Grasshopper: An orthogonally persistent operating system

†Alan Dearle, *Rex di Bona, *James Farrow, *Frans Henskens,
*Anders Lindström, *John Rosenberg, †Francis Vaughan

†Department of Computer Science
University of Adelaide
S.A., 5001, Australia
{al,francis}@cs.adelaide.edu.au

*Department of Computer Science
University of Sydney
N.S.W., 2006, Australia
{rex,matty,frans,anders,johnr}@cs.su.oz.au

Abstract

For ten years researchers have been attempting to construct programming language systems that support orthogonal persistence above conventional operating systems. This approach has proven to be poor; researchers invariably construct a complete abstract machine above the operating system with resulting loss of efficiency. This paper describes a new approach, the construction of an operating system designed to support orthogonal persistence. The operating system, Grasshopper, relies upon three powerful and orthogonal abstractions: *containers*, *loci* and *capabilities*. Containers provide the only abstraction over storage, loci are the agents of change, and capabilities are the means of access and protection in the system. This paper describes these three fundamental abstractions of Grasshopper, their rationale and how they are used.

1. Introduction

The aim of the Grasshopper project is to construct an operating system that supports orthogonal persistence [3]. This begs two questions, what is orthogonal persistence and why does it require support from an operating system?

The two basic principles behind orthogonal persistence are:

- that any object may persist (exist) for as long, or as short, a period as the object is required, and
- that objects may be manipulated in the same manner regardless of their longevity.

The requirements of a system which supports orthogonal persistence can be summarised as follows.

- Uniform treatment of data structures:

  Conventional programming systems require the programmer to translate data resident in virtual memory into a format suitable for long term storage. For example, graph structures must be flattened when they are mapped onto files or relations; this activity is both complex and error prone. In persistent systems, the programmer is not required to perform this mapping since data of any type with arbitrary longevity is supported by the system.
- Location independence:

  To achieve location independence, data must be accessed in a uniform manner, regardless of the location of that data. This principle is the cornerstone of virtual memory – the programmer does not have to be concerned whether the data is in RAM or on disk; the data is accessed in a uniform manner. In

Published in: *Computer Systems Journal*, 7(3), pp 289-312, Summer, 1994.

distributed persistent systems, location independence is extended to the entire computing environment by permitting data resident on other machines to be addressed in the same manner as local data [14, 15, 21, 46, 47]. This approach is also followed in distributed shared memory systems [42].

- Data resilience:

  All systems containing long-lived data must provide a degree of resilience against failure. In conventional operating systems, tools such as *fsck* in Unix permit the repair of long lived data (the file system) after a system crash. Persistent systems must also prevent the data stored in them becoming corrupt should a failure occur. However, the problem of resilience is more acute with persistent systems. In a conventional file system each file is essentially an independent object, and the loss of a single file following a system crash does not threaten the integrity of the overall system. In a persistent system, there may be arbitrary cross references between objects and the loss of a single object can be catastrophic. In addition, since one of the goals of persistence is to abstract over storage, resilience mechanisms should not be visible at the user level. In this sense the problem of recovery within a persistent system is more closely related to recovery in database systems [2].

- Protection of data:

  Persistent systems provide a large persistent store in which all data resides and against which all processes execute. A process may only access data for which it holds access permission. Failure by the operating system to provide a protection mechanism could result in erroneous processes corrupting data owned by other users. Therefore a protection mechanism must be provided to protect data from accidental or malicious misuse. In persistent systems this is typically provided via the programming language type system [31], through data encapsulation [28], using capabilities [11] or by a combination of these techniques.

To date, most persistent systems, with a few exceptions [6, 9, 38], have been constructed above conventional operating systems. Implementors of persistent languages are invariably forced to construct an abstract machine above the operating system, since the components of a persistent system are different in nature to the components of a conventional operating system. For example, in [43], Tanenbaum lists the four major components of an operating system as being memory management, file system, input-output and process management. In persistent systems, the file system and memory management components are unified. In many operating systems, input-output is presented using the same abstractions as the file system; clearly this is not appropriate in a persistent environment. Some persistent systems require that the state of a process persists. This is not easily supported using conventional operating systems in which all processes are transitory.

The principal requirements of an operating system that supports orthogonal persistence may be summarised as follows [45]:

i. Support for persistent objects as the basic abstraction: persistent objects consist of data and relationships with other persistent objects. The system must therefore provide a mechanism for supporting the creation and maintenance of these objects and relationships.

ii. The system must reliably and transparently manage the transition between long and short term memory.

iii. Processes should be persistent.

iv. Some protection mechanism to provide control over access to objects must be provided.

2. Grasshopper

Grasshopper is an operating system that provides support for orthogonal persistence. It is intended to run on a conventional hardware base, which has constrained the design of the system. A conventional hardware platform implies the lack of sophisticated features for the control of memory other than page-based virtual memory. Hence all notions of access control and structuring in Grasshopper are based on page-oriented virtual memory.

Grasshopper relies upon three powerful and orthogonal abstractions: *containers*, *loci* and *capabilities*. Containers provide the only abstraction over storage, loci are the agents of change (processes/threads), and capabilities are the means of access and protection in the system.

Conceptually, loci execute within a single container, their *host container*. Containers are not virtual address spaces. They may be of any size, including larger than the virtual address range supported by the hardware. The data stored in a container is supplied by a *manager*. Managers are responsible for maintaining a consistent and recoverable stable copy of the data represented by the container. As such, they are vital to the removal of the distinction between persistent and volatile storage, and hence a cornerstone of the persistent architecture.

This paper describes the three fundamental abstractions of Grasshopper, their rationale and how they are used. Section 3 describes the memory model (containers), Section 4 describes the process model (loci). Data sharing is described in Section 5. Section 6 describes managers and how they operate. Sections 7 deal with protection and the capability system. How the abstractions are combined to provide resilient persistent storage is described in Section 8. The paper concludes with some examples of how the abstractions provided by Grasshopper may be used.

3. Containers

In systems which support orthogonal persistence the programmer does not perceive any difference between data held in RAM and that on backing store. This idea leads naturally to a model in which there is a single abstraction over all storage. A major issue that arises is the addressing model supported by the system for this abstraction. There appear to be three basic models:

1. the single flat address space model,
2. the single partitioned address space model and
3. the fully partitioned address space model.

Models 1 and 3 represent opposite ends of the design spectrum whilst model 2 is a hybrid architecture. These models are described in the following sections.

3.1. Single flat address space

In the first model all data resides in a single flat address space with no structure is imposed upon it. This provides an addressing environment similar to that provided to a conventional

Unix process. This model is used to implement the Napier88 persistent programming system [32].

The construction of very large stores using this technique was, until recently, not feasible on conventional architectures due to address size limitations. However, the advent of machines such as the DEC Alpha [41] and the MIPS R4000 [17] which (logically) support a 64 bit address has created renewed interest in this approach. A number of research groups have suggested that this is an appropriate direction for modern operating systems [22]. Such an approach is tempting since it fits in well with the goals of orthogonal persistence, i.e. to abstract over all physical attributes of data. However, there are some difficulties:

i. Most persistent systems rely upon a checkpointing mechanism to establish a consistent state on stable storage such as disk. If the operating system supports a single massive address space, the stabilisation mechanism must either capture the entire state of this store at a checkpoint, or record the dependencies between processes and data in the store and checkpoint dependent entities together. Even using incremental techniques, the first option could take a considerable amount of time due to I/O bandwidth limitations. The second option requires system knowledge of the logical entities stored in the system so that dependency information can be maintained.

ii. If a single address space is shared by multiple processes, the ability to protect separate areas of the address space must be provided. Whilst protection systems have been designed for single address spaces [22], they do not provide any support for distribution.

iii. The resulting store would be huge and the management of large stores is difficult. In particular allocation of free space, garbage collection, unique naming of new objects and the construction of appropriate navigation tools are all more difficult in large flat stores. These and other difficulties are discussed in [34].

Solutions to these and other management issues effectively partition the flat address space. It would seem that the implementation of a single partitioning scheme would be more efficient than the use of separate schemes to support each management requirement

3.2. Single partitioned address space

In the second model the notion of a large address space in which all objects reside is retained. This address space is, however, partitioned into semi-independent regions. Each of these regions contains a logically related set of data and the model is optimised on the assumption that there will be few inter-region references. Such an approach is the basis of the Monads architecture [40]. Provided that control can be retained over the inter-region references it is possible to garbage collect and checkpoint regions (or at least limited sets of regions) independently, alleviating problems (i) and (iii) above [5, 39]. In addition, the partitioning provides a convenient mechanism for the generation of unique object names [14].

The major problem that remains with this approach is the issue of protection. It is necessary to restrict the set of addresses which can be generated by a process. One possibility is to provide special-purpose hardware to support protection in a partitioned store; an implementation of such an architecture has been described previously in [38]. However, conventional architectures provide only page-based protection and therefore protection mechanisms similar to those proposed for flat stores must be employed.

3.3. Fully partitioned address space

In the third model the store is fully partitioned. Each partition is logically equivalent to an instance of the flat address space described in 3.1 and defines an independent addressing environment; there is no global address space. As we shall see, however, there is no reason for the size of a partition to be restricted by the address size of the host architecture.

In this model, processes execute within a single partition and may only access the data visible in that partition. The use of multiple independent partitions has several advantages. Firstly, partitions may be of arbitrary size, not restricted (individually and collectively) by the size of a global address space. Secondly, partitions are truly independent and not part of some larger structure, allowing different management techniques to be implemented for each region. Lastly, partitions may have names and operators may be provided that operate over them.

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

Facilities must be provided which allow the transfer of data between containers. The mechanisms provided in Grasshopper are mapping and invocation. These are described in the following sections.

4. Loci

In Grasshopper, loci are the abstraction over execution (processes). In its simplest form, a locus is simply the contents of the registers of the machine on which it is executing. Like containers, a locus is maintained by the Grasshopper kernel and is inherently persistent. Making the locus persistent is a departure from other operating system designs and frees the programmer from much complexity.

Throughout its life, a locus may execute in many different containers. At any instant in time, a locus executes within a distinguished container, its *host containe*r. The locus perceives the host container's contents within its own address space. Virtual addresses generated by the locus map directly onto addresses within the host container. A container comprising program code, mutable data and a locus forms a basic running program. Loci are an orthogonal abstraction to containers. Any number of loci may execute within a given container; this allows Grasshopper to support multi-threaded programming paradigms.

An operating system is largely responsible for the control and management of two entities: objects, which contain data (containers); and processes (loci), the active elements which manipulate these objects. One of the most important considerations in the design of an operating system is the model of interaction between these entities. There are two principal models of computation called *message oriented* and *procedure oriented* [25]. In the message oriented model, processes are statically associated with objects and communication is achieved through the use of messages. By contrast, the procedure oriented model provides processes that move between objects. Processes access objects by invoking them; the invoke operation causes a (possibly remote) procedure call to code within the object. By executing this code the process may access data stored within the object. The message oriented model cannot be used to efficiently simulate any other computational models. The procedure oriented model is more flexible; for instance, it can easily simulate the message oriented model by associating a message passing process with every object. For this reason, Grasshopper uses the procedure oriented model in which a locus may invoke a container thereby changing its host container.

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

The invoking locus provides a parameter block to the kernel mediated invoke operation. This parameter block is made available to the code in the invoked container. Larger amounts of data may be passed via an intermediate container. Appropriate and arbitrarily sophisticated communication protocols may be built on top of this simple facility. Since a minimal parameter block is the only context that is transferred to the invoked container, invocation is therefore inherently low cost. In this respect, the invoke primitive is very similar to the message passing system used in the V-kernel [7].

A locus may invoke and return through many containers in a manner similar to conventional procedure calls. The Grasshopper kernel maintains a call chain of invocations between containers. Implicitly each locus is rooted in the container representing the kernel: when a locus returns to this point it is deleted. However some loci may never need to return to the container from which they were invoked. Such a locus may meander from container to container. In such circumstances, an invoke parameter allows the locus to inform the kernel that no return chain need be kept.

5. Container mappings

The purpose of container mapping is to allow data to be shared between containers. This is achieved by allowing data in a region of one container to appear in another container. In its simplest form, this mechanism provides shared memory and shared libraries similar to that provided by conventional operating systems. However, conventional operating systems restrict the mapping of memory to a single level. Both VMS [26] and variants of Unix (such as SunOS) provide the ability to share memory segments between process address spaces, and a separate ability to map from disk storage into a process address space. Several other systems [7, 8] provide the notion of a *memory object*, which provides an abstraction of data. In these systems, memory objects can be mapped into a process address space, however memory objects and processes are separate abstractions. It is therefore impossible to directly address a memory object, or to compose a memory object from other memory objects.

By contrast, the single abstraction over data provided by Grasshopper may be arbitrarily recursively composed. Since any container can have another mapped onto it, it is possible to construct a hierarchy of container mappings as shown in Figure 1. The hierarchy of container mappings form a directed acyclic graph maintained by the kernel. The restriction that mappings cannot contain circular dependencies is imposed to ensure that one container is always ultimately responsible for the data. In Figure 1, container *C2* is mapped into container *C1* at location *a1*. In turn, *C2* has regions of containers *C3* and *C4* mapped into it. The data from *C3* is visible in *C1* at address *a3*, which is equal to *a1* + *a2*.

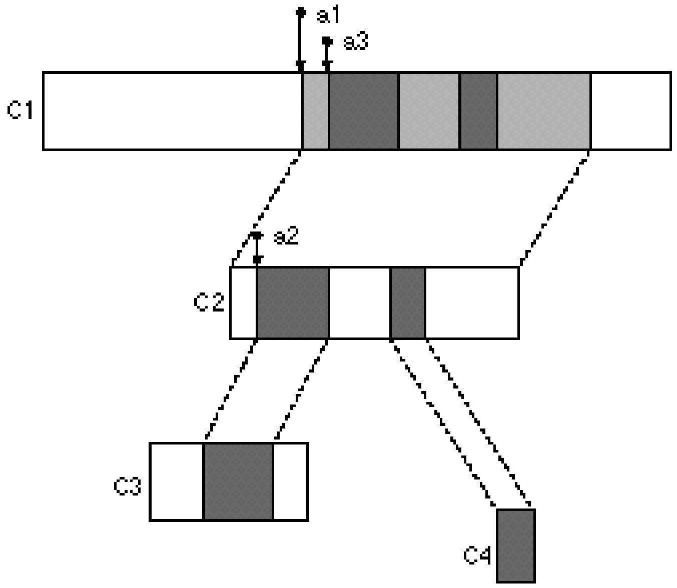

Figure 1: A container mapping hierarchy

Loci perceive the address space of their host container. Therefore all loci executing within a container share the same address space. However, a locus may require private data, which is visible to it, yet invisible to other loci that inhabit the same container. To satisfy this need, Grasshopper provides a viewing mechanism known as *locus private mapping*.

Locus private mappings are similar to container mappings in that they make data from one container appear in another. However, instead of being globally visible, locus private mappings are only visible to the locus in which they are created and take precedence over host container mappings. This facility allows, for example, each locus to have its own stack space with the stacks of all loci occupying the same address range within their host containers as shown in Figure 2. The effect of locus private mappings remain visible at all times until they are removed.

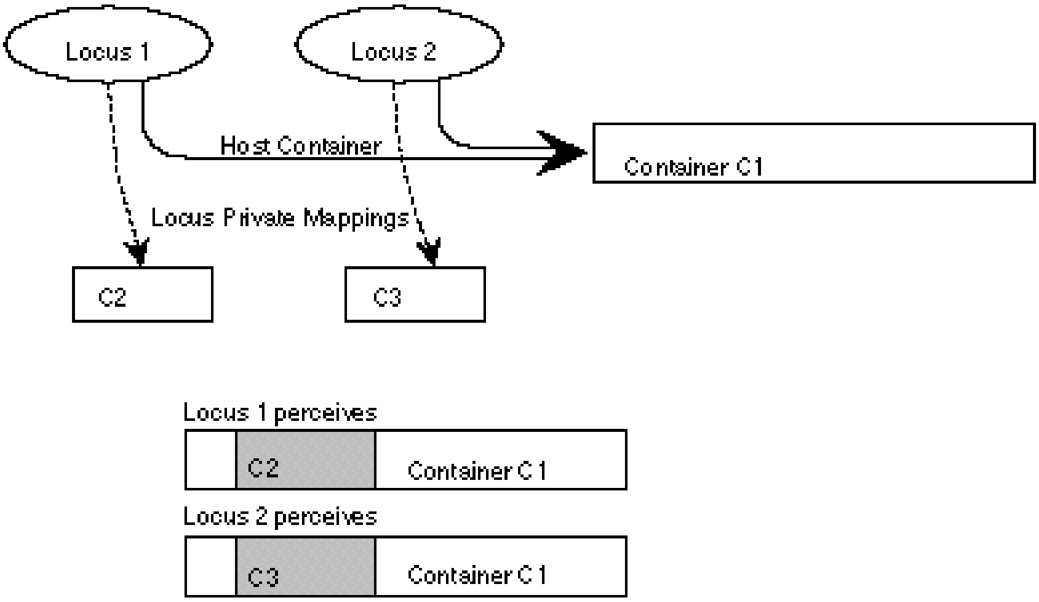

Figure 2: Loci with private stack space

Consequently if locus 1 in Figure 2 were to invoke some other container C4 and map in container C1 it would see the data from C2 from the mapping instantiated whilst running in container C1. This is shown in Figure 3 below. This technique both simplifies multi-threaded programming and provides a useful security mechanism that is unavailable using conventional addressing mechanisms.

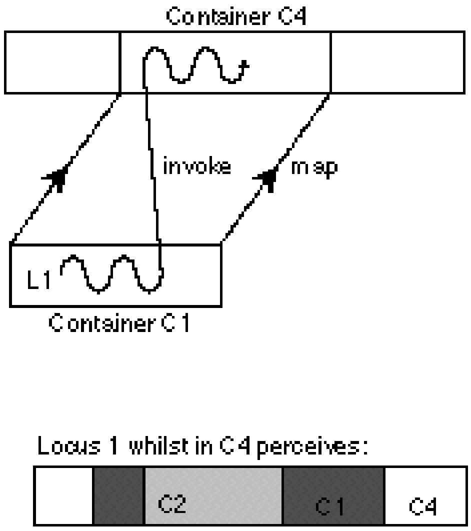

Figure 3: Multi level mappings

6. Managers

Thus far we have described how all data storage in Grasshopper is provided by containers. However, we have not described how containers are populated with data. When data in a container is first accessed, the kernel is required to provide the concrete data that the container represents. A locus executing within a container accesses the data stored in it using container addresses. The container address of a word of data is its offset relative to the start of the container in which it is accessed. Managers are responsible for providing the required data to the kernel and are also responsible for maintaining the data when it is not RAM resident. Rather than being part of the kernel, managers are ordinary programs which reside and execute within their own containers; their state is therefore resilient. The concept of a manager is similar to the Mach external pager [36, 50] which has been successfully used to implement a coherent distributed persistent address space [21]. In common with Mach and more recent systems [13, 20], managers are responsible for:

- provision of the pages of data stored in the container,
- responding to access faults, and
- receiving data removed from physical memory by the kernel.

In addition, Grasshopper managers have the following responsibilities:

- implementation of a stability algorithm for the container [5, 24 , 29 , 39, 47], i.e. they maintain the integrity and resilience of data, and
- maintenance of coherence in the case of distributed access to the container [15, 27, 35].

A manager is invoked whenever the kernel detects a memory access fault to data stored in the container it manages. Making data accessible in a container takes place in two steps:

i. the manager associated with a particular address range must be identified, and,

ii. the appropriate manager is requested to supply the data.

The kernel is responsible for identifying which manager should be requested to supply data. This is achieved by traversing the container mapping hierarchy. Once the correct manager has been identified, the kernel requests this manager to supply the data. The manager must deliver the requested data to the kernel, which then arranges the hardware translation tables in such a way that the data is visible at an appropriate address in the container.

In Grasshopper, the manager is the only mechanism by which data migrates from stable to volatile storage. This is in contrast to conventional operating systems in which the usual abstraction of stable storage is the file system. Grasshopper has no file system in the conventional sense.

Managers are responsible for maintaining a resilient copy of the data in a container on stable media. It is only within a manager that the distinction between persistent and ephemeral data is apparent. Managers can provide resilient persistent storage using whatever mechanism is appropriate to the type of data contained in the managed container. Since managers are responsible for the storage of data on both stable media and in RAM they are free to store that data in any way they see fit. An important class of managers are those that store data on stable media in some form other than that viewed by a locus in a container; we term these manipulative managers. Some examples of manipulative managers are:

i. swizzling managers,

ii. encrypting managers, and,

iii. compressing managers.

Swizzling managers are particularly interesting in that they permit the use of containers that are larger than the address space supported by the host hardware. A locus executing within a large container will generate addresses constrained by the machine architecture. Access faults will be delivered by the kernel to the manager associated with a container. A swizzling manager will provide a page of data in which addresses that refer to objects anywhere within the container are replaced with shorter addresses (ones within the machine's address range) which, when dereferenced will be used by the swizzling manager to provide the correct data [4, 45, 48].

It is possible for a locus to execute in a container whose manager provides a one-to-one mapping between data in virtual memory and data on disk. This is the abstraction delivered by demand paged virtual memory and memory mapped files in conventional operating systems. Most managers will not operate in this manner since this does not adequately support orthogonal persistence.

7. Capabilities

Capabilities provide an access control mechanism over containers and loci. For containers, access control is required over:

- container maps,
- the containers which may be invoked,
- the ability to set an invocation point,

- the containers which may be mapped and the access type available to mapped regions (read/write), and
- deletion of containers,

and for loci access control is required over:

- creation of locus private mappings,
- blocking and unblocking loci,
- management of exceptions and
- deletion of loci.

In conventional operating systems these access controls are usually provided by the file system interface. This is clearly inappropriate for Grasshopper. In several existing persistent systems protection is provided via the programming language type system [31] or through data encapsulation [28]. Grasshopper is intended to support multiple languages and therefore cannot rely solely on a type system.

The protection abstraction provided by Grasshopper is the *capability* [11]. Capabilities were first proposed by Dennis and Van Horn [10] and have been used in a variety of contexts as an access control mechanism [16, 38, 44, 49]**.** A capability consists of a unique name for an entity, a set of access rights related to that entity, and rights pertaining to the capability itself, in particular whether the capability can be copied. An operation can only be performed if a valid capability for that operation is presented. There are two important points about capabilities from which they derive their power: the name of the entity is unique and capabilities cannot be forged or arbitrarily modified. Capabilities can only be created and modified by the system in a controlled manner.

There are well-known techniques for achieving the above requirements. Unique names may be generated by using a structured naming technique where each machine is given a unique name and each entity created on that machine has a unique name. Such a technique is described in [15]. Protection of capabilities can be achieved in one of three ways:

tagging: in which extra bits are used by the hardware to indicate memory regions representing capabilities and to restrict access to these regions,

segregation: in which capabilities are stored in a protected area of memory,

passwords: in which a key embedded in a sparse address space is stored with the entity and a matching key must be presented to gain access to that entity.

The merits of each of these techniques have been well discussed in the literature [1, 12, 18, 44]. Given that Grasshopper is to be implemented on conventional hardware, tagging is not an option. Segregation is used in Grasshopper since it avoids the problems associated with knowing when to garbage collect unreferenced entities. This occurs with password capabilities since a user may keep a copy of the capability somewhere outside of the kernel's control. Since the kernel cannot know how many (if any) of these externally recorded capabilities may be in existence it cannot perform garbage collection except on entities it is specifically told to destroy. Using segregated capabilities allows garbage collection to be performed in association with explicit destruction of entities by a locus. When the reference count on a capability falls to zero, i.e., there are no more extant references to the corresponding entity, the entity may be deleted.

One of the criticisms of capabilities as a protection technique is that they are expensive to implement without hardware support. This would be of some concern if capabilities were to be used to control access to all objects (e.g. records and integers). This is not the case in Grasshopper; capabilities are only used to control operations on containers and loci, i.e. coarse grain objects. In fact we expect that protection system in Grasshopper to be considerably more efficient than the equivalent in a conventional system, i.e. the file and directory system.

In Grasshopper, every container and locus can have an associated list of capabilities. A capability list is constructed out of tuples containing a unique fixed length key and a capability. Operations are provided for copying capabilities and for adding and removing them to and from lists. At any time, a locus has access to:

i. all the capabilities in its own list,

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

Programs can refer to capabilities by specifying a capability list (locus or host container) and a key. Grasshopper checks that an entry with the given key exists in the specified list. An appropriate capability must be presented for most operations involving the manipulation of entities, such as invocation and mapping.

A number advantages arise from the use of capabilities for access and protection:

i. Distribution is completely transparent: A locus wishing to invoke a container simply presents the capability. The capability uniquely identifies the container and its physical location is irrelevant.

ii. The system does not force any particular protection structure upon users. It is possible to construct hierarchical protection or more general policies using arbitrary naming mechanisms which map some representation of a name onto a key and thereby onto a capability.

iii. It is possible to create restricted views of objects. For example, two different capabilities for a container could be created, one of which allows the container to be mapped, while the other only allows it to be invoked.

iv. It is possible to revoke access. A holder of a capability with appropriate access rights can invalidate other capabilities for the same entity.

8. Persistence

Containers and their associated managers provide the abstraction of persistent data. Managers are responsible for maintaining a consistent and recoverable stable copy of the data represented by the container. As part of its interface, each manager must provide a stabilise operation [5, 39]. Stabilisation involves creating a consistent copy of the data on a stable medium.

Managers alone are not able to maintain a system-wide consistent state. For example, consider the case where two containers, *A* and *B*, both provide data which is used and modified by a single program. The manager for container *A* stabilises the state of *A*, and execution of the main program continues. At a later time, container *B* is stabilised. This does not result in a consistent view of data from the point of view of the executing program, since after a system crash the recovered state of the two containers are inconsistent relative to one another.

The simplest approach to global consistency is to enforce system wide simultaneous stabilisation in which the kernel blocks all executing loci and requests each manager to stabilise. The disadvantage of this approach is that the entire system freezes whilst the stabilise occurs.

An alternative approach is to stabilise parts of the store separately. In such a system it is necessary to determine which containers and loci are causally dependent upon each other's state and only force these to stabilise together, leaving the rest of the system to run. Such inter-dependent units are termed *associates* [46]. The kernel may determine which containers and loci are inter-dependent by annotating the container mapping graph with a history of invocations and dependencies on kernel data. The internal state of kernel data structures also forms part of the state of a user program. For example, the granting of capabilities to loci must be recorded. The causal dependencies must therefore be extended to include kernel data. Thus a complete, consistent and recoverable representation of a subsection of the system can be produced.

When a consistent subset of the system has been determined, the kernel proceeds with a two phase commit protocol, requesting the appropriate container managers to stabilise the data in their charge. When all user data is stable, the kernel will proceed to stabilise its own state, and finally, as an atomic action, will commit a descriptor block that describes the new state to stable medium.

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

9. Providing A Persistent Environment

A key concept behind orthogonal persistence is that the programmer is not required to manage the movement of data between primary and secondary storage. Instead, application programs execute within a stable, resilient addressing environment in which data locality is invisible. A number of persistent systems have been constructed which support particular programming languages on a variety of architectures.

In the first systems to be called persistent [3] all data was stored in objects. Prior to every object dereference, a run time check was performed to ensure that the object was resident in memory. If not, the object was loaded from persistent storage by the Persistent Object Management System (POMS) [4]. This technique requires considerable complexity to ensure that objects are not loaded more than once and that objects are copied back to persistent storage atomically.

Recently a large number of object-oriented database implementations have appeared both commercially and as research vehicles [23, 37]. A large number of these have been based on the language C++ and rely upon the implementor changing the C++ run time system in some manner to load and store persistent data. One popular technique used by these systems is to overload the deference operator “∅” to perform residency checking [6]. Like the PS-algol systems, if the object is not resident in virtual memory it is first loaded from the persistent store and the dereference is allowed to proceed. Another technique used by Moss [33] to implement persistent Smalltallk-80 uses dummy methods to load non resident objects in a similar manner.

Rather than implement persistence with explicit object management, the persistent address space may be embedded within the virtual address space. Using this technique, pages of persistent data may be incrementally loaded from a page based object store using conventional page faulting techniques. Dirty pages must be written to a different site to guarantee that at least one self consistent copy of the store exists at any time. Systems using this technique have been implemented above VMS [30], Mach [47] and Unix systems [48].

All the above techniques require considerable implementation effort by the provider of the persistence mechanism who often expends significant effort working around the inappropriate mechanisms provided by conventional operating systems. These work arounds inherently compromise the efficiency of these implementations. Secondly, the techniques are tied to a particular language system; for example, the techniques used to implement PS-algol are not necessarily appropriate for Smalltalk or C++.

By contrast, the abstractions described in this paper provide an ideal platform for the construction of persistent systems. In the remainder of this section we will demonstrate how different user level systems may be constructed using the abstractions provided by the Grasshopper kernel.

In Grasshopper even the most primitive language, for example C, may be provided with a resilient, persistent execution environment without modification to the compiler or run time system. This may be achieved in a variety of ways with varying amounts of sophistication. However, the simplest is to execute each program in a Grasshopper container with its own manager as shown in Figure 4 below.

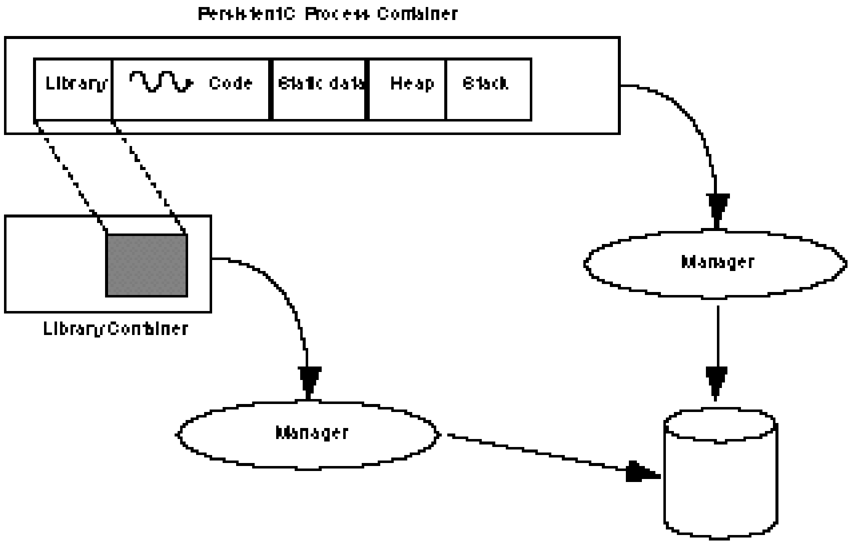

Figure 4: Running a C program under Grasshopper.

In this scheme, a C program is organised in memory in the same manner as under Unix, with code followed by static data, heap and stack space. The managers provide both the functionality of conventional demand paged virtual memory plus resilience. The manager may save the entire state of the running process on disk and restart it at any arbitrary point in its execution in a manner that is totally transparent to the running process. Libraries may be provided by mapping library code from other containers into the address space of the process. Libraries provide both the usual Unix style libraries and code for communicating with other processes and for binding to persistent data. The last of these is crucial since to be useful, a process will require to bind to global data.

In Unix, binding to persistent data is achieved through the use of a file system. Symbolic names (file names) are dynamically mapped to sequences of bytes (files). All access to persistent data is via a set of predefined system calls (open, creat, write etc.). In Grasshopper, persistent data external to a process is accessed via *binding servers*.

Binding servers are implemented as containers whose invocation interface present functions that provide access to external persistent data. Access may be provided either through further invocation, via container mapping or both as appropriate. For example, consider a locus wanting access to a database service. The locus must first call a binding server with the name of the database and capabilities as a parameters. Assuming that the locus has appropriate capabilities, the server will install the capability for the database in the locus. The locus now has the capability to directly call the database without further assistance by the binding server. Whenever access is required to the database, its container is invoked with an appropriate request. Access to the database will be effected by the database code which may map data from appropriate parts of the database into the address space of the requesting locus.

The above scenario assumes that the locus requiring database access does not already hold a capability for the database. Such an assumption is the norm in a system such as Unix in which processes are typically created with no knowledge of the environment in which they operate. In Grasshopper loci may be populated with capabilities at any time in their lifetime. In particular, a locus may be populated with all the capabilities it requires at the time it is created. This facility has two major benefits. Firstly it is more efficient, since loci do not have to perform dynamic name binding in order to acquire resources. Secondly it is more secure; a locus may be loaded with exactly the resources it requires to perform some task. If that locus is denied access to a binding server, it cannot access any system components for which it does not already hold a capability.

As an example of how Grasshopper may be used at the user level consider a user logging into a Grasshopper system. In Grasshopper, user environments are embodied in a distinguished container and a collection of loci and other containers which are persistent. When a user logs out, their environment continues to exist. Therefore, rather than create a new environment on each login, login consists of binding to the extant environment. This task is performed by a login server which maintains a mapping between (user name, password) pairs and capabilities for the distinguished containers. Such a scheme is described in [19].

A user wishing to reconnect with their environment, provides the authentication server with a user name and a password. This request must be accompanied by a capability for the devices on which the user wishes to interact. These may be provided by the locus that mediates the connection with the grasshopper system (c.f. getty in Unix). Once the authentication process is complete, the server invokes the user's distinguished container with a locus carrying with it the capabilities for the devices. Using these capabilities, the locus may re-establish conversation with the user in whatever manner is appropriate.

10. Conclusions

In this paper the initial design of The Grasshopper Persistent Operating System has been described. Grasshopper satisfies the four principle requirements of orthogonal persistence namely:

i. support for persistent objects as the basic abstraction,

ii. the reliable and transparent transition of data between long and short term memory,

iii. persistent processes and

iv. control over access to objects.

This is achieved through the provision of three powerful and orthogonal abstractions namely *containers*, *loci* and *capabilities*. Containers provide the only abstraction over storage, loci are the agents of change, and capabilities are the means of access and protection in the system. These abstractions are supported by a fourth mechanism, the manager, which is responsible for data held in a container.

Based on our experience of constructing persistent systems, we believe that these abstractions provide an ideal base for persistent programming languages. At the moment we cannot prove this assertion since the Grasshopper system is still under construction on a DEC Alpha platform.

Acknowledgements

We would like to thank Karen Wyrwas, Alex Farkas, Stephen Norris, Fred Brown and David Hulse for comments on earlier versions of this paper. We also thank the anonymous referees for their many helpful comments on the paper.

This work is supported by a grant number A49130439 from the Australian Research Council and by an equipment donation from Digital Equipment Corporation under the Alpha Innovators Program.

References

1. Anderson, M., Pose, R. and Wallace, C. S. “A Password-Capability System”, *The Computer Journal*, vol 29, 1, pp. 1-8, 1986.
2. Astrahan, M. M. “System R: Relational Approach to Database Management”, *TODS*, vol 1, 2, pp. 97-137, 1976.
3. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. “An Approach to Persistent Programming”, *The Computer Journal*, vol 26, 4, pp. 360-365, 1983.
4. Atkinson, M. P., Bailey, P. J., Cockshott, W. P., Chisholm, K. J. and Morrison, R. “POMS: A Persistent Object Management System”, *SPE*, vol 14, 1, pp. 49-71, 1984.
5. Brown, A. L. “Persistent Object Stores”, Ph.D thesis, Computational Science, Universiy of St. Andrews, 1988.

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

7. Cheriton, D. R. “The V Kernel: A Software Base for Distributed Systems”, *Software*, vol 1, 2, pp. 91-42, 1984.

8. Chorus-Systems “Overview of the CHORUS Distributed Operating Systems”, *Computer Systems - The Journal of the Usenix Association*, Vol 1 No 4., 1990.

9. Dasgupta, p., LeBlanc, R., Mustaque, A. and Umakishore, R. “The Clouds Distributed Operating System”, Arizona State University, Technical Report 88/25, 1988.

10. Dennis, J. B. and Van Horn, E. C. “Programming Semantics for Multiprogrammed Computations”, *Communications of the A.C.M.*, vol 9, 3, pp. 143-145, 1966.

11. Fabry, R. “Capability-Based Addressing”, *Communications of the ACM*, vol 17, 7, pp. 403-412, 1974.

12. Gehringer, E. F. and Keedy, J. L. “Tagged Architecture: How Compelling are its Advantages”, *Twelfth International Symposium on Computer Architecture*, pp. 162-170, 1985.

13. Harty, K. and Cheriton, D. R. “Application-Controlled Physical Memory Using External Page-Cache Management”, *ASPLOS V*, Boston, ACM, 1992.

14. Henskens, F. A. “Addressing moved modules in a capability based distributed shared memory”, *25th Hawaii International Conference on System Sciences*, Kauai, Hawaii, IEEE, pp. 769-778, 1992.

15. Henskens, F. A., Rosenberg, J. and Keedy, J. L. “A Capability-based Distributed Shared Memory”, *Proceedings of the 14th Australian Computer Science Conference*, Sydney, Australia, pp. 29.1-29.12, 1991.

16. Hurst, A. J. and Sajeev, A. S. M. “Programming Persistence in χ”, *I.E.E.E. Computer*, vol 25, 9, pp. 57-66, 1992.

17. Kane, G. and Heinrich, J. “MIPS RISC Architecture”, Prentice-Hall, 1992.

18. Keedy, J. L. “An Implementation of Capabilities without a Central Mapping Table”, *Proc. 17th Hawaii International Conference on System Sciences*, pp. 180-185, 1984.

19. Keedy, J. L. and Vosseberg, K. “Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System”, *25th Hawaii International Conference on System Sciences*, Kona, Hawaii, IEEE, pp. 747-756, 1992.

20. Khalidi, Y. A. and Nelson, M. N. “The Spring Virtual Memory System”, Sun Microsystems Laboratories, Technical Report TR-93-09, 1993.

21. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. “Cache Coherence and Storage Management in a Persistent Object System”, *Proceedings, The Fourth International Workshop on Persistent Object Systems*, Marthas Vineyard, Morgan Kaufmann, pp. 99-109, 1990.

22. Koldinger, E., Chase, J. and Eggers, S. “Architectural Support for Single Address Space Operating Systems”, *Architectural Support for Programming Languages and Operating Systems*, Boston, Mass, ACM, pp. 175-197, 1992.

23. Lamb, C., Landis, G., Orenstein, J. and Weinreb, D. “The Objectstore Database System”, *CACM*, vol 34, 10, pp. 50-63, 1991.

24. Lampson, B. “Atomic Transactions”, *Distributed Systems – Architecture and Implementation*, Lecture Notes in Computer Science, vol 105, Springer-Verlag, Berlin, pp. 246-265, 1981.

25. Lauer, H. C. and Needham, R. M. “On the Duality of Operating System Structures”, *Operating Systems Review*, vol 13, 2, pp. 3-19, 1979.

26. Levy, H. M. and Lipman, P. H. “Virtual Memory Management in the VAX/VMS Operating System”, *Computer*, vol 15, 3, pp. 35-41, 1982.

27. Li, K. “Shared Virtual Memory on Loosely Coupled Multiprocessors”, Ph.D. thesis, Yale University, 1986.

28. Liskov, B. H. and Zilles, S. N. “Programming with Abstract Data Types”, *SIGPLAN Notices*, vol 9, 4, pp. 50-59, 1974.

29. Lorie, R. A. “Physical Integrity in a Large Segmented Database”, *Association for Computing Machinery Transactions on Database Systems*, vol 2, 1, pp. 91-104, 1977.

30. McLellan, P. and Chisholm, K. “Implementation of a POMS: Shadow Paging via VAX VMS Memory Mapping”, Computer Science, University of Edinburgh, Technical Report 1982.

31. Morrison, R., Brown, A. L., Connor, R. C. H., Cutts, Q. I., Kirby, G. N. C., Dearle, A., Rosenberg, J. and Stemple, D. “Protection in Persistent Object Systems”, *Security and Persistence*, Workshops in Computing, Springer-Verlag, pp. 48-66, 1990.

32. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. “The Napier88 Reference Manual”, University of St Andrews, Technical Report PPRR-77-89, 1989.

33. Moss, E. and Sinofsky, A. “Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface”, *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science, vol 334, Springer-Verlag, berlin, pp. 298-316, 1988.

34. Moss, J. E. B. “Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach”, *2nd International Workshop on Database Programming Languages*, Salishan, Oregon, *San Mateo, California*, Morgan Kaufmann, pp. 358-374, 1989.

35. Philipson, L., Nilsson, B. and Breidegard, B. “A Communication Structure for a Multiprocessor Computer with Distributed Global Memory”, *10th International Symposium on Computer Architecture*, Stockholm, pp. 334-340, 1983.

36. Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures”, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, ACM Order Number 556870, pp. 31-39, 1987.

37. Richardson, J. E. and Carey, M. J. “Implementing Persistence in E”, *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, Springer-Verlag, pp. 175-199, 1989.

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

39. Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. "Stability in a Persistent Store Based on a Large Virtual Memory", *Proceedings of the International Workshop on Architectural Support for Security and Persistence of Information*, Bremen, West Germany, Springer-Verlag and British Computer Society, Workshops in Computing, pp. 229-245, 1990.

40. Rosenberg, J., Keedy, J. L. and Abramson, D. "Addressing Mechanisms for Large Virtual Memories", *The Computer Journal*, vol 35, 4, pp. 369-375, 1992.

41. Sites, R. L. "Alpha Architecture Reference Manual", Digital Press, 1992.

42. Tam, M., Smith, J. M. and Farber, D. J. "A Taxonomy-based Comparison of Several Distributed Shared Memory Systems", *Operating Systems Review*, vol 24, 3, pp. 40-67, 1990.

43. Tanenbaum, A. S. "Operating Systems: Design and Implementation", Prentice Hall, International Editions, 1987.

44. Tanenbaum, A. S. "Experiences with the Amoeba Distributed System", *CACM*, vol 33, 12, pp. 46-63, 1990.

45. Vaughan, F. and Dearle, A. "Supporting Large Persistent Stores Using Conventional Hardware", *5th International Workshop on Persistent Object Systems*, San Miniato, Italy, Springer-Verlag, vol (to appear), 1992.

46. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "A Persistent Distributed Architecture Supported by the Mach Operating System", *Proceedings of the 1st USENIX Conference on the Mach Operating System*, Burlington, Vermont, pp. 123-140, 1990.

47. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "Casper: A Cached Architecture Supporting Persistence", *Computing Systems*, vol 5, 3, California, 1992.

48. Wilson, P. "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", *Computer Architecture News*, June, pp. 6-13, 1991.

49. Wulf, W. A., Levin, R. and Harbinson, S. P. "C.mmp/Hydra: An Experimental Computer System", McGraw-Hill, NewYork, 1981.

50. Young, M. W. "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System", Ph.D. thesis, Computer Science, Carnegie Mellon University, 1989.