CS/TR-92-52

A model for persistent shared memory addressing in distributed systems

Paulo Amaral, Christian Jacquemot, Rodger Lea

approved by:

abstract: This paper was published by IEEE Computer Society and presented in the International Workshop of Object Oriented Operating Systems IWOOOS'92

September 1992

Contents

Abstract

COOL v2 [1] *is an object oriented persistent computing system for distributed programming. With* COOL v2 *, C++ objects can be persistent and shared freely between applications and distributed across sites in a completely transparent manner from the programmer's point of view.*

To address the problem of maintaining distributed shared data coherency, data persistency and address allocation coherency, we developed the persistent context space *model which encapsulates distributed shared memory and persistent memory, and controls distributed shared memory address allocation.*

This paper outlines existing solutions of object addressing in persistent and distributed environments and contrasts these with the persistent context space *model and its integration in an operating system architecture.*

1 Introduction

The use of the UMA [2] programming model on distributed systems has been investigated as a means to preserve programming simplicity and benefit at the same time from the increased parallelism of a MIMD multiprocessor architecture. Chandy [8] expects future operating systems to mask distribution and integrate NUMA [3] architectures with a single address space programming model in order to ease the use of distributed resources. In terms of performance, Larowe [22] verifies that UMA applications can come close to highly tuned NUMA ones on distributed architectures if memory management is carefully implemented. We think that this apparently small performance penalty is worthwhile trading for a simpler programming model.

Distributed shared memory [25] extends the concept of shared memory to a loosely coupled network of sites. This can already be found in commercial operating systems like CHORUS [27] and DOMAIN [24].

Unfortunately, the use of the distributed shared memory model alone is not completely transparent to applications, that is, information on shared memory segments has to be passed around between remote applications so that each one instructs its local kernel to map the correct memory segment at the correct address.

There are many solutions to this, however the most promising, and successful, use language semantics to denote language units, and hence memory that must be shared. Object oriented languages implicitly provide these units of sharing, objects, and hence provide an excellent starting point from which to tackle the distributed shared memory problem.

Using the object oriented paradigm, can we avoid the problem of distributed address allocation? Can applications share objects freely and transparently as if they were local without caring about address allocation and control at application level? In a language

[1] *COOL v2* stands for the second version of the CHORUS Object Oriented Layer.
[2] UMA stands for Uniform Memory Access
[3] NUMA - Non UMA

like C++, where objects are identified by the use of local virtual memory pointers, this transparency seems not so easy to achieve because there is an obvious relationship between virtual memory and object semantics, that is, n C++ object identifiers (pointers) are address dependent. We argue that run-time and operating system support is necessary to solve this. Examples of distributed computing systems with a centralized programming style are Clouds [15], Amber [9] and Orca [3].

Our goal in the *COOL v2* project is to build a distributed computing system with a UMA programming metaphor for a loosely distributed type of architecture. We have adopted the object oriented paradigm for its simplicity and programming power and developed a first prototype for C++. Not wanting to touch the compiler, we explored the transparent use of C++ objects in a persistent distributed environment, relying on run-time and operating system support and on a pre-processor. Objects only have meaning to applications; the *COOL v2* operating system treats objects as data that is persistent and can be shared.

To achieve this level of transparency and because we are using a UMA pointer-based language, we have to deal, at the operating system level, with four main problems:

data persistency In a persistent programming system all data is potentially persistent, i.e., objects outlive the scope of applications [2].

address allocation coherency for persistent data Persistent programming with a language where objects are identified by virtual memory pointers assumes that an object referenced at some address has to be the same when referred by another application later on at the same address. Address allocation has to take into account persistent addresses already in use.

address allocation coherency for shared data Low-level object pointers are address dependent. In order to be shared concurrently, applications have to use the same addresses for the same data.

shared data coherency Data has to be kept coherent when used at the same time by various distributed applications. Sharing data concurrently assumes a one-copy semantics of shared data with a *single-writer multiple-reader* coherence with some level of granularity (object, segment or page).

We developed the *persistent context space* model to address the problems above and implemented it in *COOL v2*, allowing the support pointer based languages like C++ in a distributed environment. The model is based on the following items:

1. all data is potentially persistent;
2. persistent memory addresses are allocated in a persistent manner, i.e., addresses are allocated until explicitly deleted;
3. memory is shared concurrently with the *distributed shared memory* model of Li [25] at fixed addresses;
4. data can be shared at different addresses in exclusive periods of time, providing memory relocation;

5. data is relocated accessing the semantics of the objects that are stored inside.

This paper is organized as follows. In section 2 we overview background work on this subject. In section 3 the *persistent context space* is presented. In section 4 we explain the operating system requirements to accommodate the *persistent context space* model, and we draw our conclusions in section 5.

2 Background

In this section we outline basic concepts used throughout the rest of the paper and we will overview background work under the light of the above issues, that is, how do current research systems provide distributed coherency of data sharing, address allocation and data persistency.

2.1 Basic concepts

In current programming systems, e.g. UNIX, applications execute on virtual machines in processes; processes abstract computation and contexts represent resources. In particular, a context has a linear and contiguous address space of memory from 0 to some maximum value. Data can be mapped in context memory using some allocation request. The address space is limited to the machine architecture [4] and is used only for volatile entities. Persistent data is preserved on secondary storage in external formats (objects or simple data structures).

Several contexts can exist on each processor with separate address spaces. With the shared memory model, it is possible to map at the same time, and in a single site [5], the same memory into different address spaces, thus in different contexts.

In the following section we overview some background work in terms of data sharing and persistency as well as address allocation. We are particularly interested in systems that are distributed.

2.2 Uniform address spaces for shared data

2.2.1 Clouds

The Clouds system [14] also works on a connected network of workstations; the system behaves like a single large computer. It implements a local segmented machine [15] and objects are a set of segments that can be shared across machines; it embeds a coherency protocol to assure one-copy semantics of objects (and objects' segments) with multiple cached copies of the same segment on different compute servers [6]. It thus uses distributed shared memory (DSM).

Objects are global and seen in all compute servers. Objects' segments are virtual memory segments and are always assigned the same virtual address [28]. So, objects are identified by

[4] Current 32 bit machines have 4 gigabytes of virtual memory.
[5] A site is a set of tightly coupled processors and memory (volatile and persistent) with network connection.
[6] Clouds uses a minimal kernel approach with the Ra kernel whereas we use the Chorus micro-kernel

address and there is a uniform address space for all shared (and persistent) objects. They are installed in virtual memory the moment they are instantiated.

Clouds segments are persistent by nature, and so are objects. There is a naming system to identify persistent segments by capability given the segment's address.

2.2.2 Orca

Orca [3] is a programming language and run-time system to program distributed applications. It implements the *shared data object* model and was tested on both multiprocessor and distributed architectures. Shared objects are replicated, with a given policy, on processors within the address space of specialized object managers (one per processor). User applications share memory directly with these object managers, which ensure one-copy semantics of objects. We can see this as distributed shared memory.

Object managers have a global single address space, providing a uniform and unique address space for the shared objects.

2.2.3 Monads

Monads [21] is a computer system that simulates a virtual and global shared memory in a network of workstations. Each MONAD-PC computer has an address translation unit that can swizzle on-the-fly long addresses (60 bits - 2^{48} pages of 4 kilobytes) to smaller main memory addresses. It allows the existence of 2^{32} objects identified by global virtual addresses and each address encapsulates a coded capacity. Each object can have a size of 2^{28} bytes.

2.2.4 μC++

μC++ [7] is a concurrent system, although not a distributed one; it executes on shared memory uniprocessor or multiprocessor computers. μC++ maps objects directly on virtual memory and, like the systems presented above, it uses a single address space model, that is, parallel μC++ applications are spawned in UNIX processes that share the same and complete address space.

2.2.5 Amber

The Amber system [9] spawns applications on a set of Firefly processors that share a common global address space, so that each object is assigned a different and unique virtual address. Each object, although shared by a number of processors, will only exist on one processor and will be remotely invoked by processes running on the others; to forward object invocations there are object representatives on each client processor (this is the *proxy* [30] mechanism). *Proxies* exist for each object at the same virtual address. So, Amber preserves a global address space for parallel applications without using distributed shared memory.

2.2.6 MMK

MMK [7] [4] uses an Amber-like methodology with a uniform address space, where objects have only one copy and remote procedure calls (RPC) are employed to perform remote invocations just like the Amber system. It runs on a iPSC/2 Hypercube of 32 processors.

2.2.7 ARCADE

The ARCADE system [11] offers a kernel interface to allocate and share memory between applications through explicit mention of the memory addresses of data units that are to be shared; data units are identified by address. Direct pointers into data units can not exist; a special system pointer *data unit link* exists in order to build dynamic data structures. Data units are associated with the semantics of what is stored inside the moment they are created. There is a mechanism that uses this information to transform data between different heterogeneous representations thus providing a mechanism to have heterogeneous distributed shared memory. It is not an object oriented system and our interest resides in the fact that distributed data structures can be built with ARCADE.

2.2.8 PLATINUM

PLATINUM [13] is an operating system kernel for NUMA multiprocessors and implements the abstraction of *coherent memory* which can be accessed uniformly by all processors in the system. It is interesting to note that the PLATINUM *coherent memory* assures the same memory object mappings throughout the different processors and thus also implements global and unique address spaces for shared data.

2.2.9 Linda

Linda [17] is not a traditional system nor a complete language. It is a set of objects and operations that are intended to be injected in another language, so it can be considered a set of low level mechanisms to add distributed programming to a language. It provides a tuple-space which is a global content-addressable shared memory to support distributed data structures. Kaashoek et al. [20] conclude that even if Linda provides more support to build distributed applications than traditional message passing or shared-variable models, it is still unclear how to build distributed data structures with a language that seems to be too low-level.

Linda applications share global data structures in a distributed environment in an address dependent way.

[7] MMK stands for Multiprocessor Multitasking System

2.2.10 Conclusion

To summarize, Clouds, Monads, Amber, Orca, MMK and μC++ all have a uniform and single address space for shared objects. Clouds and Orca use distributed shared memory and μC++ local shared memory. Amber and MMK provide one-copy semantics of objects with proxies. Clouds is the only persistent distributed system overviewed here.

2.3 Models for persistency in object oriented systems

Object oriented persistent programming avoids the use of an external persistent representation of objects and the conversion between both formats. Persistent systems have to deal with object identification, both internal and external, and thus with object pointers. Ideally, persistent and volatile structures do not differ at all because, if they do, pointers have to be swizzled [34] between in-core and passive formats. The system can still avoid pointer swizzling if objects occupy permanently a piece of virtual memory if their identification is address dependent, but in this case the total system memory available is limited by the virtual machine memory size. This is indeed the case with C++ where objects are identified by a virtual memory pointer. So, if all persistent objects are global, the user sees its address space reduced to what is provided by the memory management unit of the processor that runs the application. This is, for example the case of Clouds and an implementation of Napier [26].

The address space provided by a context is too small for a persistent system with a single level store because persistent data can be of the order of magnitude of secondary storage. Several solutions to this problem have been proposed.

2.3.1 Second level storage

In our previous *COOL v1* system [19] we experimented with explicit context persistency and object persistency. Object persistency was supported with a basic store and a load mechanism that, coupled with language level relocatable pointers, allowed us to break the binding between objects and address spaces. In addition, we allowed objects to be migrated between address spaces, in the same or different sites, with object relocation when needed [23].

This method can be used in a transparent manner by the use of pre-processors [29], inserting secondary level store/retrieve commands whenever persistent objects are used.

2.3.2 Name-based addressing of objects

Gehringer's model [16] adopts a name-based mapping where objects are referenced by name. He defines modules of objects (or groups of objects). Each module is an address space by itself and objects in different modules inter-reference themselves by name rather than by address. The object space for each process using this scheme is essentially unlimited. As we will see later, the *persistent context space* model also allows multiple separate address spaces but in a manner where objects do not need to be address independent.

2.3.3 Capability-based addressing

Buhr [6] proposes an interesting model to mitigate persistent and volatile data, structuring object name spaces like a file system hierarchy; objects can reference themselves also by name. He proposes a segmented architecture with low-level segment support possibly with Intel iAPX*86 microprocessors. Objects are made accessible within an address space through paging. There is no need for fixed address allocation because references between objects in the same segment are always relative and inter-segment references are name based; also, segments can be mapped at any address. Although, with this model, objects can not have direct pointers and this solution requires hardware support..

If pointers to objects are the problem, a different approach is the use of pointer swizzling [34] to convert between in-core small object pointer formats and persistent long formats. The later can be seen as a capability for persistent passive objects. The shortcoming of this approach is the need to convert always all object pointers whenever persistent objects are to be used. Even if these operations can be clustered to reduce overhead [33] we think that maintaining separate persistent contexts can allow valid direct object pointers without the need for external format conversion and diminish the number of these operations.

2.4 Direct addressing in a global address space

A frequently used approach in persistent systems is a single address space for all persistent objects and the consequent use of direct object memory pointers as object identifiers (arguments in favor of the single address space approach can be found in [21]). Trying to access passive objects on not yet mapped memory pages produces a page-faults that will load directly the right object at the right address. An example of such a system is an implementation of Napier [26] developed by Vaughan et al. [32] over the Mach [1] microkernel.

Napier is a persistent type system developed by the PISA project. It consists of a language and a persistent store and executes on a persistent abstract machine (PAM) [12]. Objects in this implementation of Napier are assigned permanent virtual addresses and persistent applications execute against a single global address space.

3 The persistent context space model

The *persistent context space* model is intended to be used by systems that need to manipulate address dependent data in a distributed and persistent environment. It can be used in an object oriented system if we consider objects as data. The *persistent context space* model is composed of:

- **persistent contexts** to provide persistent data and control address allocation of persistent data inside an address space of virtual memory,
- **context spaces** to provide distributed shared data coherence and maintain address allocation coherence between a collection of contexts that support parallel processes and share data.

Data is structured in units that are shared and turned persistent as a whole.

In this section we define the basic component of the model - *complete data space*, and the extension of contexts with two orthogonal properties, persistence and distribution, giving place respectively to the *persistent context* and the *context space* abstractions. The complete model, the *persistent context space*, deals with both.

3.1 Persistent complete memory spaces

Data is mapped in memory at arbitrary addresses. Mapping persistent data enhances traditional context memory with the property of persistence. *Persistent memory* outlives the scope of its creators and persists from on program activation to the next. There is the problem of the coherence of persistent memory because it depends on the semantics of applications. For example if an object is turned persistent and points to another object, then the second object has to be saved as well. One can assume that all data of an address space is persistent or try to create smaller persistent elements in order to break the limitation of a single address space and still be able to support direct object pointers. This control can also highly benefit garbage collection algorithms [18]. Assuming that objects are address dependent, it is mandatory that all references maintain its meaning between different applications and on the activation of the same data structures.

An object ensemble has references between its objects, and is considered *complete* if there are no unresolved references, that is, if all referenced objects are virtually present in the ensemble. This object ensemble defines *complete data*. When mapped into an address space it becomes a *complete memory space*.

An application which has unresolved references can not be guaranteed to execute correctly; an application with no unresolved references is said to execute in a *complete memory space* of persistent and complete data.

3.1.1 Structuring complete data: *clusters* and *containers*

Complete data is structured with the system notion of *container*. In this respect we follow Eos [18] terminology.

In [6], *name spaces* of objects inside segments are structured like a traditional file hierarchy for the following reasons:

- Segments can be tailored to accommodate one or more objects according to user needs. This raises the notion of object clustering.
- The model needs to enable the construction of *active memory trees* of segments that execute applications with the right objects; this is already captured with the notion of *container*.

In order to support the need for object clustering and to improve system efficiency, by reducing page-faults and increase the efficiency of in-core memory use [31], we organize *containers* in a collection *clusters* (also like Eos). A *cluster* is a set of *segments* and a *segment* is a linear

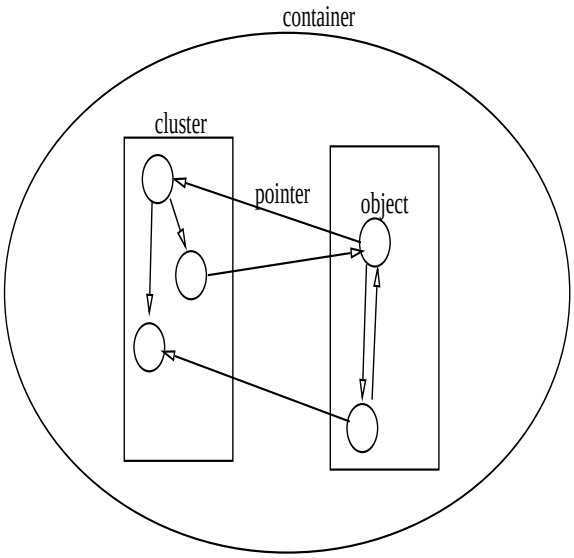

Figure 1: A container with two clusters

sequence of bytes and is persistent. A *segment* can be part of only one *cluster* and is always bound to an address. *Segments* are considered the smallest data unit at system level, but *clusters* are the smallest data unit that can be be shared and turned persistent in a single operation.

Direct references between data in different *containers* is not allowed (between different *containers* objects reference each other through representatives - proxies). An example of one *container* with two *clusters* having several objects and object references is depicted in figure 1.

An application needs at least one mapped *container* in order to be executed. When mapped, a *container* becomes a *complete memory space*. The limit of this model, regarding persistent data sizes for applications, is that each *container* has to be smaller than the maximum size of an address space for the target machine. If applications need to work with greater data sizes then the address space of the virtual machine implemented by the operating system in a context, then data has to be divided into smaller *containers*.

Applications can also share data, using the same *containers*. They can not share a simple open structured *cluster* that supports data that is not topologically closed because at least one of the applications would not execute on a complete data space (the cluster would have data pointing directly to the other application's data).

The smallest unit of intersection between two *containers* is in fact another smaller *container*. The smallest *container* has a single *cluster* with a single *segment*. *Containers* are similar to segments of the model proposed by Buhr [6]. In his model, references between objects in different segments are global and make explicit mention of segments; this is supported at language level (in the *persistent context space* model this is made with object representatives).

With the *persistent context space* model there is no need to organize objects in name

spaces. When mapping a *container*, objects that exist within the *container* are virtually mapped and their direct pointers are valid without the need for pointer swizzling [34]. Moreover, *clusters* can tailor objects in sensible sized units for object sharing, persistence, and locality of related objects.

3.2 Persistent contexts

Like the models of Buhr and Gehringher presented in section 2, we also think that the best solution is to provide multiple persistent address spaces. It is simpler to adapt a programming system based on C++ if we can keep address dependent object identification. We want to allow direct object pointers at system level and retain the multiple address space environment model at the same time.

A *persistent context* is a simple context that has one or more *containers* mapped into its address space. A *persistent context* sees the address space of persistent memory provided by a virtual machine and assures coherent address allocation for persistent data, so that objects occupy always disjoint address spaces. Multiple *persistent contexts* can exist on the system executing on different virtual machines exactly like common contexts. Persistent data retains statically allocated addresses and suffers complete relocation if moved inside the *persistent context* (this is possible for complete data units). This guarantees the validity of low level direct object pointers (or identifiers) for all *containers* mapped into a *persistent context.*

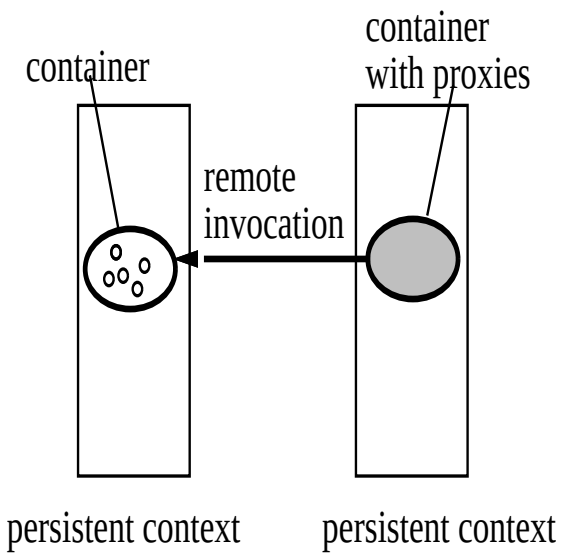

Figure 2: Remote invocation between 2 persistent contexts

Applications running on different persistent contexts can cooperate distributedly with the two following models:

- **Message passing** Applications can communicate explicitly, or the system transforms direct object addresses in an address space independent scheme like capability-based

addressing (by the use of the proxy mechanism [30]). Figure 2 draws an example of object invocation using message-passing with two persistent contexts.

- **Memory sharing** *Containers* in different *persistent contexts* can have smaller *containers* in common. The same *container* can occupy different address in each different *persistent context* so it has to be shared exclusively in different periods of time and relocated each time it is mapped into a *persistent context*. This is viewed as transparent *container* migration between persistent contexts and is possible because all information regarding a *container* is present inside the *container* itself being used by the system to perform relocation (like virtual memory that is only virtually present).

3.3 Context spaces

In traditional programming systems, shared memory is mapped simultaneously into address spaces of different contexts, normally at the same addresses; in most situations the shared information is in fact address dependent.

An application runs in different contexts if it is distributed across different sites and wants to take advantage of increased parallelism and suffer different protection policies (in protected address spaces of different contexts). In this case, the application has to communicate information around with the traditional message passing mechanism or by sharing memory directly (distributed shared memory is implemented with message passing [25]). Applications can work in different virtual machines (local or remote to each other) and share memory int separate address spaces within what we call a *context space*.

A *context space* is a set of contexts that are willing to share potentially all memory they can address. The processes executing in the virtual machines that are part of a *context space* will see only a single and uniform address space. The granularity of memory shared between them is a *cluster*.

All systems presented in section 2 have a single *context space*. We want to be able to create multiple *context spaces*.

3.4 Persistent context spaces

A *persistent context space* mitigates both *persistent contexts* and *context spaces*.

A *persistent context space* is a set of distributed *persistent contexts* that are able to share a single uniform address space where one or more *containers* can be mapped. Figure 3 depicts a *container* mapped in two *persistent context spaces*, one with three contexts and another with one context only.

In this model, a persistent application is executed in parallel by a number of processors that have contexts in one or more sites with a common global address space of persistent memory. Within a *persistent context space* there is the guarantee of the uniqueness of address allocation for shared persistent objects.

Several *persistent context spaces* can exist and share the same data in parallel. This data is assured access coherence and address allocation coherence. *Containers* migrate between

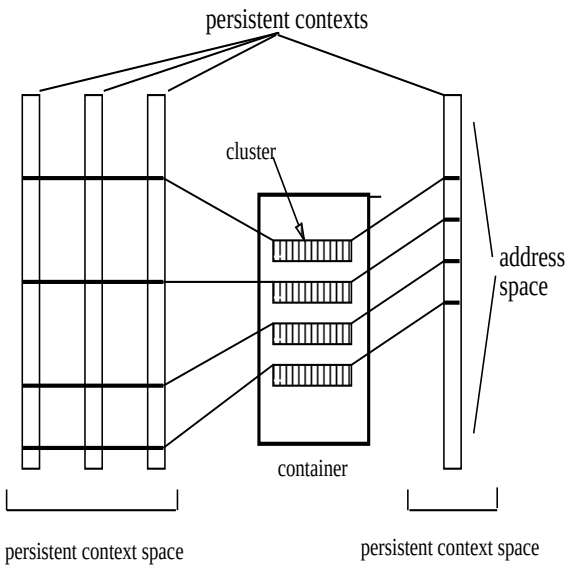

Figure 3: Persistent context space example

containers on different *persistent context spaces* being unmapped from one *persistent context space*, mapped in the target and relocated. If this migration degrades the system performance there are two possibilities to react in favor of a better configuration:

- use proxies to access remote objects; if objects are uses intensively this is a good approach [5];
- join the *persistent contexts spaces* into a single one; the decision can be taken at user level, by the user itself or by the administrator, or the system can react automatically. This choice depends on the run-time system, on the user interface and on the implementation of the model.

4 Operating system requirements to support the *Persistent Context Space* model

The model presented above was implemented in *COOL v2*. Instead of presenting the *COOL v2* architecture and the implementation of the model we will concentrate on the mechanisms that an operating system ought to have in order to support *persistent context spaces*. System needs to implement the model relate mainly to memory management at various levels:

- memory persistency;
- memory clustering and structuring, i.e. *clusters* and *containers*;
- *cluster* mapping into contexts and relocation;
- mutual exclusion control when the same *container* is used simultaneously at different addresses;

- and shared memory coherency.

Each functionality above needs a set of system services and key architecture particularities.

4.1 Persistency support: mappers and secondary stores

Persistent memory is organized in *containers* as explained above. Each *container* is further subdivided in *clusters*; a *cluster* being a set of persistent *segments*.

All the entities mentioned above should be capability-based addressable. This can be done via special mappers (segment, *cluster* and *container* mappers).

Segments are persistent and need to be saved on secondary stores; this is also true for the information on *clusters* and *containers*. Mappers need to address directly or indirectly persistent storage devices. Capabilities identifying these persistent entities also have to be unique during the lifetime of the entity they represent.

4.2 Amalgamation of complete memory in *containers*

In traditional systems, virtual memory is allocated explicitly. In computing systems with persistent virtual memory that memory persists once allocated until explicitly deleted.

The system has to guarantee memory allocation growth in *clusters* and *containers* (that define complete memory as explained in section 3). It all starts with an empty *container* that consists of an empty *cluster* with a segment of null size; then, each time memory is allocated it is appended to that *cluster* within the *segment* (or a new one).

The system has also to provide the user with tools choose on which *cluster* memory should be allocated; the system assumes always a default *cluster*. This information is of fundamental importance because, during run-time, the operating system has to know if direct pointers between two *clusters* can be used, if not *proxies* are installed instead.

This functionality to manipulate agglutination of memory has to exist at the language interface.

4.3 The exception mechanism for mapping

An application starts to run within a first process in a context. The first step is to establish the persistent complete memory space that is to be activated (one or more *containers*). This memory can change, of course, during application execution.

An exception mechanism should exist to produce memory faults that will map the correct memory at the right place. The first *container* mapping can be considered the highest level fault. It makes visible the next level in the structure, that is, *clusters*.

The second exception level is the segment fault: upon an access to some unmapped memory address, the exception handler verifies if there is some existing *cluster*, part of the current complete memory space, that contains a segment with the needed address when

mapped; it then maps the right *cluster*. Finally, in a page-based architecture, each segment is divided in pages, so it is only effectively read to in-core memory if it is really accessed (in a third level fault).

After mapping, memory may need to be relocated. This is dependent on the semantics of memory contents and only application levels are aware of it. The relocation itself is based on symbolic information and only known symbols can be relocated. The problem is that there may be pointers that have no correspondent symbols generated normally by the compilation chain, so, special high level run-time code has to exist in order to access intrinsic semantic information of memory contents at user-level in a transparent manner.

The run-time system can relocate *clusters* through all the symbols it finds regarding a particular *cluster*.

4.4 The upcall mechanism for unmapping

The upcall mechanism [10] can also be considered an exception (but a distributed one). As we saw, *cluster* mapping and relocation is done automatically by the run-time system at memory fault. But to produce a memory fault, and thus oblige relocation on some *cluster* within a particular *persistent context space*, there is a need to unmap it first from another *persistent context space* because it could be in use at different addresses.

An upcall has to be issued from the kernel to the run-time system in order to unmap the *cluster* transparently from the application contexts. This upcall can be performed with a message passing mechanism to allow distribution.

4.5 Combining exception and upcall mechanisms to assure mutual exclusion

With the exception and upcall mechanisms in place it is straight forward to assure mutual exclusion of *clusters* that need to be mapped at different addresses, i.e., that belong to different active *persistent contexts spaces*.

During memory fault handling, if the system sees that a *cluster* is being used by another *persistent context space* it upcalls all contexts in that *context space* to force the unmapping, and proceeds. Later on, if any one of the other contexts needs that *cluster* again, it will do exactly the same in the inverted sense.

After remapping a *cluster*, the system has to verify if the *container* information on that *cluster* is still valid. The *container* may have a different set of *clusters* changed by the *context* of another *persistent context*. This has to be done immediately after *cluster* mapping because it may now reference directly another *cluster* after being changed in the *persistent context space* where it was previously mapped.

4.6 Shared memory coherency

Memory mapped in a *context space* needs to be assured single-writer multiple reader coherence between all distributed contexts that have it mapped into its address space. A distributed shared memory system such as proposed by Li [25] should be used.

5 Conclusion and future work

We strongly believe that future parallel computing systems will evolve towards simpler user distributed environments. With *COOL v2* we accomplished a persistent computing system with a centralized object oriented interface.

One of the problems in distributed and persistent operating systems is the transparent use of shared data. In *COOL v2*, we have developed and implemented the *persistent context space* model to control persistent memory and distributed shared memory allocation. Distributed shared memory mechanisms are used to control data coherency. We have presented related work on the subject, the reasons that led us to create the *persistent context space* model, and the model itself.

We are currently experimenting further with *persistent context spaces*, studying memory migration between separate *persistent contexts*, and separation/mitigation of *persistent contexts* based on *containers*. The key issue is the overhead associated with these operations, which are necessary in order to keep the system running. We want to find out which configurations are appropriate to which applications and a way to evolve automatically towards configurations with better performance. Also, extending C++ with *container* semantics is still matter of research in the *COOL v2* project.

The first *COOL v2* prototype is currently running on a network of 386 based workstations running the CHORUS micro-kernel.

6 Acknowledgments

We would like to thank Marc Guillemont and Peter Strarup Jensen for their helpful comments on this paper.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: A new kernel foundation for UNIX development. Technical Report CMU-CS-87-150, Carnegie Mellon University, Pittsburgh, PA, USA, August 1986.

[2] M.P. Atkinson. Programming languages and databases. In *Proceedings of the 4th VLDB*, 1978.

[3] Henry E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. A distributed implementation of the shared data-object model. In *Proceedings of the Workshop on Distributed and Multiprocessor Systems (SEDMS)*, Fort Lauerdale FL, USA, October 1989. USENIX Association.

[4] Thomas Bermmerl, Hubert Ertl, and Thomas Ludwig. A multiprocessor operating system kernel offering dynamic global objects for distributed memory multiprocessors. In *Proceedings of the IFIP Workshop on Decentralized Systems*, 1989.

[5] Gordon S. Blair and Rodger Lea. The impact of distribution on the object-oriented approach to software development. *IEE Software Engineering Journal*, 7(2), March 1992.

[6] P. A. Buhr. Addressing in a persistent environment. In *3rd Workshop on Persistent Object Systems*, Newcastle, Australia, 1989. Springer-Verlag.

[7] P. A. Buhr, Glenn Ditchfield, R. A. Stooboscher, and B. M. Younger. μc++: Concurrency in the object-oriented language C++. *Software Practice and Experience*, 22(2), February 1992.

[8] K. Chandy and C. Kesselman. Parallel programming in 2001. *IEEE Software*, November 1991.

[9] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, USA, December 1989.

[10] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180. Association for Computer Machinery, December 1985.

[11] D. Cohn, P. Greenawalt, M. Casey, and M. Stevenson. Using kernel-level support for distributed shared data. In *Proceedings of the Workshop on Distributed and Multiprocessor Systems (SEDMS II)*, Atlanta USA, March 1991. USENIX Associaton.

[12] R. Connor, R. Carrick, A. Dearle, and R. Morrison. The persistent abstract machine. In *Proceedings of the 3rd Workshop in Persistent Object Systems*, Newcastle, Australia, 1989.

[13] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with platinum. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park USA, December 1989.

[14] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and R. Chen. Distributed programming with objects and threads in the Clouds system. *Computing Systems*, 4(3), 1991.

[15] P. Dasgupta, R. Chen, S. Menon, M. Person, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc, W. Appelbe, J. Barnabeu-Auban, P. Hutto, M. Khalidi, and C. Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1), 1990.

[16] Edward F. Gehringer. Name-based mapping: Addressing support for persistent objects. In *Proceedings of the 3rd Workshop in Persistent Object Systems*, Newcastle, Australia, 1989. Springer-Verlag.

[17] David Gelertner. Multiple tuple spaces in linda. In *Proceedings of PARLE*, 1989.

[18] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez. EOS, an environment for object-based systems. Technical Report 1499, Institut National de la Recherche en Informatique et Automatique, September 1991.

[19] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: Kernel support for object-oriented environments. *SIGPLAN Notices*, 25:269–277, 1990.

[20] M. Kaashoek, H. Bal, and A. Tanenbaum. Experience with the distributed data structure paradigm in linda. In *Proceedings of the Workshop on Distributed and Multiprocessor Systems (SEDMS)*, Fort Lauerdale FL, USA, October 1989.

[21] James L. Keedy and John Rosenberg. Support for objects in the monads architecture. In John Rosenberg, editor, *Proceedings of Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 202–213, Newcastle, Australia, January 1989.

[22] Richard Larowe Jr. and Carla Schlatter Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Transactions on Computer Systems*, 9(9), November 1991.

[23] Rodger Lea and James Weightman. Supporting object oriented languages in a distributed environment: The COOL approach. In *Proceedings of the 5th TOOLS Conference*, pages 37–47, Santa Barbara, USA, 1991. Prentice Hall publishing.

[24] P. Leach, P. Levine, B. Douros, J. Hamilton D. Nelson, and B. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas on Communication*, 1(5), November 1983.

[25] Kay Li. Ivy: a shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, St Charles, IL, August 1988.

[26] R. Morrison, A. Brown, R. Carrick, R. Connor, and A. Dearle. The Napier type system. In *Proceedings of the 3rd Workshop in Persistent Object Systems*, Newcastle, Australia, 1989. Springer-Verlag.

[27] Maria Ines Ortega, François Armand, and Vadim Abrossimov. Coherent distributed shared memory on to the CHORUS virtual memory system. In *Proceedings of the Workshop on Distributed and Multiprocessor Systems (SEDMS III)*, Newprot Beach, CA, USA, 1992. Usenix Association.

[28] D. Pitts and P. Dasgupta. Object memory and storage management in the Clouds kernel. In *The 8th International Conference on Distributed Computer Systems*, S. Jose USA, June 1988.

[29] Joel E. Richardson and Michael J. Carey. Implementing persistence in e. In *Proceedings of the 3rd Workshop in Persistent Object Systems*, Newcastle, Australia, 1989. Springer-Verlag.

[30] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proceedings of the 6th ICDS conference*, May 1986.

[31] James W. Stamos. Static grouping of small objects to enhance performence of a paged virtual system. *ACM transactions on Computer Systems*, May 1984.

[32] Francis Vaughan, Tracy Schunke, Bett Koch, Alan Dearle, Chris Marlin, and Chris Barter. A persistent distributed architecture supported by the mach operating system. In *Proceedings of Mach Workshop*, pages 123–140, Burlington, VT, USA, October 1990. Usenix.

[33] Paul R. Wilson. Operating system support for small objects. In *Proceedings of the First International Workshop in Object Orientation on Operating Systems*, Palo Alto, CA,, USA, October 1991.

[34] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the Second International Workshop in Object Orientation on Operating Systems*, Dourdan, France, September 1992. IEEE Computer Society.