persistent systems is not clear due to the lack of sufficient measurement. For example, Moss asserts that software address translation is more efficient than utilising hardware mechanisms as exposed by conventional operating systems [45]. This may be a condemnation of the time modern operating systems take to service a trap rather than praise for software addressing techniques. However, since the ratio of processor speed over time required to service a trap is increasing, this is likely to become increasingly true in the future.

The memory-mapping abstraction provided by most modern Unix systems permits regions of the file space to be mapped into virtual memory and accessed transparently (that is, without needing to invoke the standard file system API). However, this mechanism was not designed to support large address spaces such as those found in persistent systems. This leads to problems, for example the SunOS memory-mapped file implementation of `Napier88` suffers from slow start up times due to the operating system's eager construction of large virtual address maps. Another problem associated with memory mapping is that there is little or no control over the location to which evicted data is swapped which in turn makes failure recovery more complex and less efficient. This may, in part, be addressed by the use of external pagers.

Early micro-kernels such as Mach [3] and Chorus [57] provide external pager mechanisms that permit the implementation of an application-specific virtual memory manager. Both of these systems are deficient in that

---

<sup>2</sup> By conventional operating systems we mean systems such as OS360, VMS, VME, NT, Unix, Windows '98.

they do not permit total control over the virtual address space. In particular, in both systems it is the kernel that selects pages to remove from an address space when the supply of free physical memory runs low. This functionality causes many problems for the persistent system implementer who may be using memory in complex ways unknown to the kernel.

No widely used operating system constructed to date provides sufficiently flexible mechanisms to enable the hardware facilities to be exploited to their full potential. However, the trend in operating system support is to expose more hardware functionality to user applications. We shall explore this in Section 5.

### **Resilience mechanisms**

The requirement for resilience is not peculiar to persistent systems. However, the problem of consistency is perhaps more acute with persistent systems. In a conventional file system, each file is essentially an independent object. Therefore, the loss of a single file following a crash does not threaten the integrity of the overall system. In a persistent system, the data often consists of a large number of relatively fine-grain, typed objects, each of which might cross-reference a number of other objects in the store. Therefore, the loss of a single object can result in total system failure. When a collection of objects is modified, their state must be written to non-volatile storage in a consistent and atomic manner to avoid the possibility of corruption. A number of different techniques have been used to provide resilience in persistent systems including:

1. Shadow-paged stores [40] [10]
2. Log-based stores [27, 42, 58, 60], and,
3. Log-structured stores. [31, 32]

These different store technologies all require control over the granularity of disk reads and writes as well as when and where data is written to disk. Often operating systems provide little control over these aspects, which are of paramount importance to a persistent system. For example, in a file system there is little control over the size of buffers used in the I/O system, which makes it hard to tune performance. In most cases, the application must be manually tuned to the operating system rather than the operating system being compliant with the application. This is due to the black box nature of the operating system and the policies it implements. Furthermore, the complexity of the operating system buffering and caching policies makes it hard to implement application resilience efficiently.

Memory-mapped files provide convenient mechanisms for accessing objects within files. However, if the persistent system is to be made resilient, objects cannot destructively overwrite their original versions on disk. Therefore, in the majority of systems that employ memory-mapping, the implementer must take convoluted

measures to ensure that modified data does not overwrite stable versions at inappropriate times, which may make recovery impossible.

Like the problems encountered with address translation, the majority of problems when using conventional operating systems as platforms to support stability and resilience revolve around the unsuitability of the abstractions provided by the operating system.

### **Concurrent computation**

Modern operating systems usually provide two different mechanisms for concurrent computation: *heavyweight processes* and *threading*. Most operating systems (including VMS and Unix and its derivatives) support heavyweight processes, each running in its own address space and associated with a particular user. The main problem with using the heavyweight process model to implement persistence is the complexity of sharing data. In most persistent systems, the high level model is a single persistent store shared by multiple concurrent processes. Thus the persistent system implementer is forced to choose between the use of processes, which make sharing difficult, or threads which lack protection. We explore these issues further in this subsection.

Conventional systems such as Unix [51] implement two levels of protection for data: *file-level* protection using degenerate access control lists, and *process-level* protection, which protects ephemeral data inside a process. There are two problems associated with the use of files in persistent systems: atomicity and granularity. Many file systems, notably Unix, offer poor concurrency control over access to files. The control that is provided requires the use of system calls which, like all traps, are relatively slow. There is often a granularity mismatch between language level objects which are generally small and the block size used to hold files. Even if objects were mapped to files, such an approach would be inefficient due to the relatively high amounts of file system meta information that must be maintained. Furthermore, the number of files supported by a file system is often limited thus restricting the maximum number of objects.

Process level protection mechanisms have three problems associated with them. Firstly, the granularity is often inappropriate, usually being constrained to the size of a page frame; one notable exception being IBM's AS400 operating system. Secondly, as discussed above, protection provided at page granularity is often inefficient due to the cost of handling page faults. Lastly, there is often a need to associate different access privileges with different concurrent tasks within a persistent environment (for example to implement data ownership) however, most operating systems do not support this.

Recent operating systems have added support for threads that operate within a single shared address space. There are two varieties of threading systems: threads implemented by the kernel (such as those supported by

NT) and threads provided by user level libraries (such as those supported by Solaris). Both of these threading models are more appropriate for the implementation of persistence than conventional process models. However, threads running in the same address space usually share the same protection attributes. It is therefore impossible to provide adequate inter-thread protection (An exception to this are the threads supported by Linux which are represented as independent processes by the kernel). In particular, it is possible for one thread to accidentally (or deliberately) corrupt the data of another. This problem has been attacked by the Single Address Space Operating System (SASOS) community [14] who have advocated that addressing and protection should be orthogonal to each other.

Perhaps the most serious problem associated with the concurrency model supported by existing operating systems is that, unlike the file system, there is no notion of persistence. Process state is maintained in volatile memory and in the event of system shutdown or failure, this state is lost. In order to implement any form of persistence, the programmer must provide explicit code to checkpoint the state of each process and to restore it. This problem of the non-persistence of processes extends itself to login/logout time and results in many operating systems having a proliferation of *ad hoc* start-up files (e.g. login.com, .login, autoexec.bat, etc.) which effectively rebuild the process environment each time the machine is booted or a user logs on to the system. It should also be noted that these start-up files only rebuild a statically defined environment – they do not recreate the dynamic environment of the user at the time of the last logout<sup>3</sup>.

Just as it is possible to engineer persistent data, it is also possible to implement persistent processes. However, this is fraught with difficulty for three reasons. Firstly, if the process state is to be saved so that it may be restarted, all of the relevant information about the process must be discovered such as the position and state of the heap and stacks. These problems are surmountable and some Unix process checkpointing systems have been constructed, some by the persistence community and some outside of it.

Secondly, if a process engages in communication with other processes and is made persistent, one would like to be able to re-instantiate the process at a later time and transparently re-establish that communication. Furthermore, one would like the re-instantiated process to be assigned the same name it had before termination. Such facilities are provided in systems such as Orbix [7] [2], which supports the logical naming, if not persistence of processes. However, this facility is not provided in conventional operating systems.

---

<sup>3</sup> Some modern desktop environments provide advanced session management tools that attempt to capture the state of a user's desktop so that it can be restored on login. This goes some way towards recreating the dynamic process environment, but it is by no means entirely adequate.

A final problem relates to the persistence of process meta-data. Many operating systems maintain private meta-data associated with each process in kernel data structures. Examples of such data include file descriptors, scheduling information, synchronisation state variables, communications buffers, and message queues. When the state of a process is made persistent without the cooperation of the kernel, it is impossible to recover this meta-data and therefore the saved state of the process is incomplete. This adds considerable (avoidable) complexity to the task of making processes persistent. In addition, the coupling between the logical process and the kernel meta-data inhibits the migration of processes between machines and has been discussed elsewhere [54] [20].

### 3. Operating Systems Attempting to Provide Support for Persistence

The need for operating systems to support persistence has arisen because the construction of persistent application systems on top of conventional operating systems such as Unix, Mach, or Windows NT is prone to inefficiency [25]. There are numerous examples of operating systems that support persistence in one form or another. Examples include *MONADS* [53] [35] [52], *Eumul/L3* [37], *Clouds* [18], *Choices* [11], *KeyKOS* [30], and *Grasshopper* [20]. Each of these persistent operating systems attempts to address some or all of the issues described above. This Section investigates several of these operating systems. In each case, the abstractions relevant to the implementation of persistent systems are described.

#### **Monads**

The MONADS project [53] [35] [52] began at Monash University in 1976. The aim of the project was to provide an environment for software engineering using data encapsulation and information hiding. These features were supported by a custom-built hardware architecture, the main features of which were a capability-based addressing scheme, large virtual addresses, and an inter-address space procedure-calling mechanism.

The MONADS architecture comprises two systems named MONADS-PC [53] and MONADS-MM [56]. The first of these machines supports 60-bit virtual addresses whilst the second supports 128-bit virtual addresses. In both systems, virtual addresses are divided into two parts, an *address space number* and an *offset* within an address space. The address space number specifies a particular address space that is used to hold related data such as the code and data of a module or a stack. When an address space is created, it is assigned a unique number that is never reused. Address spaces are paged to and from secondary storage by the kernel; the MONADS-PC uses a 4 KB page size whilst the MONADS-MM provides both 4 KB and 64 MB page sizes.

Address spaces may be divided into a number of *segments*, which are orthogonal to the paging mechanism. Segments are defined by capabilities held within *segment lists*. Each entry in a segment list specifies an address space number, a base and limit defining the extent of the segment within the address space, and a set of access rights. The capabilities in a segment list define the visible regions within an address space.

The addressing environment of a process is defined by a table containing entries known as *bases*. Each such entry points at a segment list thereby granting the process access to the segments defined within it. Therefore, at any point in time, a process may address any of the segments defined by the segment lists in its current table of bases.

MONADS supports the notion of persistence via the segment abstraction which provides support for large-grained persistent objects. In this regard, the virtual memory environment of MONADS is much like that of Multics [49]. Segments are implemented by the innermost core (the kernel) of the MONADS operating system which runs on top of the bare hardware. The persistence mechanisms are integrated with the operation of the virtual memory enabling the contents of secondary storage to be used for recovery purposes. The upper layers of the operating system, which implement the process and module abstractions, use segments to store internal data structures related to their implementation. Hence, processes and modules are both persistent. The stability of segments is provided through the virtual memory paging mechanisms in conjunction with a checkpoint mechanism. In the absence of failure, this enables the system to be restarted using the consistent state produced at the last checkpoint.

## **Clouds**

The Clouds system [18, 19] was developed at the Georgia Institute of Technology in the mid 1980's. The first version of Clouds ran on VAX hardware and consisted of a monolithic kernel written in C. This version was subsequently abandoned due to its complexity and replaced with an improved design which benefited from the experience gained during the first attempt. The second version of Clouds is based on a micro-kernel called *Ra* written in C++ and running on Sun 3/60s. The basic abstractions provided by Clouds are *objects*, which abstract over storage, and *threads*, which abstract over computation. All data, programs, devices, and resources are encapsulated in objects, which are purely passive entities. Activity is provided by threads, which execute within objects.

A Clouds object is a persistent virtual address space which is not tied to any form of computation and is best suited for the storage of large-grained data and programs. The contents of an object are protected such that stored data may only be accessed by code within the object. Each object may provide a procedural interface via a

set of entry points at which threads may commence execution. Code accessible through such an entry point is known as an *operation*. Each object has a global system-level name which is a unique bit-string that provides a location independent form of naming. User-level symbolic names are mapped onto system names using a nameserver. Each Clouds object is implemented as a single-level store that is demand-paged from a backing store. In this respect, objects are similar to the concept of segments in Multics and MONADS. The backing store for objects is provided by an abstraction called a *partition*, which is responsible for managing the paging of segments to and from secondary storage.

Threads are the only form of computation in Clouds and, unlike objects, are not persistent. They execute within the context of an object and may manipulate the data within it. However, threads may also move between objects by *invoking* an entry point of another object. The invocation mechanism is much like a procedure call except that the addressing environment of the thread is switched on call and return such that it may only access code and data within the current object. This is similar to the mechanism provided by MONADS and later Grasshopper.

The Clouds object/thread paradigm permits multiple threads to execute concurrently within a particular object. To ensure the integrity of the system, the invocation mechanism is augmented with a consistency control system [15] that provides semantics similar to nested transactions [44]. It operates by labelling each thread and entry point with a particular "flavour" of consistency. The supported types are *Global Consistency Preserving* (GCP), *Local Consistency Preserving* (LCP), and *Standard* (S). In simple terms, when a thread invokes a particular entry point, the thread's label changes to that of the entry point and on return, it reverts to its previous value. When a thread's label transforms in this manner, it has implications for consistency control which depend on the label of the entry point.

To recap, Clouds supports coarse-grain persistent objects as a basic abstraction. Stability of objects is provided via the virtual memory system in conjunction with the partition abstraction. This stability mechanism is similar to that used by MONADS so it is likely that Clouds experiences the same difficulties with resilience of objects. The persistence model in Clouds does not extend to threads, which must be restarted after failure. Protection is provided by a capability mechanism, which controls access to various aspects of the system such as the invocation mechanism.

### Eumel/L3

Eumel, and its successor L3 are part of a long running project begun in 1979 at the GMD in Germany. Unlike the previous two systems examined, support for orthogonal persistence was an explicit aim of Eumel/L3. As stated in [37]:

*"We did not find any conceptual reason for anything not to be persistent. Obviously all things should live for as long as they are needed; and equally obviously turning off the power should neither be connected with the lifetime of a file nor with the lifetime of a local variable on a program stack."*

In Eumel/L3 a computation or task is comprised of a virtual address space and at least one thread that executes within that address space. Data is kept in *data spaces* that must be mapped into an address space to be accessed. Communication is achieved by synchronous message passing between tasks. The sending thread waits in the kernel until a message is received. The chief advantage of this approach is that no message buffering is required. As a result of this simple design the communication mechanism is very fast. However, a consequence of this design decision is that the kernel must be, at least partially, responsible for scheduling and synchronisation and must understand the layout of address spaces in order to permit the data transfer to occur.

Eumel/L3 employs a global checkpointing regime. Periodically a snapshot, or *fixpoint*, of the entire state of the machine is taken which includes the state of all data spaces and threads. The *fixpoint* mechanism is resilient so after a crash, programmers need never perform any data recovery nor restart computations. The system restarts from the most recent fixpoint.

To alleviate the overhead associated with taking a fixpoint, a concurrent checkpoint scheme similar to [28] is used. The implementers claim that there is virtually no interruption to on-going computations but only if about 30% extra physical memory is provided. This memory is used to accommodate copies of pages that are modified before they are written to disk. Of course, being a persistent operating system, Eumel/L3 has no file system, eliminating the need for file buffers and so the extent of this problem may be overstated. Nevertheless, performance might be improved by employing a finer grain checkpointing mechanism. A process-oriented scheme such as that used in CASPER [62] or the scheme proposed by Jalili and Henskens [34] would reduce the need for extra memory by eliminating the need for global checkpoints.

A drawback of this approach is that application programs have no knowledge of shutdowns or restarts. Consequently they cannot adjust their behaviour to take these events into account. Whilst this is acceptable and even desirable in isolation, such a scheme can cause difficulties in a networked environment. For example,

since arbitrary time may have passed between restarts, machines may have shut down or application level network protocols may have timed out.

## **Grasshopper**

Grasshopper is an operating system explicitly designed to support orthogonal persistence [25] [54]. To this end, it provides three abstractions, *containers*, *loci* and *capabilities*, all of which are inherently persistent. *Containers* are the only storage abstraction provided by Grasshopper. They are conceptually very large address spaces capable of holding persistent data and are therefore suitable as coarse-grained storage repositories for fine-grained persistent objects. The data stored within a container may be directly referenced by a computation using an address relative to the start of the container. In contrast to address spaces in conventional operating systems, which are inextricably part of the *process* abstraction, containers can exist independently of computation. Thus, at certain times a container may be active and support multiple concurrent computations, while at other times it may be purely passive and void of activity.

The *locus* is the only abstraction over execution. Loci are scheduled by the kernel and execute within separate address spaces. The address space of a locus is composed from views of contiguous regions of various containers called *mappings*. In the simplest case, the address space of a locus contains a single mapping allowing it to access the contents of a particular container known as its *host*. Although each locus executes within a separate address space, a single container may host many loci allowing them to share the code and data stored within. A locus is not bound to any particular container and may move to a new host container via the *invocation* mechanism. Thus, containers and loci are completely orthogonal abstractions.

The Grasshopper kernel must be able to control access to abstractions such as containers and loci. Furthermore, a naming mechanism is required to identify a particular container or locus to the kernel when performing system calls. Grasshopper provides the *capability* abstraction for this purpose [21]. A capability is conceptually a typed reference to an instance of a kernel abstraction combined with a set of access rights. The rights determine the operations that the holder of the capability may perform on the referend. Capabilities may be held by both containers and loci and are stored in a list associated with the holder. Capability lists are maintained within the kernel to prevent forgery.

*Mapping* is a viewing mechanism used to compose address spaces from regions of other address spaces. In particular, mapping allows a contiguous region from one address space to be viewed from within a region of the same size in another address space. Changes made to data visible in either region are immediately reflected in both regions. Grasshopper provides two forms of mapping, *container mapping* and *locus mapping*. Container

mapping allows the address space of a container to be composed from the address spaces of other containers. Since the address space of a locus is composed of regions mapped from the address spaces of various containers, any mappings affecting these containers will also be visible to the locus. For this reason, the effect of container mappings is said to be *global*. In contrast, locus mapping allows loci to install *private* mappings to container regions within their address spaces. These are typically used to provide access to per-locus data structures such as stacks.

From an external view of the Grasshopper system, it appears as if all data is stored within containers. In reality however, the actual data is maintained by *managers* [22]. A manager is a distinguished container holding code and data to support the transparent movement of data between primary and secondary storage. They are the only component of Grasshopper in which the distinction between long and short-lived data is apparent. To support the implementation of various different store architectures, managers have control over the virtual memory system in a similar manner to external pagers in micro-kernel operating systems such as Mach [3] and Chorus [57].

A final aspect of the Grasshopper system is its consistency model. The kernel performs lazy causal consistency tracking between loci at a page granularity [23]. This is achieved by the kernel tracking page faults and maintaining inter-locus dependency information. When a locus makes a resilient copy of its state, known as a *snapshot*, a dependency matrix describing its dependencies on other loci is recorded on non-volatile storage by the kernel. Using this information at recovery time or from transient data structures, the kernel can always determine the latest consistent global state. It is this state that is restored in the event of a failure or shutdown.

Whilst meeting its original design goals, like the other systems described in this section, Grasshopper suffered from a number of problems. These are examined in more detail in Section 4, which uses Grasshopper as a case study.

## 4. Problems with Operating System Support for Persistence

Being designers and implementers of Grasshopper the authors have a much more detailed understanding of its good and bad points. For this reason, we will use Grasshopper as a case study to throw light on the problems of providing direct support for persistence in the operating system. It should be stressed that we do not believe Grasshopper to be any better or worse than any of the other systems described in Section 3.

One of the design goals of Grasshopper was to permit multiple store architectures to coexist on the same machine without changing the operating system. This was motivated by the desire to permit experimentation with

different storage architectures. To fulfil this requirement, the concept of a *manager* was developed to provide user-level control over the storage of persistent data. Since each container can have its own manager, many different store architectures can co-exist within the same system allowing each application to manage its persistent data in the most appropriate manner.

The problem with managers, and with external pager mechanisms in general, is that moving control over the virtual memory subsystem out of the kernel and into a user-level server adds to the latency with which page faults can be serviced [50]. In the best case when the required page is already resident, it is necessary to call and return across the user-level/kernel boundary *three* times. Each time this occurs, certain registers must be saved and restored and capabilities must be checked to enforce protection. In the worst case when the page must be retrieved from disk, *five* boundary crosses are required. Although the cost of the disk I/O far outweighs the cost of the boundary crosses, it represents a significant overhead that could be better spent running other loci.

Another source of inefficiency is the duplication of information within managers and the kernel. The most notable example of this involves the virtual address translation tables. Within the current implementation of the Grasshopper kernel, this information is effectively stored in three separate locations. Firstly, the information is held within the three-level hierarchical page tables (termed *contexts*) used by the memory management hardware. These contexts are typically sparsely populated making them expensive to maintain on a permanent basis. Therefore, a fixed-size pool of memory is used to cache the most recently used contexts. Since contexts merely cache recently used address translations, a permanent record is maintained by the kernel using *local container descriptors* (LCDs) [39]. An LCD represents a mapping from container addresses to physical addresses and contains an entry for every resident page of the container with which it is associated. Each LCD stores address translations very concisely within an extensible hash table. During the resolution of page faults, managers enter address translations into the appropriate LCDs via a system call and the kernel uses this information to create corresponding context entries when required. It was originally intended that the presence of an LCD entry for a particular page would stop the kernel from notifying the manager to resolve future faults on the same page. However, this policy prevents managers from using page faults to implement transactions. Therefore, the current implementation passes all faults on a page through to the manager, requiring them to track the current set of address translations. Although they can obtain this information using a system call to query the appropriate LCDs, it is more time-efficient if a separate hash table is maintained within the manager.

In order to address some of the protection problems described in Section 2, a Grasshopper container may host many loci simultaneously. When this occurs, the host container forms the basis for the address space of each locus. However, since a locus can privately map regions from other containers into its address space, it is

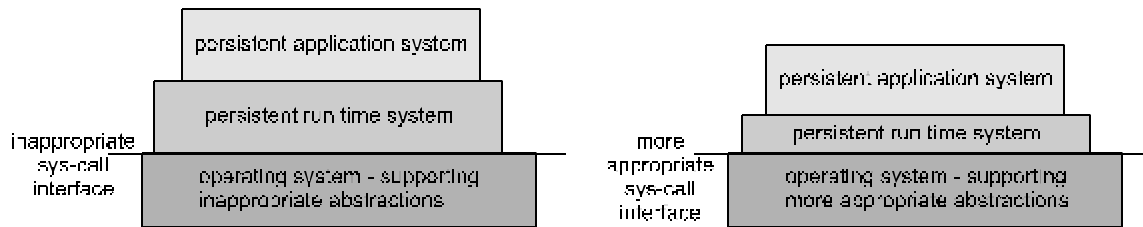
necessary for loci to have separate address spaces. Restating this in another way, Grasshopper does not provide a way for two loci to share the same address space other than by arranging for them to share the same set of mappings. Thus, when context switching from one locus to another, the virtual address space must also be switched to that of the new locus. This is unfortunate since loci were intended to be extremely lightweight processes somewhat akin to threads, although in this implementation they are forced to use a heavyweight context switch much like conventional processes.

The Grasshopper system suffered from several problems associated with locus synchronisation. Frequently these were caused by management conflicts between the kernel and user level. For example, when a locus enters a container for the first time, the kernel must lock part of the container in order to perform a few atomic operations associated with stack creation. However, portions of the container may have been locked by other loci making it impossible for the invocation to take place atomically. This conflict is caused by a combination of inappropriate abstractions and kernel policy decisions.

Whilst the Grasshopper consistency model did not cause any interference at user level, it did impose a causal consistency model on all loci whether they required it or not. It therefore added run-time overhead to all computations running under Grasshopper. Furthermore, it added complexity to all container managers which were required to interact with the consistency protocol and recovery mechanism.

### **Suitability of Abstractions**

All the operating systems examined in Section 3 provide abstractions designed to support persistence. This approach has been motivated by a desire to bridge the semantic gap between the abstractions offered by conventional operating systems and those required by persistent application systems. The difference in these approaches is shown in Figure 1. Figure 1(a) shows the traditional approach to operating system construction where the kernel exports interfaces to the abstractions it implements. These traditionally include files and processes. Implementing persistence using this approach requires the implementer to construct an often complex virtual machine to deliver appropriate abstractions to the persistent application system. If more than one persistent system is constructed on the system, much of this functionality will be duplicated. This has motivated a desire to provide explicit support for persistence at the operating system level. The systems described in Section 3 all follow this approach.



*Figure 1(a):*

*Traditional OS Model*

*Figure 1(b)*

*Persistence oriented OS Model*

In these systems, the abstractions supported by the traditional kernel are replaced by a different set of abstractions, for example, in the case of Clouds the abstractions are objects and threads and in the case of Grasshopper, they are containers, loci and capabilities. Like the abstractions provided by traditional operating systems, the appropriateness or otherwise of these abstractions for the application system being written is irrelevant. The implementer of the persistent application system must work with whatever abstractions are provided by the kernel. This has two consequences, firstly any expense associated with inappropriate abstractions must be borne by the application system. For example, in Grasshopper all applications must pay for the causality tracking mechanisms implemented in the kernel. Secondly, the runtime-system writer must implement the abstractions that are required, possibly using inappropriate building blocks.

In general, the approach of providing direct operating system support for persistence suffers from three identifiable problems:

1. The abstractions supported by the operating system can never be appropriate to all persistent application domains.
2. The operating system interface can never exactly match the requirements of all application systems: this results in inefficiencies and complexities in application systems which are not inherent to the system but are an artefact of implementation.
3. The operating system policy can never match the requirements of all application systems, again generating complexity not inherent in the applications.

If the abstractions provided by an operating system are inappropriate to a persistent application system, it is extremely difficult to implement that system efficiently. This is the situation observed when attempting to implement persistent application systems above conventional operating systems. However, even if appropriate abstractions are provided, an appropriate and fast interface to those abstractions must be provided. For example, in Grasshopper the LCD abstraction is appropriate, but the cost of calling into the operating system makes it

expensive. If a shared memory interface had been provided, the utility of LCDs would be greatly enhanced. Finally, if the policy implemented by the kernel is inappropriate, this increases application complexity. For example, in the CASPER system, the page eviction policy dictated by the Mach kernel increased the system complexity enormously.

For the above reasons we now believe that the operating system should not provide abstractions to support persistence. Instead, it should provide basic abstractions and policy-neutral building blocks from which efficient application systems may be constructed. An alternative operating system architecture is shown in Figure 2 below in which the abstraction-oriented kernel-user level interface has been replaced by a much lower level interface. Rather than providing abstractions, the kernel does little more than to multiplex the hardware and provide some very basic protection-oriented building blocks from which high-level protection mechanisms can be built. The abstractions with which applications programmers are familiar are composed from a set of libraries. The application program may choose to be totally dependent on these libraries, thereby isolating itself from the kernel interface, or it may access the kernel directly. Some of the libraries will be concerned with persistence and some with more mundane functions such as maths operations. The goal of the operating system provider is to engineer an interface that takes no stance on any operating system policy to provide the maximum degree of flexibility and efficiency.

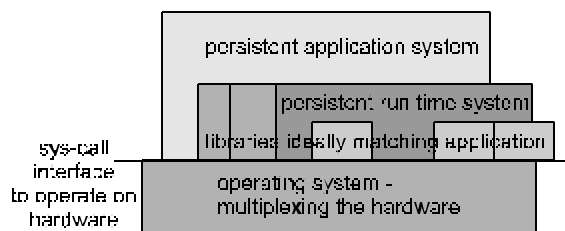


Figure 2: Desirable situation

Some may ask if the operating system kernel is to be minimal, why bother having one at all. There are two reasons: multiplexing and protection. If a machine is to be shared by more than one application system, some central mechanisms need to be provided to permit sharing of resources. The second reason is protection. If multiple application systems exist on a platform and they need to be protected from each other, a kernel is required. Without either of these requirements, the application could be linked with the operating system kernel to form a single executable image containing no interfaces.

The goal of providing a flexible set of policy free building blocks from which high-level mechanisms can be built does not shed any light on what the building blocks should be. However, this question has been the

subject of much recent research in operating systems. In the next section we examine some of the more influential operating systems architectures.

## 5. Current Approaches to Operating System Design

Over the past few decades, a variety of approaches have been used in the construction of operating systems. The modern trend is towards smaller kernels in which the application programmer is provided with ever-increasing levels of flexibility. This has been brought about by the separation of mechanism and policy wherever possible and by providing simple abstractions that mimic the functionality of the hardware rather than high-level abstractions that are overly general. In this section we examine a number of operating system architectures and discuss their relative merits. From this review, we have drawn a number of conclusions on which we have based the design of our new kernel architecture.

### **Monolithic Architectures**

Many regard the *monolithic* kernel design, typified by OS360, VMS, VME and Unix systems, to be the progenitor of modern operating system architecture. Its basic premise is to abstract over the machine hardware to provide high-level abstractions. Monolithic architectures have three different classes of problems associated with them.

Firstly, as we have demonstrated above, the high-level abstractions provided by monolithic systems, such as Unix, are inappropriate for the construction of persistent systems. This is evidenced by the sheer size of the persistent systems (for example Napier88 [43], PS-algol [1] or Pjama [6]) that have been implemented using them. Much of the complexity of these systems is due to the engineering of idealised virtual machines above inappropriate platforms.

Secondly, the evolution of monolithic kernels is extremely difficult due to the combination of poor internal structuring and monolithic nature. This has two effects. Firstly, it makes the maintenance and development of the kernel difficult due to numerous complex interactions between internal modules. Secondly, it makes the field enhancement of kernels that are in use extremely complex. This can be seen in the complexity of utilities for installing patches such as drivers into Unix kernels.

Thirdly, since all of the kernel modules typically reside within the same address space, a bug in one of the modules can bring the entire system down.

By contrast a system structured from separate components may be more easily maintained by developers, extended in the field and exhibits better resiliency characteristics than monolithic designs.

## Micro-Kernel Architectures

The *micro-kernel* architecture was devised to solve some of the problems inherent with monolithic systems. The rationale behind its design is to separate mechanism from policy allowing many policy decisions to be made at user level. This goal is typically realised by building a kernel that supports only a small number of abstractions such as threads, address spaces, and inter-process communication. These abstractions can be composed to build user-level *servers* that implement common services such as file systems, processes, and virtual memory. Examples of micro-kernel architectures include Mach [3], Chorus [57], Amoeba [26, 47], V [16], Choices [11], Psyche [59], L3/L4 [37], and Arena [41].

The most notable effect of the micro-kernel design is to separate the implementation of the system into smaller, more manageable pieces in the form of the kernel and the user-level servers. This arrangement greatly improves the modularity of the system and allows individual modules to be replaced if necessary. Because of the modular design, the reliability of the system is increased since faults are isolated to individual modules. For example, a file system crash would not bring down the entire system as it would in a monolithic architecture. Instead, the file system server can simply be restarted.

Micro-kernels are undoubtedly a step in the right direction towards addressing the problems with monolithic architectures. However, in solving some problems they have introduced others and have yet to demonstrate compelling advantages over monolithic designs. For instance, the introduction of user-level servers to implement high-level policy does not offer significantly more flexibility than monolithic kernels. This is because the server processes are typically crucial to the operation of the system and cannot be replaced without the necessary privileges. In addition, many micro-kernels do not give user-level servers ultimate control over the hardware. For example, in Mach it is possible to build external pagers at user level which are capable of exploiting application specific knowledge. However, certain important decisions such as which pages to replace are made within the kernel. The challenge for OS designers is to provide sufficient flexibility at user-level without compromising the security of the system.

Performance has also proven to be a problem in many micro-kernels due to the increased modularity [50]. Since much of the functionality of systems such as Mach resides outwith the kernel, there is an increase in inter-process communication and interactions with the kernel compared to a monolithic design. As Anderson notes [4], crossing from one protection domain to another is a relatively expensive operation that is not getting faster as the speed of machines increase. Therefore, IPC and system call performance can be a serious bottleneck in many micro-kernels. However, by way of contrast, Liedtke claims that the inefficiency of micro-kernels and their abstractions is largely to do with poor implementation [37]. His own micro-kernels, L3 and L4 in particular, have

demonstrated that it is possible to build extremely efficient thread and IPC mechanisms through clever use of the hardware provided.

### **Customisable Kernels**

A competing approach to the micro-kernel operating system paradigm is the idea of providing a minimal kernel that may be dynamically specialised to safely meet the performance and functionality requirements of applications. The best example of this approach is SPIN [9]. The SPIN system consists of a set of core system services such as memory management and scheduling, and a set of extension services that enable developers to customise the behaviour of the operating system. Customisation is achieved by loading extensions into the kernel where they are integrated with the existing infrastructure. The security of loaded extensions is guaranteed by requiring them to be written in Modula-3 [48]. The strict modularity and strong type system enforced by the language prevents extensions from accessing memory directly or executing privileged instructions unless explicit access is provided through a module interface. Loaded modules execute in kernel mode in response to system events such as exceptions or system calls. All events are declared within Modula-3 interfaces and can be dispatched with the overhead of a procedure call.

The customisable kernel approach suffers from a number of problems. Firstly and most importantly, ensuring the security of the kernel against malicious or erroneous code is difficult. Systems generally use one of three techniques: the use of code digitally signed by a trusted compiler, the use of an interpreter in the kernel, or the inspection of code when it is loaded into the kernel. All of these techniques add some complexity to the kernel since checking must be performed. Another problem with this approach is that it is hard to provide a customisable kernel without having some abstractions with which the extensions interact. To some extent, this approach is at odds with the idea of replacing abstractions with policy-neutral building blocks.

### **Library Operating Systems**

Due to the inadequacy of conventional operating systems, including those based on micro-kernels, a new style of operating system has recently come to light. The goal of these new systems is to allow all policy decisions to be made at user-level. Kernels should contain only those mechanisms that must be implemented securely. The majority of traditional operating system functionality is implemented at user-level and can be changed on a per-application basis. Applications that do not require custom made facilities are simply linked with standard libraries implementing common operating system interfaces such as POSIX. Tailoring the operating system to exploit application-specific knowledge is simply a matter of linking against a different library. The result is a system in which each application may potentially be running a completely different operating system. Since all of the

typical services and abstractions are implemented within user-level libraries, the term *library operating system* has emerged to describe systems of this nature. There have been two independent, but related, approaches to library operating systems: the *exo-kernel* approach and the cache kernel approach. These approaches are described in the following sub-sections.

- **The Exo-Kernel Approach**

The philosophy behind *exo-kernels*, as typified by Aegis [29], is that abstracting over the hardware resources is the wrong approach for an operating system to take. No matter what abstractions an operating system provides, they will always be inappropriate for some class of applications. Therefore, rather than provide a plethora of abstractions, an *exo-kernel* aims to export the hardware resources directly, its only responsibility being the secure multiplexing of resources. For example, the resources exported by Aegis include physical memory (divided into pages), the CPU (divided into time slices), disk storage (divided into blocks), TLB entries, and interrupt/trap events. Thus, in contrast to monolithic and micro-kernels, the interface to an *exo-kernel* is non-portable. However, this is not seen as a problem since portability can be achieved by providing a standard set of libraries that implement portable interfaces such as POSIX. The only difference between this arrangement and that of a monolithic design is that the portability boundary no longer coincides with the kernel interface. The advantage of this is that specialised applications such as database systems, language run-time modules, and garbage collectors can replace the standard interface and work directly with the hardware to achieve optimal performance.

Allowing the functionality of the operating system to be tailored on a per-application basis addresses many of the needs of persistent systems implementers. The only issue is how to expose the hardware resources in a secure and efficient manner to make the provision of such flexibility feasible. In Aegis, security is controlled using *secure bindings* implemented by associating each resource with a password capability [46] that determines the rights of its owner. Every time a resource is used, Aegis checks the rights encoded within the associated capability. Previously allocated resources are reclaimed using a *visible revocation* protocol in which applications are notified of a resource shortfall and may make their own decisions as to which resources to relinquish. To protect itself against recalcitrant applications that either refuse or take too long to comply with requests, Aegis also employs an *abort* protocol in which resources are forcibly reclaimed.

One of the effects of providing such a low-level kernel interface is that the frequency of system calls may be increased. Thus, implementing higher level abstractions such as threads or address spaces may incur the overhead of several system calls per higher level operation. This overhead is not present in monolithic- or micro-kernel-based systems since all of the abstractions requiring access to the hardware are implemented within the

kernel. This problem may be addressed in two ways. As noted in [29] and demonstrated in the SPIN system, the first way is to allow code implementing higher-level operations to be downloaded into the kernel. Using this technique, performance critical operations such as thread scheduling or interrupt handling can execute in supervisor mode and avoid the overhead of system calls. The problem may also be addressed by exposing kernel maintained data structures and thus obviate the need for (some) system calls.

### **The Cache Kernel Approach**

Rather than exporting an interface to the hardware as do exo-kernels, the Cache Kernel [17] supports a small number of primitive abstractions for which all policy decisions are implemented by library operating systems at user-level. The interface to the Cache Kernel supports three types of object: *address spaces*, *threads*, and *descriptors*. Each object is represented by a descriptor that is managed by the application (library operating system) that created it. Descriptors contain the state or meta-data required to realise an object. For example, a thread descriptor includes the saved register values and a stack pointer for use when handling exceptions. Applications may load the descriptor for an object into the kernel where it is cached. During this time, the kernel maintains the descriptor and executes the performance-critical actions supported by the objects. Like most caches, the insertion of a new entry into the cache may displace an existing entry. Displaced object descriptors are returned to the applications that own them.

To illustrate the operation of the Cache Kernel consider how an application might control the scheduling policy of its threads. Applications load thread descriptors into the kernel when their respective scheduling policies determine that the threads are eligible to run. The Cache Kernel will schedule and dispatch the set of cached thread objects in round robin fashion, effectively sharing the CPU resources among the active threads. When a cached thread blocks, its descriptor is unloaded from the kernel and returned to the application that owns it.

According to Cheriton [17], the approach taken by the Cache Kernel offers three benefits. First, the low-level caching interface allows applications to control resource management according to any desired policy. Second, the caching approach prevents applications from exhausting the supply of object descriptors as may occur in conventional operating systems. Finally, the cache model greatly reduces the complexity of the kernel compared to previous micro-kernel designs.

## 6. New Approaches to Operating System Support for Persistence

In Section 3, we argued the inappropriateness of operating system abstractions for persistence. In Section 5, we examined current trends in operating systems. Based on lessons learned in Section 3 and from our observations of the systems described in Section 4, we embarked on the construction of a new operating system architecture. Like its predecessor, Grasshopper, support for persistence was an explicit aim. However, unlike Grasshopper, there are no abstractions that directly support persistence, nor is the kernel persistent.

All persistent systems contain some non-persistent functionality. This functionality is required to bootstrap the persistent system by reinstantiating the necessary persistent data-structures. In the Grasshopper system, the kernel contained a non-persistent core which reinstantiated persistent kernel data structures. Similarly, in a traditional kernel, a non-persistent core is responsible for bootstrapping the persistent data structures that implement the file system abstractions. However, the management of persistent data may be completely moved out of the kernel resulting in a kernel that is totally ephemeral. This has two added benefits, firstly the kernel is much simpler and smaller making it faster and more manageable. Secondly, it lessens the amount of meta-data held in the kernel and therefore reduces the degree of coupling between application level code and the kernel.

The next concern, which guided the design, was the desire to enforce protection domains between different application programs. As stated earlier, if protection is not an issue, there is no reason why application programs should not be directly linked with the kernel. However, if protection is required, the kernel must provide some mechanism for constructing and identifying a protection domain. Furthermore, the protection mechanism provided must be synergistic with the hardware. The desire to keep all policy and management out of the operating system results in the concept of a self-managing protection domain. We therefore espouse the theory that a self-managing protection domain is all that is required as a building block. Consequently, in our new operating system, *Charm* [24], the self-managing protection domain is the only building block supplied to the application programmer.

In order to support self-managing protection domains, the kernel needs to provide some mechanisms with which self-management may be implemented. These include communication mechanisms and mechanisms for providing secure access to the hardware resources supported by the machine.

Communication mechanisms are required so that:

1. self-managing protection domains may communicate with the kernel (for example to securely access some hardware entity),

2. the kernel may communicate with self-managing protection domains to inform them of events such as clock ticks, and,
3. self-managing protection domains may communicate with each other.

Secure access is required to all resources provided by the hardware and is the principle reason for the kernel's existence. The hardware resources multiplexed by the kernel must include: processor time, physical page frames, hardware devices, access to interrupts including device interrupts, clock ticks and protection faults. Depending on the type of hardware, it may also include control over virtual memory. For example, this is necessary on the Pentium implementation of Charm due to the hardware's intimate knowledge of page table layout.

The amount of meta-data in the kernel has been kept to a minimum. The meta-data that is in the kernel is ephemeral and may be viewed from any protection domain. This makes the implementation of both persistent and mobile computations, as discussed in [20], [12] and [55] much simpler.

The Charm kernel does not support threads since thread support is not necessary and would introduce policy into the kernel. It does not provide any high level scheduling models, as once again, this would introduce policy. Instead, it provides a mechanism for multiplexing processor time between protection domains. It also does not provide any direct device support other than allowing access to the I/O space supported by the hardware.

### **Protection Domains**

The basic building block provided by the Charm kernel is a self-managing protection domain called an *arena*. An arena is an ephemeral protection domain which has an associated fixed set of *upcall points*. The upcall points are used to communicate events from the kernel to the arenas. Each upcall point specifies the address of a piece of code that is activated when an event occurs. If the arena has control of the CPU when the event occurs, the operating system transfers control to the upcall point in much the same way as a hardware interrupt is delivered to the operating system. Consequently, the upcall handler is responsible for the preservation of registers, except for one or two scratch registers whose contents are saved in a globally visible kernel data structure. This permits arbitrary concurrency mechanisms, such as thread models, to be safely and efficiently constructed at user level.

The Arena upcall points are specified using system calls. The set of upcall points currently supported by the implementation includes:

1. The activation upcall point, which specifies the piece of code to be called when the arena gets control of the CPU for the first time.
2. The time slice start (TSS) upcall point, which is called whenever the arena subsequently gets control of the CPU.

3. The time slice end (TSE) upcall point, which is called when the current processor allocation ends.
4. A clock tick upcall point, which is used to accept hardware generated clock events.
5. An exception upcall point, which is called when an synchronous exception occurs (including virtual memory exceptions).
6. An interrupt upcall point which is called when asynchronous interrupts occur.
7. An inter-arena call (IAC) upcall point, which is used for inter-arena communication.

### **Managing meta data**

As stated above, in order to reduce coupling between the user level and the kernel, the amount of meta-data in the kernel has been kept to a minimum. To model this, the kernel manages two classes of meta-data: public and private. Public meta-data is made visible to code running in arenas using a system call (*get\_kernel\_meta\_data*). This call returns the address of a public data structure containing a variety of data such as system constants (e.g. the page size), system variables (such as the number of ticks since system start-up), the time slice array (described in Section 6.3), the active arena table, the status of interrupt lines (described in Section 6.4) and a view of physical memory (described in Section 6.5). Making this data visible at user level at low cost (due to shared memory) obviates the need for a large number of system calls. It also makes the management of arenas simpler since all pertinent data structures are publicly visible.

In addition to the public kernel data structures, the kernel also maintains private data structures. These are used to implement security. In the sections below, for ease of description, these are described as being components of other data structures. In reality, the kernel maintains totally separate data structures to ensure protection is maintained. For example, the kernel maintains a separate array of capabilities that protect the time slices described in the next section.

### **Multiplexing the CPU**

Time is treated as a physical resource by slicing one second into a number of discrete intervals known as *time slices*. The number of slices is a kernel configuration parameter<sup>4</sup>. The kernel maintains an array of time slices, each of which contains two fields:

---

<sup>4</sup> Currently the number of time slices is set to 1000, permitting the CPU to be allocated at the resolution of 1 millisecond. We have no current evidence to suggest whether this is too coarse or too fine a granularity.

1. a public field indicating the identity of an arena which will be given the CPU during that time slice,
2. a private field containing a capability which protects the time slice.

A system call (*ts\_bind*) is provided to associate an arena with a time slice. Depending on its needs, an arena may bind itself to a single time slice or a contiguous range of slices. A capability matching the one stored in the slice must be presented in order for the bind operation to succeed. System calls are also provided for setting the capability (*ts\_change\_caps*) and for disassociating an arena from a time slice (*ts\_unbind*). When the system first boots up, all capabilities are set to a known value of NO\_CAP.

The kernel cycles around the time slice array in response to clock interrupts (or ticks). On each tick, the kernel determines the arena that is to be given control of the CPU by examining the corresponding time slice. An example of this is illustrated in Figure 3 in which four arenas are shown. Arena 4 is bound to a contiguous range of 4 time slices and while it has control of the CPU, the operating system delivers *tick* upcalls in response to each clock interrupt. This continues until the last time slice owned by arena 4 is over and at this time, a *time slice end* (TSE) upcall is made by the kernel. This notifies the arena that its allocated time is over and that it should preserve its state and yield control of the CPU. Naturally, this processing takes a small amount of time, shown as *yield time* in the diagram. Once an arena has yielded control of the CPU following a TSE upcall, the kernel consults the time slice array to determine the next arena to be assigned control of the CPU and delivers a *time slice start* (TSS) upcall to that arena. Following a TSS, an arena has complete control of the CPU.

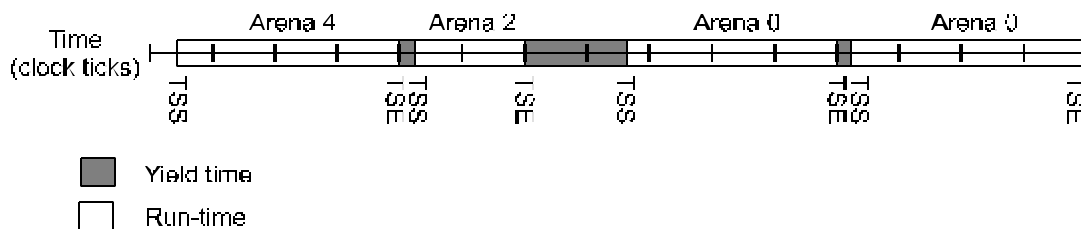


Figure 3: Multiplexing the CPU

Since the kernel does not preempt running arenas, instead informing them that their allocated time is over using a TSE upcall, an arena may fail to yield the processor. To deal with this problem, Charm counts the number of time slices that pass whilst waiting for an arena to yield and this number of slices is deducted from the arena's next allocation of CPU time until the "debt" is repaid. If an unreasonable number of time slices pass before an arena yields, the kernel simply destroys the arena without warning and the slices occupied by it are set back to NO\_CAP.

Thus, the kernel does not preempt arenas other than terminally. This approach makes the arena totally self-managing, even to the extent of having to maintain register sets. The consequence of this is that user level concurrency control is simpler, faster and more efficient. It is easier to write efficient thread code and it makes the creation of mobile and persistent processes much simpler.

We anticipate that most systems implemented using this mechanism will choose to incorporate a user level scheduler to control the allocation of most, if not all, of the time slices. Consequently, such a scheduler would manage the capabilities for each of the time slices. This approach has been followed in several library operating systems such as Nemesis [36] and Arena [41].

### **Managing Devices**

The Charm kernel does not contain any conventional device drivers. All device code must be implemented at user level. Once again, the motivation for this is the separation of policy and mechanism. Most device drivers currently in use contain much policy related to buffering, synchronisation, processor time etc.

In Charm, the kernel maintains an array of interrupt flags visible at user level. When an interrupt occurs (other than a clock interrupt), the corresponding flag is set by the kernel. If the arena running is registered as a handler of that interrupt, the arena is upcalled with an interrupt upcall. Otherwise the running arena continues execution and the handler of the interrupt will service it when it is next scheduled. A similar scheme has been successfully used in Nemesis/Pegasus [36]. The astute reader will notice that this is a form of pre-emption. However, we view this as the kernel *stealing* time from the currently active arena. This also happens when clock ticks occur and may be regarded as operating system overhead. This is the only computation performed by the kernel that is not at the behest of an application program.

There are a number of advantages to this scheme. The most important of these, identified by the Nemesis group [36], is that it avoids devices from swamping the system. In many systems, there is no control over the rate at which interrupts are taken and consequently the kernel accepts requests that the application code can never service in a timely fashion. This regime gives the application level control over the servicing of devices and thus avoids this problem.

Another benefit of this approach is that it does not require applications to be pre-empted on device interrupts. Instead the kernel notes the interrupt in a piece of stub code and returns control to the currently running arena. We believe the lack of pre-emption and consequent control over the saving and restoration of registers to be an important component of the self-managing arena principle.

## **Multiplexing physical memory**

Physical memory is another resource requiring multiplexing between applications. The control of access to physical memory is of paramount importance for security. It is also of great importance for the construction of persistent systems in which providing abstractions over memory is a central tenet. In line with our design goals, control over memory allocation schemes must be performed in a policy free manner. In particular, application systems must have control over the allocation of physical memory in order to a) efficiently implement algorithms and b) interact with hardware devices such as screens.

The Charm kernel treats memory in a similar fashion to time – it models physical memory as a user visible array of records known as the *core map*. Each entry in the array represents a page frame of physical memory and is identified by its location in the array called a *page frame number* (pfn). Each entry in the core map contains three fields:

1. A public bit field containing a various reference counts indicating the number of extant mappings for the page frame.
2. A private field indicating the read capability for the corresponding page frame,
3. A private field indicating the write capability for the corresponding page frame,
4. A private field indicating the management capability for the corresponding page frame.

The read capability must be presented whenever a page is mapped for reading, similarly the write capability when write access is required. The management capability is required in order to change the read or write capability for a physical page. With appropriate use of these capabilities, it is possible for a user level manager to grant controlled access to a page whilst maintaining management of it. The physical pages unused by the kernel have their capabilities set to NO\_CAP at boot time.

## **Managing virtual memory**

Ideally, an arena should be free to represent virtual to physical mappings in any way it sees fit. In support of this, modern RISC architectures such as the Power-PC, Alpha, and MIPS, support a software loadable translation look-aside buffer rather than forcing a particular page table data structure upon the implementer. In such systems, the implementer is free to represent the virtual to physical mappings using whatever data structure best suits the application domain. Many of the library operating systems have implemented facilities that expose the soft-TLB architecture provided by the hardware [8].

However, on the Pentium platform that hosts Charm, the format of page tables is mandated by the architecture and little software control over the TLB is possible. In order to give arenas as much self-autonomy as possible and obviate the need to replicate meta data, each arena has read-only access to the page tables that implement the virtual address space associated with it. As we argued earlier, this eases the integration of persistent and virtual memory abstractions considerably. However, arenas are not permitted to create new page table entries nor change existing entries directly since this would permit users to construct arbitrary mappings which represents a serious security breach. Consequently, the Pentium implementation of Charm provides three system calls to manage page tables. These permit user level code to create level 1 and level 2 page tables and to install virtual to physical mappings. Although we regret the need for system calls to construct page table entries, it is necessitated by a combination of the Pentium architecture and security considerations.

### **Inter-arena calls**

The last mechanism supported by Charm is protected control transfer known as *inter-arena-call* (IAC). Like all mechanisms supported by Charm, IAC is extremely simple and therefore fast. The kernel mediated IAC requires the callee to specify a target arena. The current register set and control of the CPU are transferred to the target arena IAC upcall point.

Should the application level require call and return semantics, these may be engineered by appropriate use of parameters. If more data than can be transported in registers is required, the application code may construct user level protocols by passing page frame numbers with their capabilities in registers. This has been used to construct an efficient CORBA like inter-arena inter-object calling mechanism.

When an IAC occurs, the remainder of the current time-slice is donated to the callee. The donation of time occurs only for the duration of the invocation; bindings in the time-slice array are not changed by the IAC. In many cases, this permits a call and return to occur without intervention from either the time-slice scheduler or any thread level scheduling mechanism provided by the caller or callee. Time slice ends are always delivered to the arena with control of the CPU. Consequently, if a time-slice end occurs during the processing of an IAC, it is sent to the callee arena not the caller. This permits the callee to lazily perform any management tasks that may be necessary due to the new thread of control running within it. If the call and return occurs before a time slice end, these management tasks need not be performed at all. Typically, these tasks include the creation of a new thread for the computation and perhaps informing a user level scheduler of the increase in concurrent computations in the arena allowing it to change its scheduling regime. Of particular interest is an arena that contains no running

threads prior to invocation. This situation is somewhat analogous to processes moving from a blocked to a ready state in a conventional system.

## 7. Implementing Persistent Application Systems Using Charm

Having described Charm, we can now describe how persistent systems may be constructed above the Charm kernel. Since Charm provides a relatively minimal kernel and a policy-neutral set of building blocks, the implementer of a persistent application is free to choose whatever techniques are appropriate to the application. This makes Charm conducive to the construction of persistent application systems. For example, it is possible to efficiently construct a persistent virtual address space, or an object store utilising either hardware or software address translation.

Perhaps, the most difficult issue to resolve in the construction of any persistent system is how to rebind to persistent data following a crash or shutdown. Typically, systems will contain a non-persistent fixed point, which bootstraps the persistent applications. The Charm bootstrap mechanism permits an arbitrary number of binary images to be loaded along with the kernel. When the kernel is bootstrapped, it creates an arena, known as a *bootstrap arena*, for each loaded binary image and binds the arenas into the time slice array. This causes the code in the arenas to be executed. Since the images may contain arbitrary code they may have arbitrary behaviour when executed. These arenas form the fixed point for systems running under Charm.

By way of an example, we now demonstrate how a collection of bootstrap arenas may be used to support persistent processes embodied within arenas. An important observation is that while a process can save its own state, it cannot entirely recreate its own state. Some external agent is required to create the arena, set up an initial virtual addressing environment, inject some code into it and set some entry points. These tasks are the responsibility of the fixed-point known as the *resurrector*, which is implemented by a bootstrap arena. We require one other bootstrap arena: a disk driver which is unremarkable except for the fact that it runs at user level and makes use of the interrupt architecture described in Section 6.4.

The resurrector is responsible for the restoration of persistent processes that have registered with it. It has functionality similar to the daemon in CORBA systems except that it restores the entire state of a process rather than merely restarting a program. We will assume that the process being made persistent requests the resurrector to make it so and that some of the state of the process is on disk (swapped out or on persistent storage) and some is in main memory.

In order to recreate the state of the process, the resurrector must be informed of: the arena's entry points, the root of the virtual addressing environment and the set of physical page frames (and associated capabilities) required to resurrect the process. This information is contained in a data structure termed the *resurrection block*. This is conveyed to the resurrector by the arena performing an IAC into it, passing the page frame number and read capability for the page containing the resurrection block. This process is illustrated using an example in Sections 7.1 and 7.2 below.

### Saving the state of Persistent Processes using Charm

Figure 4a shows a running Charm system, which for simplicity contains three processes: a user level process (which is persistent), the resurrector, and the disk driver. The persistent process manages its own persistence and we assume that it is writing its persistent data to the disk using one of the techniques described in Section 2. Since the operating system imposes no policy on the process, it is free to use whatever techniques the implementer feels is appropriate to the application domain.

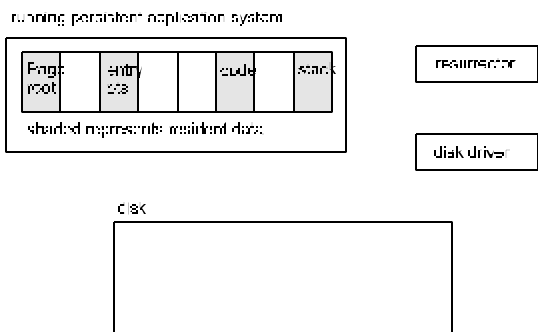


Figure 4a

A running persistent application system

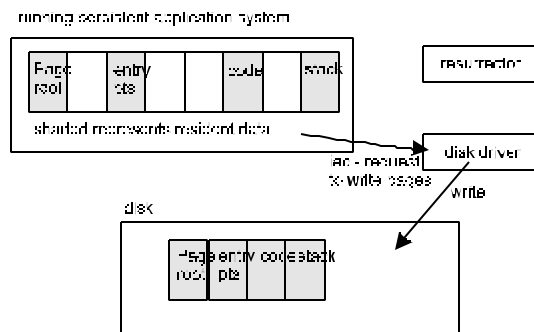


Figure 4b

A checkpointed persistent application system

In Figure 4b, the process has carried out a checkpoint operation, making a non-volatile copy of its state. This is achieved by the process making an IAC to the disk driver. Note that following the checkpoint, the process is still running and need not convey its checkpointed state to the resurrector.

In Figure 4c, the persistent application system has created a resurrection block (shown in dark grey). This contains the non-volatile address of the root of the virtual addressing environment and the non-volatile addresses of the set of pages written to disk in Figure 4b. The resurrection block also contains the virtual addresses of the entry points, without which, the arena cannot be resurrected to a self-managing state. In this example, the set of pages referenced by the resurrection block are coincidentally those active at the time of registration with the resurrector. In many cases, it is possible to statically identify a smaller set of pages, which

may be used to recover an arena. For example, in the Grasshopper system this was achieved using a single page containing carefully crafted recovery code along with the entry point(s). However, the choice of the set of pages contained in the resurrection block is at the discretion of the arena implementer.

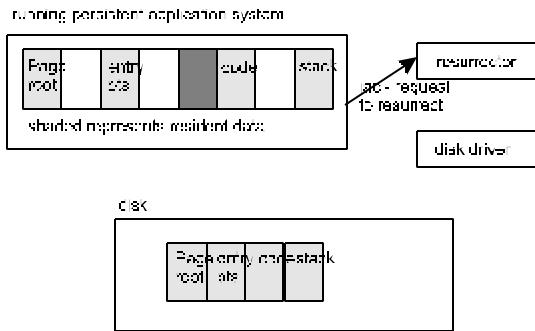


Figure 4c

Requesting the resurrector to checkpoint

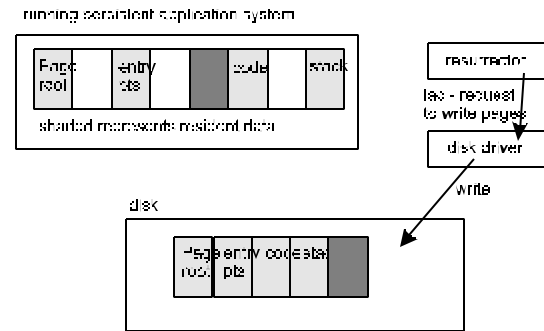


Figure 4d

A recoverable persistent application system

In Figure 4d, the resurrector writes the recovery block to non-volatile storage. The way in which the resurrector performs this is at its own discretion. Since the resurrector is a bootstrap arena, it must either explicitly write the data, or be capable of recreating its own persistent state. The latter approach was used in Grasshopper self-managing managers [38]. However, this is resurrector policy and is at the discretion of the resurrector implementer. Following the writing of the resurrection block, the process state is totally recoverable and may consequently be recreated following a crash or shutdown.

### Recovering the state of Persistent Processes using Charm

On bootstrap, the Charm kernel creates arenas for, and starts executing, the bootstrap arenas. In our example, the resurrector and the disk driver will automatically be started. As shown in Figure 4e, the first task of the resurrector is to recover the resurrection blocks describing the processes requiring resurrection. Next, the resurrector recovers the pages necessary to recreate the self-managing arena. This is achieved by reading the necessary pages into pages which it manages as shown in Figure 4f. At this point in time, all the information necessary to (re)create the self managing arena has been recovered from non-volatile storage.

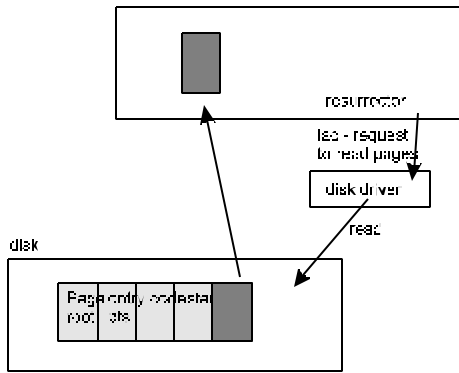


Figure 4e

Following a shutdown

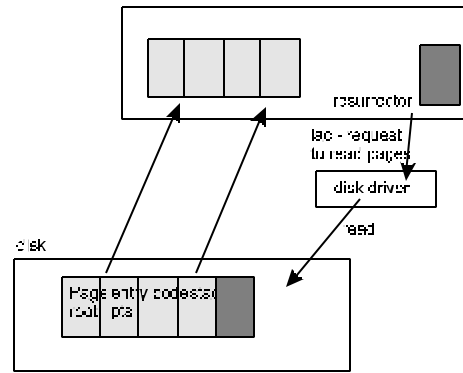


Figure 4f

The resurrection process in progress

Next, as shown in Figure 4g, the resurrector creates a new arena which is an exact clone of the process running in Figure 4a. Firstly, this requires the entry points to be set using the values extracted from the resurrection block. Next, the virtual address space must be set up. This is possible since the resurrector knows which pages are resident (since it has just created them) and knows where the root page table is. The capabilities associated with each page must be set so that the arena can manage them once it is operational. The simplest approach is for the resurrector to set the capabilities to the values contained in the resurrection block. Finally, the newly created arena must be allocated some time-slices to permit it to execute. The resurrector will commonly achieve this by interacting with a user level scheduler. Alternatively, in this simple example, the resurrector could donate some of its own time-slices to the newly created arena.

Finally, as shown in Figure 4h, the arena needs to be informed of its resurrection so that it may perform any application-oriented recovery that is required. This is shown by the resurrector performing an IAC into the newly established arena. Note that this is not strictly necessary since the arena will be sent an activate upcall when it is first given control of the CPU.

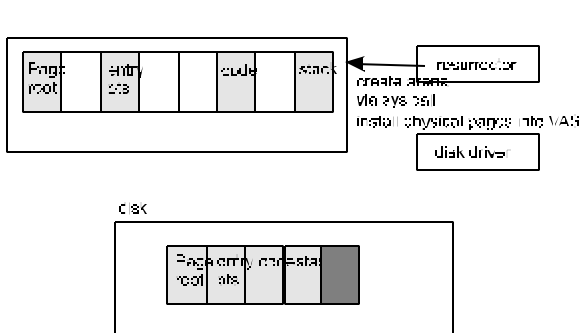


Figure 4g

Recreating the arena

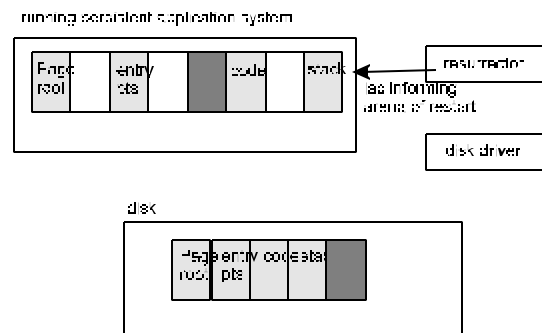


Figure 4h

The completed resurrection process

This example shows how a persistent process may be implemented using the building blocks provided by the Charm architecture. This scheme is only one of many ways in which persistent architectures may be constructed. It is possible to construct much more sophisticated architectures than the one described here. In particular, if the persistent processes and the resurrector are more tightly coupled, for example by sharing knowledge of the format used to store snapshots on disk, a more sophisticated recovery regime may be used. To illustrate, if the disk driver were replaced by a more sophisticated component such as the log-structured storage manager used in Grasshopper [32] and Lumberjack [33], a more sophisticated recovery scheme could be employed. This represents the introduction of policies and conventions at the architectural layer implemented above the kernel. How best to do this is a separate area of research.

## 8. Conclusions

In this paper we have examined operating system support for the construction of persistent systems and observe that inappropriate operating system abstractions have hindered their development since their inception. That conventional operating system abstractions are inappropriate for the construction of persistent systems and database systems has been known to many for a considerable period of time [61] [13]. The major hurdles encountered are:

1. That no widely used operating system provides sufficiently flexible mechanisms to enable the hardware facilities to be exploited to their full potential in order to manage persistent objects and their inter-relationships.
2. The abstractions provided with which to construct resilience mechanisms do not provide enough control over when and where data is written to persistent storage nor over the buffering or caching policies that are implemented. Thus they are unsuitable for the construction of persistent systems.
3. The mechanisms for the construction of independent concurrent activity presume that processes are not persistent. The construction of persistent processes is hindered by inaccessible process meta-data, process naming mechanisms and inappropriate protection mechanisms.

The inappropriateness of these mechanisms and abstractions have led some researchers (including ourselves) to construct operating systems whose abstractions were (intended to be) ideally suited to the construction of

persistent application systems. However, experience gained from implementing and experimenting with these systems has shown that the abstractions provided by these operating systems also suffer from deficiencies. In particular, there is often a problem impedance mismatch between the abstractions provided by the operating system and those required by the persistent application system.

A new thrust in operating systems has been the movement towards operating systems that do not provide any abstractions. Instead, these systems do little more than multiplex the hardware and provide some basic functionality from which security mechanisms may be constructed. Amongst their many appeals are: the ability to construct application systems in whatever manner is appropriate, low kernel/user-level coupling, ease of constructing synchronisation mechanisms and the ability to implement in an environment that is free of operating system imposed policy. In this paper we have described a new operating system, called Charm, which embraces this trend in an attempt to construct an ideal platform on which persistent application systems may be constructed.

The Charm operating system addresses many of the problems highlighted in this paper and attempts to offer an ideal platform for the construction of persistent systems. The Charm kernel does not provide any thread or process abstraction. Instead, self-managing addressing environments are provided from which arbitrary concurrency models may be constructed. This makes user level concurrency control mechanisms both easier to write and more efficient thus making the creation of mobile and persistent processes simpler.

The sharing and protection of data within these addressing environments are under the complete control of the application programmer. In particular, page faults are efficiently vectored to the self-managing protection domains permitting access violations to be handled efficiently. This considerably reduces the boundary crossing problems associated with both external pagers and Grasshopper managers. Virtual address space management is further improved by permitting secure user-level access to the page tables. This has the dual benefit of reducing the amount of duplicated meta-data and removes the need for a considerable number of system calls.

Since all device drivers are implemented at user level it is possible to write device drivers whose policies are suited to the application systems being executed. This is in stark contrast to conventional systems whose device drivers contain much (fixed) policy related to buffering, synchronisation and caching. Similarly the system provides much flexibility over scheduling regimes and how interrupts are processed.

The Charm kernel has been implemented and runs on the Pentium architecture. It consists of 5000 lines of C++ and 800 lines of assembler and compiles into an executable image 90KB in size. We are currently implementing various user level device managers, experimenting with concurrency models and implementing various persistence models above the kernel. We hope to be able to report on these in the near future.

## 9. Acknowledgements

This work is partially supported by ESRC Integrated Operating System Support for Large Heterogeneous Archives Project HS 519 25 5043 which is part of the Analysis of Large and Complex Datasets initiative and by EPSRC grant number GR/M78403 under the Distributed Information Management initiative. It benefits from discussions with colleagues in Esprit Project 22552 – Pastel (Persistent Application Systems, Technologies, Environments and Languages). The paper benefits from comments made by Prof. Ron Morrison who “liked it, but...” and from *anonymous* comments made by Prof. Richard Connor. The paper benefits from the proof reading by Heather Brennan for which we are grateful.

## 10. REFERENCES

- [1] "PS-algol Reference Manual - fourth edition", *Persistent Programming Research Report 12/88*, University of Glasgow and St Andrews, 1988.
- [2] "The Orbix Architecture", *Iona Technologies Ltd*, <http://www.iona.com>, 1993-9.
- [3] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, pp. 93-112, 1986.
- [4] T. Anderson, E. Lazowska, and H. Levy, "User-Level Interprocess Communication for Shared Memory Multiprocessors", *ACM Transactions on Computer Systems*, **9**(2), pp. 175-198, 1991.
- [5] M.P. Atkinson "Programming Languages and Databases", in *Proceedings of Fourth IEEE International Conference on Very Large Databases*, pp. 408 - 419, 1978.
- [6] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence, "An Orthogonally Persistent Java", *ACM Sigmod Record*, **25**(4), 1996.
- [7] S. Baker, "CORBA Distributed Objects Using Orbix", Addison-Wesley, Harlow, England, 1997.
- [8] K. Bala, M.F. Kaashoek, and W.E. Weihl "Software Prefetching and Caching for Translation Lookaside Buffers", in *Proceedings of First Symposium on Operating System Design and Implementation (OSDI)*, Monterey, California, 1994.
- [9] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility, Safety and Performance in the SPIN Operating System", in *Proceedings of Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP 15)*, December 1995.
- [10] A.L. Brown, "Persistent Object Stores", *Ph.D*, University of St. Andrews, <http://www-fide.dcs.st-and.ac.uk/Info/Papers4.html#thesis.ab>, 1988.
- [11] R.H. Campbell, G.M. Johnston, and V.F. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)", *ACM Operating Systems Review*, **21**(3), pp. 9-17, 1987.
- [12] L. Cardelli and A.D. Gordon, "Mobile Ambients", *Lecture Notes in Computer Science*, **1378**, pp. 140-155, <http://www.luca.demon.co.uk/Bibliography.html#Abstractions for Mobile Computation>, 1998.
- [13] D. Chamberlin, M.M. Astrahan, *et al.*, "A History and Evaluation of System R", *Communications of the ACM*, **24**(10), pp. 632-646, 1981.
- [14] J. Chase, H. Levy, M.J. Feeley, and E.D. Lazowska, "Sharing and Protection in a Single Address Space Operating System", *Transactions on Computer Systems*, **12**(2), 1994.
- [15] R. Chen and P. Dasgupta. "Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation", in *Proceedings of Proceedings of the Ninth International Conference on Distributed Computing Systems*, pp. 1-17, 1989.
- [16] D.R. Cheriton, "The V Kernel: A Software Base for Distributed Systems", *Software*, **1**(2), pp. 9-42, 1984.
- [17] D.R. Cheriton and K.J. Duda. "A Caching Model of Operating System Kernel Functionality", in *Proceedings of the First Symposium on Operating System Design and Implementation*, Monterey, California, pp. 179-194, 1994.
- [18] P. Dasgupta, R. Chen, *et al.*, "The Design and Implementation of the Clouds Distributed Operating System" 88/25, Georgia Institute of Technology, 1988.
- [19] P. Dasgupta, R. LeBlanc, A. Mustaque, and R. Umakishore, "The Clouds Distributed Operating System", *Technical Report 88/25*, Arizona State University, 1988.

- [20] A. Dearle, "Towards Ubiquitous Environments for Mobile Users", *Internet Computing*, 2(1), pp. 22-32, <ftp://jeroboam.cs.stir.ac.uk/pub/papers/mobileic.pdf>, 1998.
- [21] A. Dearle, R. di Bona, J. Farrow, F. Henskens, D. Hulse, A. Lindström, S. Norris, J. Rosenberg, and F. Vaughan. "Protection in the Grasshopper Operating System", in *Proceedings of Proceedings of the 6th International Workshop on Persistent Object Systems*, pp. 60-78, <ftp://persistence.cs.stir.ac.uk/pub/papers/GH-04.ps.Z>, 1994.
- [22] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan, "Grasshopper: An Orthogonally Persistent Operating System", *Computer Systems*, **Summer**, pp. 289-312, 1994.
- [23] A. Dearle and D. Hulse. "On Page-based Optimistic Process Checkpointing", in *Proceedings of IWOOOS '95*, Lund, Sweden, pp. 24-32, 1995.
- [24] A. Dearle and D. Hulse, "The Charm Operating System Web Pages", <http://persistence.cs.stir.ac.uk/Charm/>, 1999.
- [25] A. Dearle, J. Rosenberg, F.A. Henskens, F.A. Vaughan, and K.J. Maciunas. "An Examination of Operating System Support for Persistent Object Systems", in *Proceedings of 25th Hawaii International Conference on System Sciences*, Poipu Beach, Kauaii, pp. 779-789, 1992.
- [26] F. Douglass, J. Ousterhout, M.F. Kaashoek, and A. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite", *Computing Systems*, 4(4), pp. 352-385, 1991.
- [27] K. Elhardt and R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems", *Transactions on Database Systems*, 9(4), pp. 503-525, 1984.
- [28] E. Elnozahy, D. Johnson, and W. Zwaenepoel. "The Performance of Consistent Checkpointing", in *Proceedings of 11th Symposium on Reliable Distributed Systems*, pp. 39-47, 1992.
- [29] D.R. Engler, M.F. Kaashoek, and J.W.O.T. Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Management", in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, Colorado, pp. 251-266, 1995.
- [30] N. Hardy, "The Keykos Architecture", *Operating Systems Review*, **September**, pp. 1-8, 1992.
- [31] D. Hulse and A. Dearle, "A Flexible Persistent Architecture Permitting Trade-off Between Snapshot and Recovery Times", *Technical Report GH-16*, University of Stirling, Stirling, <ftp://persistence.cs.stir.ac.uk/pub/papers/GH-16.ps.Z>, 1996.
- [32] D. Hulse and A. Dearle. "A log-structured persistent store", in *Proceedings of 19th Australasian Computer Science Conference*, pp. 563-572, 1996.
- [33] D. Hulse and A. Dearle. "Lumberjack: A Log-Structured Persistent Store", in *Proceedings of Eighth International Workshop on Persistent Object Systems (POS-8)*, Tiburon, California, pp. 187-198, 1998.
- [34] R. Jalili and F.A. Henskens. "Using Directed Graphs to Describe Entity Dependency in Stable Distributed Persistent Stores", in *Proceedings of Hawaii International Conference on System Sciences*, 1994.
- [35] J.L. Keedy and J. Rosenberg. "Support for Objects in the MONADS Architecture", in *Proceedings of Proceedings of the 3rd International Workshop on Persistent Object Systems*, Persistent Object Systems, pp. 392-406, 1989.
- [36] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", *IEEE Journal on Selected Areas in Communications*, 14(7), <http://www.cl.cam.ac.uk/Research/SRG/pegasus/papers/jsac-jun97/paper.html>, 1996.
- [37] J. Liedtke. "On u-Kernel Construction", in *Proceedings of 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, pp. 1-14, 1995.
- [38] A. Lindstrom, A. Dearle, R. di Bona, J. Rosenberg, and F. Vaughan, "User-Level Management of Persistent Dqta in theGrasshopper Operating System" *GH-08*, Universities of Adelaide and Sydney, <http://persistence.cs.stir.ac.uk/~al/abstracts.html#GH-08>, 1994.
- [39] A. Lindstrom, J. Rosenberg, and A. Dearle. "The Grand Unified Theory of Address Spaces", in *Proceedings of Hot Topics in Operating Systems (HotOS-V)*, pp. 66-71, <http://persistence.cs.stir.ac.uk/~al/abstracts.html#GH-11>, 1995.
- [40] R.A. Lorie, "Physical Integrity in a Large Segmented Database", *Association for Computing Machinery Transactions on Database Systems*, 2(1), pp. 91-104, 1977.
- [41] K.R. Mayes, "Trends in Operating Systems Towards Dynamic User-Level Policy Provision", , pp. 1-38, 1993.
- [42] C. Mohan, D. Haderie, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *Research Report RJ 6650*, IBM, 1989.
- [43] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle, "The Napier88 Reference Manual", *Technical Report PPRR-77-89* URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1989.html#napier.reference.manual>, Universities of Glasgow and St Andrews, 1989.

- [44] J.E.B. Moss, "Nested Transactions: An Approach to Distributed Computing", MIT Press, Cambridge, Mass, 1985.
- [45] J.E.B. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle", *IEEE Transactions on Computers*, pp. 1-39, 1991.
- [46] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba - A Distributed Operating System for the 1990s", *IEEE Computer*, **23**(5), pp. 44-53, 1990.
- [47] S.J. Mullender, G. vanRossum, A.S. Tanenbaum, R. vanRenesse, and J.M. vanStavern, "Amoeba – High-Performance distributed computing" *CS-R8937*, Vrije Universiteit, 1989.
- [48] G. Nelson, "System Programming in Modula-3", Prentice Hall, 1991.
- [49] E.I. Organick, "The Multics System: An Examination of its Structure", MIT Press, Cambridge, Mass., 1972.
- [50] D.K. Raila, S.-M. Tan, and R.H. Campbell, "Remote Procedure Call Implementations of Micro-Kernel Virtual Memory Services Degrade System Performance", *Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.
- [51] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *The Bell System Technical Journal*, **63**(6), pp. 1905-1930, 1978.
- [52] J. Rosenberg. "The MONADS Architecture - A Layered View", in *Proceedings of Proceedings of the 4th International Workshop on Persistent Object Systems*, pp. 215-225, 1990.
- [53] J. Rosenberg and D.A. Abramson. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", in *Proceedings of 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.
- [54] J. Rosenberg, A. Dearle, D. Hulse, A. Lindstrom, and S. Norris, "Operating System Support for Persistent and Recoverable Computations", *Communications of the ACM*, **39**(No.9), pp. 62-69, 1996.
- [55] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris, "Operating System Support for Persistent and Recoverable Computations", *CACM*, **39**(9), pp. 62-69, 1996.
- [56] J. Rosenberg, D. Koch, and J.L. Keedy, "A Massive Memory Supercomputer", Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences (pages 338-345), 1988.
- [57] M. Rozier, V. Abrossimov, *et al.*, "CHORUS Distributed Operating Systems", *Computing Systems*, **1**(4), pp. 305-367, 1988.
- [58] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler, "Lightweight Recoverable Virtual Memory", *ACMTOCS*, **12**(1), pp. 33-57, 1994.
- [59] M.L. Scott, T.J. LeBlanc, B.D. Marsh, T.G. Becker, C. Dubnicki, E.P. Markatos, and N.G. Smithline, "Implementation Issues for the Psyche Multiprocessor Operating System", pp. 1-22.
- [60] V. Singhal, S.V. Kakkad, and P.R. Wilson. "Texas: An Efficient, Portable Persistent Store", in *Proceedings of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Pisa, Italy, pp. 11-33, 1992.
- [61] M. Stonebraker, "Operating System Support for Database Management", *Communications of the ACM*, **24**(7), pp. 412-418, 1981.
- [62] F. Vaughan, T. Schunke, B. Koch, A. Dearle, C. Marlin, and C. Barter, "Casper: A Cached Architecture Supporting Persistence", *Computing Systems*, **5**(3), pp. 337-364, 1992.