Lightweight Hardware Support for Protection in Object-Oriented Systems

Jörg Kaiser, Karol Czaja

German National Research Center for Computer Science
Postfach 1316, Schloß Birlinghoven
e-mail: kaiser@gmdzi.gmd.de

Abstract:

The paper describes ACOM (Access COntrol Monitor), a hardware device which we developed to enforce run time protection in an persistent object-oriented system. To obtain a wide acceptance, the efficiency of these systems must be comparable to conventional language systems. One of the key issues is to exploit the efficiency of virtual memory management of contemporary processors. We will argue that a careful analysis of the hardware-software trade-off will lead to a simple hardware device which can efficiently support encapsulation and protection of small objects in an object-oriented systems. The main idea is to separate encapsulation and protection from address translation issues.

1. Introduction

The object-oriented programming paradigm maps real world problems into a universe of objects in a machine. Ideally, everything a user of an object-oriented system is concerned with are objects. The system should provide a uniform interface to objects and remove the classical distinction between program-variables, files, or database items.

A number of research and commercial projects in the area of object-oriented operating systems tried to provide objects as a general abstraction at the user interface [1],[2],[3],[4],[5],[6],[7],[8] without assuming any specially designed hardware platform. Of particular interest are those approaches which do not distinguish between the object model of the language and the system [5],[6],[7],[8],[9]. This has the following consequences on the support system:

- the entire application is structured in arbitrarily sized objects. This means that the size of the objects is determined by the application and not by artifacts of the system architecture. Particularly, the system must cope with a large number of small objects as well as with very large objects.
- individual objects should be the entities of protection and sharing. This implies that the architecture must recognize and protect those objects.
- the system should directly support generic functions only, i.e. the least common denominator of all languages in question. This means that the system basically provides the containers for language objects, maps and protects them.

In the following, we will concentrate on this basic functionality of an object support system. We will argue that a careful analysis of the hardware-software trade-off will lead to a simple hardware device which can efficiently support encapsulation and protection of small objects in an object-oriented operating systems. The main idea is to separate encapsulation and protection from address translation issues. The paper is organized as follows:

In the next section we briefly sketch two examples of systems supporting persistent objects to show how these systems implement the persistent store on a pure software basis. We will argue that basic protection issues cannot efficiently be solved in these systems. The rest of the paper describes ACOM (Access Control Monitor), a hardware device which we developed to enforce run time protection. It easily could

0-8186-3015-9/92 $03.00 © 1992 IEEE

complement object-oriented persistent systems shown in the examples. Since only the most basic functions of encapsulation and protection are incorporated into the design, leaving the more complex and language dependent issues to software, it can be seen as a RISC approach to object-oriented hardware support.

2. Representation of persistent objects

To exhibit the benefits of architectural support, we examine two example systems which provide basic support for persistent objects. We will concentrate on the Comandos system [7] and on an approach developed by Wilson [10], although many other language and database systems use similar techniques [11],[12],[13]. We chose Comandos because it is a complete implementation of an object store addressing language and system aspects. Wilson's approach is sketched because he elegantly exploits available address mapping mechanisms to implement a persistent store. Both systems do not rely on special purpose hardware. The conceptual view of the persistent store in both systems is outlined in Fig.1. Both systems provide a shared persistent object store which includes all devices of a storage hierarchy. The system shields the programmer from the different addressing mechanisms found in the distinct storage media and allows a uniform location independent access to objects. The persistent object memory is constructed from a persistent passive space and a transient active space. The passive space is the long term object repository. Each persistent object has a representation in passive space. The active space constitutes a virtual address space where objects are directly accessible by a machine dependent address and where computations on objects are performed. However, for a programmer and even for a running program, the distinction is transparent and hence, conceptually, a single level store is provided. If a persistent object is referenced and it is not in the active space, it is automatically transferred from the passive to the active space by an appropriate manager.

Once in active space, it should be possible to operate on objects as conveniently and with the same performance as in the runtime environment of a language. This particularly means, that it is mandatory to fully exploit all the hardware facilities of the basic processor, especially, virtual memory management and address calculation. The overhead one has to pay for persistence should only occur on the activation and passivation of objects. With a sufficiently large (machine supported) virtual memory and a certain locality of computation, acceptable performance can be expected [10]. Therefore, we assume that the active space relies on a paged virtual memory because this is the standard supported by common address translation hardware and operating systems.

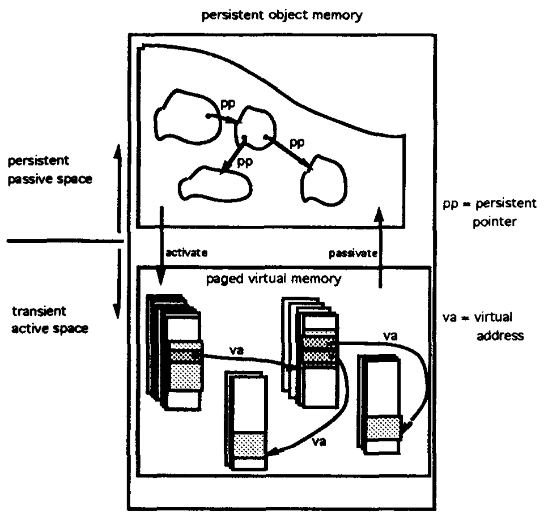

Fig. 1 Structure of persistent object memory

The following steps have to be performed to bring in an object from passive to active space. Firstly, the system must detect that a referenced object is not in active space. Secondly, the object has to be brought in, thereby converting its passive to an active representation. This conversion mainly affects persistent pointers which have to be transformed to virtual addresses, also termed transient pointers. This mapping is dynamic because the relationship between persistent and transient pointers is not fixed but determined at translation time and partial in that not all persistent pointers are mapped to virtual addresses. The technique of having multiple namespaces and translating pointers is known as pointer swizzling or pointer resolution and implementations exist in persistent languages e.g. [11] databases e.g. [12] on the OS-level and on the architectural level e.g. [13]. Thirdly, the objects, now in virtual memory have to be protected according to their specified protection attributes.

Since the movement of individual small objects from passive to active space would be too expensive, objects are grouped to larger entities for activation. If locality of computations is assumed within these entities, this can also be viewed as a look-ahead technique for activation. In Comandos, the notion of a *cluster* [7] is introduced comprising objects which according to some grouping policy belong together. When an individual object is activated, all objects residing in the respective cluster are mapped into virtual memory, i.e. space is reserved in virtual memory for the entire cluster by updating the corresponding entries in the page translation tables. Copying of data to the active space then proceeds on demand in entities of pages. In Wilson´s approach, the entire persistent space is a huge linear paged address space. The entity which is transferred to active space on object activation is a page.

Starting with the detection of a reference to an object which is still in persistent memory, we can classify different approaches. In the Amadeus implementation [14] of the Comandos system it is assumed that an access to an object always takes place via an object invocation. If an object is brought to active space, all its persistent pointers are resolved. The object may contain pointers which address some other object not yet in active space. To cope with this situation, a so called *proxy* is inserted in place of the pointed-to object. When a subsequent invocation uses the address of the proxy, the code of the proxy is executed which initiates the transfer of the associated persistent object to active space. The important points are that an executing program never sees a persistent pointer and that a resolved pointer when used, really addresses the right object. It should be noticed that the detection of a proxy relies on the proper use of the invocation mechanism.

Another possibility, also developed in the Comandos project, is to replace a pointer to an object which is not in active space by an invalid address to cause a hardware trap if this pointer is used. The handler then has to determine which object should be addressed and subsequently move it to active space. This approach allows to access objects directly using normal pointer arithmetic additionally to the invocation mechanism (this is bad style but possible e.g. in C++). However, since the invalid address does not contain location information, effort has to be devoted to determine the respective object.

Wilson exploits the trap facility of the page translation mechanism to detect accesses to unmapped persistent objects in active space [8]. If a page holding one or more persistent objects is faulted into active space, all pointers of the page are resolved. As a consequence, all pointed-to pages have to be mapped in active space i.e. the corresponding space has to be reserved. Since these pages may contain persistent pointers they have to be access protected. This assures that a running program cannot see persistent pointers. If a program attempts to access a protected page, a trap handler is invoked which copies the page into active memory and translates all persistent pointers into transient pointers, again relocating the referred-to pages as needed.

These approaches show that the problem of detecting a reference to a persistent object in passive space and the resolution of pointers can be solved on a standard hardware platform with acceptable performance.

However, once in active space, there is no way in conventional, page-based systems to individually protect the subpage-objects from inadvertent accesses and hence, assuring the reliable and secure operation of the system [15]. In Comandos, protection of objects inside a cluster is enforced by a programming convention rather than by a mechanism provided by the system. It is possible to generate a virtual address without using the invocation mechanism properly, thus, compromising system integrity. Therefore, it is recommended that only those objects are grouped in a cluster which mutually trust each other. This restricts the freedom of grouping policies and may result in additional overhead to relocate objects. One of the great advantages of the Comandos system is that object sharing is supported. In the COOL-2 [6] kernel, which provides basic support for cluster objects and constitutes a lower level component of the Comandos system, an object can concurrently be mapped into many distinct virtual address spaces for efficiently sharing one object representation. This desirable feature however is questionable if it is not possible to map objects into distinct address spaces with different protection attributes. The current solution is that in cases where this is required, a critical object can only reside in a single address space. To access the object from another address space an invocation crossing address space boundaries has to be performed. This, of course is a workaround and an expensive solution.

Wilson proposes a solution for cases where sensitive objects happen to reside on the same page as non-sensitive objects. In this case the off-limit object should be replaced by a "bogus proxy" which is made unusable. This, of course, is no solution for controlled sharing where e.g. one process is allowed to read and write an object while others are only allowed to read it.

To summarize: while the addressing problem of persistent object systems seems to be acceptably solved by the above schemes, protection is still an open problem. Although the need for protecting individual objects may be obvious, the lack of it or the inadequate solutions are the price most designers are willing to pay in favour of running their software on a standard hardware platform. In the following sections we will present ACOM, a simple hardware device which addresses the protection problem. In its design, much emphasis has been placed on easy integration in existing hardware and software platforms.

3. Can existing address translation hardware be exploited ?

Typically, an object-oriented application is constructed from a large number of small objects. Existing hardware platforms like the Intel 386/486 [16] offer a segmentation mechanism which allows to specify and protect segments of arbitrary size up to 4 Gbyte. However, due to the size of the segment index, the number of segments which can be addressed in the protection domain of a task is restricted to 8k local and 8k global segments which may not be enough in object-oriented applications.
Another approach would be to provide small pages and place only one object on each page. This would trade space to gain protection. There are some MMUs (Memory Management Units) which support page sizes down to 256 byte (e.g. Motorola 68030 architectures [17]). This, of course, has a number of drawbacks starting with the larger number of pages which have to be maintained in a multi-level hierarchy of page tables (up to 5 levels using 256 byte pages in 68030 [17]). Traversing this hierarchy of page tables slows down the address translation mechanism. Secondly, because of the still coarse granularity and the fixed size of a page, the internal fragmentation may be substantial.

Because of the insufficiencies of these approaches we propose an architecture which provides protection without touching the address translation mechanism. As a result, the most efficient address translation mechanism can be chosen, optimizing page or segment size according to the need of the hardware devices. Objects can be arbitrarily grouped together on such entities for efficient memory management but they can be protected individually.

4. The conceptual view of ACOM

ACOM controls memory accesses without interfering with the address translation path of the processor, i.e. it checks memory accesses independently and concurrently to any existing address translation hardware. Because of this independence, ACOM can easily be integrated into any hardware platform. Only if an invalid access is detected by ACOM, an exception is generated to signal the violation. Since the detection of invalid addresses by ACOM is very fast in most processor systems, the memory access can be aborted before it overwrites or illicitly reads a memory location. If this is not possible, the trap handler has the responsibility of initiating corrective actions.
Fig. 2 shows a physical addressing path and indicates how ACOM is connected to the system. Today's processors exhibit a large variety of configurations concerning memory management units and caches. Most processors have these facilities on-chip. Therefore, to allow a universal application of ACOM, no assumptions about the physical structure of the processor should be made. In fact, ACOM can be integrated into a system regardless of the individual cache/MMU configurations. This topic is discussed in more detail in section 6. For the moment, it should be noted that any address, virtual or physical, applied to main memory uniquely selects a memory location.

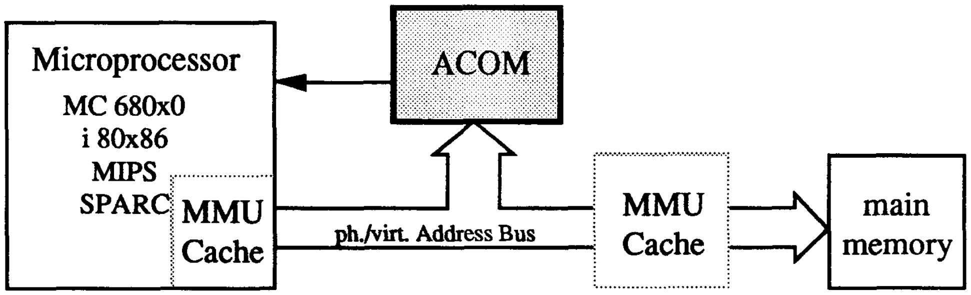

Fig. 2 Physical integration of ACOM

The subpage segment structure is superimposed on linear memory by ACOM. These segments are the guarded containers for objects defined at a higher level. For each such segment ACOM provides the corresponding access rights. ACOM monitors the address bus and and executes the necessary checks on the basis of the current address and the intended access (read/write/execute) of the processor which is also available during a memory access. A conceptual view of ACOM is presented in Fig. 3.

ACOM works much in the same way as a tagged memory [18] with the difference that ACOM logically defines the tagged architecture and substantially simplifies the management of tags. A tag comprising access rights is associated to the addressed memory location and evaluated with every access. This tag is stored in a separate memory the so called BMT (Block Map Table). For reasons of implementation efficiency, we assume small blocks of 8 or 16 words of 32 bits rather than provide a tag for each memory word. A segment then comprises a number of these blocks. For each block the specified access rights are derived from the protection state of the segment. It should be noted that the memory requirements for the BMT are very low. Assuming a block size of 64 byte and two bits per block to specify read/write/execute rights, a linear physical memory of up to 64 Mbyte can be supported by just two 1 Mbit memory chips. This is under 0.5% of the total memory hardware.

However, there are a couple of problems which cannot be solved in a straightforward implementation of a tagged memory concept. Firstly, the management overhead is unacceptable. Each tag in the memory has to be initialized and maintained. If we assume a standard page size of 4 kbyte, 64 entries have to be touched, independent of whether sub-page structures are needed or not. Even worse, in a multiprogramming environment, where the address spaces of multiple processes have to be isolated from each other, this hardly can be achieved by modifying the tags for almost the entire memory on each process switch.

Therefore, we distinguish between two kinds of pages in linear memory. *Linear pages* are not subdivided into smaller entities. *Cluster pages* contain multiple segments and are specifically supported by ACOM. As a second improvement, ACOM supports multiple address spaces efficiently. To achieve this, it comprises an additional lookup table, the page identity table (PIT). The PIT and the BMT are concurrently accessed during a memory cycle. The PIT allows the association of an address with the corresponding entries in the BMT. For each page it can be decided whether it belongs to the address space of a particular process or not. The PIT hardware is comparable to the hit/miss logic of a conventional direct mapped TLB [19]. The PIT determines whether an address refers to a page for which it already contains a valid entry and whether this entry refers to a linear page or a cluster page. If an linear page is accessed, nothing more has to be done. In case of referring to a cluster page, the corresponding tag of the BMT containing access rights is evaluated.

An additional advantage over tagged memory is the hardware entry generator of ACOM which creates the tags in the BMT for a segment automatically from the segment's base address and size. Thus, it eliminates the time consuming accesses by the main processor. The detailed

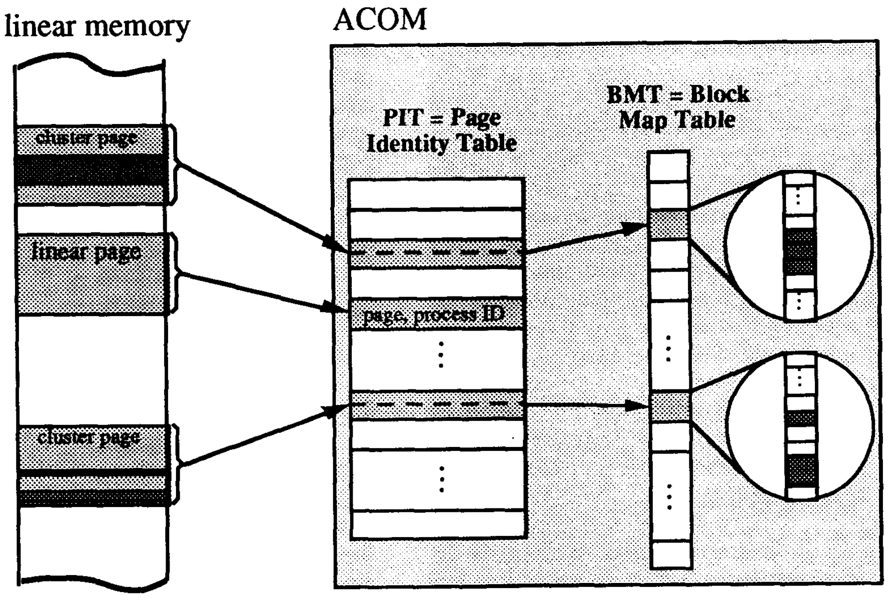

Fig. 3 The conceptual view of ACOM

description of the hardware architecture is beyond the scope of this paper. The reader is referred to [20].

5. Making the functionality of ACOM available to the application

The goal of ACOM is to provide encapsulation and protection for individual application level objects. Because of its generic functionality and its flexible design, ACOM can be embedded into an existing system in many different ways depending on the need of a specific application field. This may range from highly secure operation where intended malicious attacks to the system have to be considered to a debugging aid which can detect wrong pointer operations. In the latter case ACOM could freely be controlled by user level procedures.

In a secure protection scheme, the procedures and data structures controlling ACOM must be protected. If it can be assured that the tables of ACOM are not modified deliberately, ACOM will provide the basic fine grain protection, necessary to enforce security. The straightforward way to achieve security would be the migration of functions controlling ACOM to the operating system kernel. All functions could be executed in system space which is assumed to be protected from malicious accesses. This however would require a considerable change in the operating system, particularly, the notion of small objects must be introduced on the kernel level. An additional unacceptable overhead is the switch to the kernel level.

A more adequate way in respect of flexibility and easy system integration is to control ACOM from user space. ACOM is maintained by trusted procedures which run in user space and may be executed during an object invocation. The architectural support of ACOM to guarantee that only a privileged procedure accesses ACOM is the provision of a *key*. A key is a number which is stored in an internal ACOM register. This key can only be modified and written into the internal ACOM register by the operating system kernel. When the trusted procedures are loaded into memory by the kernel, the kernel writes the key to a dedicated slot within the procedure code. Subsequently, these procedures are "execute only" protected by ACOM. Thus, they now hold the key as local data which is not accessible by regular read or write operations. In the operation which load ACOM, this key must be presented and ACOM raises an exception if it detects a wrong key. The use of the key and the ability to protect small segments enable ACOM to enforce security with minimal kernel support.

Referring to the systems described in section 2, object fault or page fault time, respectively, is the right place to perform the necessary updates on the tables inside ACOM. At this point, the persistent pointers in an object or inside a page have to be resolved. To perform the pointer swizzling, the internal layout of an object or a page must be known. Because now, this information is available anyway, there is no overhead to additionally retrieve this information for setting the ACOM entries. Since the pointer swizzling is achieved in user space, it is highly advantageous that ACOM can also be maintained without switching to system space. The overhead of updating the entries for a segment is then reduced to two dummy read accesses as described below plus the time to internally update the entries by the hardware entry generator. Depending on the technology used we assume an overhead of about 20ns/entry. If we assume a mean object size of 256 bytes, we need four cycles resulting in a total time of about 80ns which is in the order of a single memory access.

6. Physical Integration of ACOM

As mentioned earlier, we have to consider a large variety of processor/cache/MMU configurations to achieve a wide applicability of ACOM. This involves a detailed analysis of memory access cycles as well as cache algorithms of different processors. The optimal solution for placement would enable ACOM to directly observe the virtual addresses generated by the processor. However, the use of on-chip MMUs and caches makes this solution impossible. The following discussion will give a flavour of the problems encountered.

On-Chip MMU

If the MMU is on-chip, ACOM can only observe physical addresses on the external bus. In a straightforward solution, the procedure which is in charge of loading ACOM with the appropriate segment attributes must know the physical segment address. This, however, requires support from the operating system kernel which currently is not available. In our approach to cope with on-chip MMUs no kernel support is necessary. We exploit the address translation of the on-chip MMU to load ACOM. We issue two subsequent dummy read operations indicating that ACOM now will be loaded. The information issued with these accesses comprises the key, the lower and upper segment bounds, and the corresponding protection state. The procedure which issues the dummy reads must only know the virtual addresses of the lower and upper bound, respectively. The MMU translates these addresses and ACOM can take the proper physical values from the bus.

Whenever a page is swapped out, ACOM has to invalidate the corresponding entry in the PIT and the PIT is reloaded when a new page is swapped in. When the new cluster page is swapped into physical memory all tags in the BMT are set to their proper values. This is performed by the protected procedures described above providing the base addresses, size information and protection attributes for the subpage segments. The BMT hardware entry generator sets the internal tables according to these values. The low overhead of these operations is described above.

On-Chip Caches

A more serious problem is the existence of on-chip caches since individual accesses on memory locations are invisible for ACOM if the items are cached already. We looked at many different caching strategies. It is well beyond the scope of this paper to discuss them all in detail. Therefore, we will address the basic problems only. ACOM can only control memory accesses when loading or writing back the cache contents from or to main memory, respectively. The cache is usually loaded and written back in terms of so called *lines*. Lines are of fixed size of a power of 2 (typically 16 bytes). Therefore, lines always fit into a sub-page block defined by ACOM and do not cross block boundaries. Consequently, all items in a line belong to the same block and have common protection attributes. Hence, controlling accesses which load the cache can easily be achieved by ACOM. Writing back the cache contents to main memory can be distinguished in two basic strategies. The *write-through technique* immediately transfers the modified item to main memory and hence, ACOM can directly control the access. The *buffered write-through* and the *write-back* strategies delay the transfer of modified lines. As a result, the detection of a incorrect access by ACOM is also delayed.

ACOM will indicate the access violation when eventually the cache contents is transferred to main memory. In this case, the damage may be more substantial and more complex recovery mechanisms [21] have to be applied. However, it should be noted that independent of detection latency, handling of a protection violation is difficult and needs assistance of higher system levels.

7. Conclusion

Persistent object systems try to hide the difference between language level objects and system objects. To obtain a wide acceptance, the efficiency of these systems must be comparable to conventional language systems. One of the key issues is to exploit the efficiency of virtual memory management of contemporary processors. We presented two approaches which follow this guideline and do not assume any specially designed hardware platform. Because, in these systems, controlled object sharing is highly desirable as an efficient mechanism for cooperation and communication, protection becomes a vital property. Since a fine grain protection scheme has to check individual accesses to objects, this can only be performed efficiently by hardware. However, the protection mechanisms of available high performance processors are tightly coupled with the address translation mechanism which, in these architectures is based on fixed size pages, inadequate to protect individual objects of arbitrary size.

We have developed ACOM, an architecture which provides protection for individual objects independently from any address translation issues. Separating protection from address translation results in a number of benefits:

- Exploitation of any high performance virtual memory implementation since ACOM does not interfere with the (critical) address translation path. Therefore, ACOM does not slow down memory accesses.
- ACOM can be securely controlled by user level trusted procedures. Hardware support is provided to check authority of these procedures.
- The overhead to maintain ACOM is very low. Updating ACOM is additionally supported by an entry generator.
- Easy integration in a conventional hardware platform. ACOM can be applied to systems with different hardware configurations, i.e. on-chip MMUs and caches.
- ACOM is a simple device in terms of hardware complexity. This will reduce hardware costs and make an implementation easy.
- Applications which do not need or want fine grain protection do not suffer from ACOM in terms of performance degradation or maintenance overhead. ACOM can be completely deactivated for these applications.

The paper sketches how ACOM can complement existing approaches to persistent object-oriented systems. The design of ACOM is ready to be frozen in silicon.

8. References

[1] P. Dasgupta, R.J. LeBlanc Jr., W.F. Appelbe: The Clouds Distributed Operating System Functional Description, Implementation Details, and Related Work, Tech. Report: GIT-ICS-87/42, GIT, Georgia, 1987

[2] G.T. Almes, A.P. Black, E.D. Lazowska, J.D. Noe: The Eden System: A Technical Review University of Washington Department of Computer Science, Tech. Report 83-10-05, October 1983

[3] A.S. Tanenbaum, S.J. Mullender, R. van Renesse: Using Sparse Capabilities in a Distributed Operating System, Proc. 6th int. Conf. on Distr. Computer Systems, IEEE, 1986

[4] M. Rozeir, V. Abrassimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, W. Neuhauser: CHORUS Distributed Operating Systems, Tech. Rep. CS/TR-88-7.8, Feb. 1989

[5] Y. Yokote, A. Mitsuzawa, N. Fujinami, M. Tokoro: Reflective Object Management in the Muse Operating System, in Proc. of the 1991 International Workshop on Object Orientation in

Operating Systems, October 1991, Pala Alto, California, IEEE Computer Society Press, Los Alamitos, California

[6] R. Lea, P. Amaral, Ch. Jacquemot: COOL-2: an object oriented support platform built above the Chorus Micro-kernel, Proc. of the 1991 International Workshop on Object Orientation in Operating Systems, October 1991, Pala Alto, California, IEEE Computer Society Press, Los Alamitos, California

[7] Comandos Consortium: A Guide to the Comandos Platform; Description of Comandos-2 Architecture Esprit Project 2071 - Deliverable D1-T2.2, March 1991

[8] P.R. Wilson: Operating System Support for Small Objects, In Proc. of the 1991 International Workshop on Object Orientation in Operating Systems, October 1991, Palo Alto, California, EEE Computer Society Press, Los Alamitos, California

[9] A. Black, N. Hutchinson, E. Jul, H. Levy: Object Structure in the Emerald System, Proc. 1986 ACM Conf. on Obj.-Oriented Progr. Systems, Languages and Applications, ACM, 1986

[10] P.R. Wilson: Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware, Computer Architecture News, June 1991

[11] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, R. Morrison: An Approach to Persistent Programming, The Computer Journal, Vol.26, No.4, November 1983, pp. 360-365.

[12] T. Andrews, C. Harris, K. Sinkel: The Ontos Object Database, Tech. Report, Ontologic Inc., Burlington, Ma, 1989

[13] W.P. Cockshott, M.P. Atkinson, K.J. Chisholm, P.J. Bailey, R. Morrison: POMPS: A Persistent Object Management System, Software Practice and Experience, Vol.14, No.1, January 1984, pp. 49-71.

[14] Trinity College Dublin: Overview of the Amadeus Project, Tech. Report, Distributed Systems Group, May 1991, Trinity College Dublin

[15] J. Kaiser: An Object-Oriented Approach to Support System Reliability and Security, Proc. "European Symposium on Research in Computer Security", Toulouse, France, October 1990

[16] Intel: 80386 Hardware Reference Manual Intel Corp., Santa Clara, California, 1986

[17] Motorola: MC68030 Enhanced 32-Bit Microprocessor User´s Manual Motorola Inc., 1987

[18] G.J. Myers: Advances in Computer Architecture 2nd Ed., John Wiley&Sons, 1982

[19] J. Kaiser: MUTABOR, A Coprocessor Supporting Memory Managent in an Object-Oriented Architecture, IEEE Micro, Vol.8 No.5, October 1988

[20] K. Czaja, J. Kaiser, U. Kleinhans: Ein Hardware-Monitor zur Durchsetzung von Zugriffsschutz in objektorientierten Systemen, In: A Jammel, Architektur von Rechensystemen, 12. GI/ITG-Fachtagung, Kiel 1992, Springer 1992

[21] J. Kaiser, E. Nett, R. Kröger: MUTABOR: A Coprocessor supporting Object-Oriented Memory Management and Error Recovery, Proc. HICSS-21, Vol. 1, 1988