Experiences Developing a Virtual Shared Memory System Using High-Level Object Paradigms

J. Cordsen, J. Nolte, and W. Schröder-Preikschat

[1] GMD FIRST
Rudower Chaussee 5, D-12489 Berlin, Germany
{jc,jon}@first.gmd.de
[2] University of Magdeburg, Universitätsplatz 2
D-39106 Magdeburg, Germany
wosch@cs.uni-magdeburg.de

Abstract. Shared-memory programming is still a common and popular way of utilizing parallel machines for high-performance computing. Virtual shared memory (VSM) systems promote a gentle migration path allowing the execution of shared-memory programs on distributed-memory machines. Such kind of systems are both complex and extremely sensitive to performance issues. Therefore many VSM systems still handle distribution aspects manually by means of low-level message-passing operations to gain maximum performance. In contrast, in the PEACE operating system family almost *all* distribution aspects have been covered conveniently and yet efficiently by so-called *dual objects*. In this paper the VSM subsystem, called VOTE, of PEACE is presented as a case study for complex systems services that claim for high-level but lightweight object models with an efficient implementation.

1 Introduction

Shared-memory programming is still a common and popular way of utilizing parallel machines for high-performance computing. This programming style is based on a well-known methodology and supported by high-quality programming environments (e.g. compilers and debuggers) and matured libraries. It will be still dominant in the near future due to the lack of other accepted and pioneering approaches to parallel programming. Consequently, during the last decade large efforts were spent applying the shared-memory paradigm to distributed-memory parallel machines. This led to the development of various (hardware- and/or software-supported) *virtual shared memory* (VSM) systems. Examples are IVY[15], Midway[2], Munin[4], or DASH[13].

By means of architectural transparency, VSM systems promote a gentle migration path allowing the execution of shared-memory programs on distributed-memory machines. The particular motivation behind VOTE was to support a symbiosis of architectural transparency *and* efficiency.

E. Jul (Ed.): ECOOP'98, LNCS 1445, pp. 285–306, 1998.

As an extension to the PEACE parallel operating system[22], VOTE benefits from the advantages offered by problem-oriented kernels being tailored to particular application demands. Following the pattern of PEACE, the design and implementation of VOTE was strongly influenced by the program family concept[21]. Rather than providing a single consistency protocol which manages the replicated VSM data for all kinds of shared-memory programs, users are offered problem-oriented solutions by a *family of consistency protocols*[7].

Both, VOTE and PEACE exhibit a lightweight system structure, resulting in a high-performance software backplane for parallel computing. The lightweight structure is not only a result of having applied the family concept in the system software design process. It is also the result of the exploitation of an optimized high-level object paradigm in the implementation process. This object paradigm is found by *dual objects* supporting object-oriented (system) programming in a distributed environment[20].

This paper describes the dual-object-based implementation of the VOTE system and gives a performance analysis of both the VOTE system as well as the dual object implementation. It is organized as follows: Section 2 briefly discusses the design principles and basic architecture of PEACE. Section 3 introduces the notion of dual objects and describes language-level support as well as implementation; Section 4 presents a brief insight into the VOTE system describing its building blocks and their interactions; Section 5 makes a performance analysis; Section 6 concludes the paper.

2 PEACE

PEACE is a *framework* for (distributed) parallel applications dedicated to run on distributed memory (massively) parallel architectures. Although specifically designed to support high performance parallel computing, this framework is also suitable for constructing (microkernel-based) distributed operating systems with real-time capabilities as well as object-oriented parallel computing platforms for workstation networks.

2.1 Design Principles

Parallel operating systems must exhibit a lightweight or even *featherweight system structure* that is adaptable to the individual needs of both an application program and the hardware architecture. The approach followed by PEACE is to understand a parallel operating system as a *program family*[21] and to use *object orientation*[24] as the fundamental implementation discipline. The former concept (program families) helps prevent the design of a monolithic system organization, while object orientation enables the efficient implementation of a highly modular system structure.

The program family concept distinguishes between a *minimal subset of system functions* and *minimal system extensions*. It does not dictate any particular implementation technique. The minimal subset of system functions defines a

platform of fundamental abstractions serving to implement minimal system extensions. These extensions, then, are made on the basis of an *incremental system design*[10], with each new level being a new minimal basis (i.e., *abstract machine*) for additional higher-level system extensions. A true application-oriented system evolves, since extensions are only made on demand, namely, when needed to implement a specific system feature that supports a specific application. Design decisions are postponed as long as possible. In this process, system construction takes place bottom-up but is controlled in a top-down (application-driven) fashion.

In its last consequence, applications become the final system extensions. The traditional boundary between application and operating system disappears. The operating system extends into the application, and vice versa. Inheritance is the appropriate technique to either introduce new system extensions or replace existing system extensions by alternate implementations. Either case, the system extensions are customized with respect to specific user demands and will be present at runtime only in coexistence with the corresponding application. Thus, applications are not forced to pay for (operating system) resources that will never be used.

2.2 Functional Decomposition

The global architecture assumes that a member of the PEACE parallel operating system family is constructed from three major building blocks. These building blocks are the *nucleus*, the *kernel*, and POSE, the *Parallel Operating System Extension* (Fig. 1). In addition to the system components, the *application* is considered as the fourth integral part of this architecture. The application largely determines the complexity of a family member and the distribution of the building.

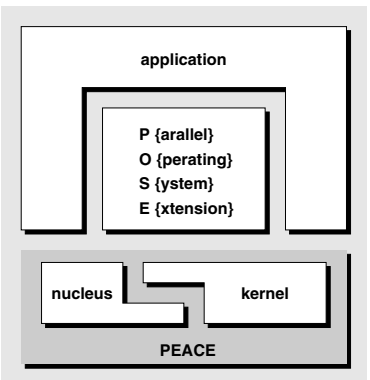

Fig. 1. Building blocks

The nucleus implements system-wide interprocess communication and provides a runtime executive for the processing of threads. It is part of the kernel domain, with the kernel being a multi-threaded system component that encapsulates minimal nucleus extensions. These extensions implement device abstractions, dynamic creation and destruction of process objects, the association of process objects with naming domains and address spaces, and the propagation of exceptional events (traps, interrupts). Application-oriented services such as naming, process and memory management, file handling, I/O, and load balancing are performed by POSE.

Kernel and POSE services are built by dual objects. As will be discussed later, the dual objects are managed by so called *clerks*. Since a clerk is implemented as an active object, the two building blocks kernel and POSE are made from active objects managing dual objects. In contrast, the nucleus is an ensemble of passive objects that schedule active objects.

An active object is implemented by a *lightweight process*. A number of these objects may share the same address space, thus constituting a *team* of lightweight processes, i.e., a *heavyweight process*. Each service that is provided by both POSE and the kernel is implemented by such a team and represents a PEACE *entity*. Entities are system extensions. They are loaded on demand and (in most cases) can be arbitrarily distributed.

The dividing line between user and supervisor mode as shown in Figure 1 is a logical boundary only. It depends on the actual representation of the interactions specified by the *functional hierarchy*[10] (and of the hardware architecture) whether this boundary is physically present.

VOTE is an autonomous POSE component. It serves as a minimal kernel extension, in particular the address space management subsystem. The PEACE family members implementing only virtual shared memory will consist of a single POSE component, i.e. VOTE, a scaled-down kernel encapsulating both the MMU driver and threads management, and the nucleus.

2.3 Functional Hierarchy

The functional hierarchy (Fig. 2) defined between POSE, the kernel, and the nucleus makes possible a very high degree of decentralization. From the design point of view, neither the kernel nor POSE need to be present on every node, only the nucleus. In a specific configuration, the majority of the nodes of a massively parallel machine are equipped with the nucleus only. Some nodes are supported by the kernel, and a few nodes are allocated to POSE. All nodes can be used for application processing, but they are not all obliged to be shared between user tasks and system tasks. The functional hierarchy of the three building blocks expresses the logical design of PEACE, but not necessarily the physical representation.

Nucleus services are made available to the application via *Nearby Object Invocation* (NOI). The logical design assumes a separation of the nucleus from the application (and POSE), which calls for the use of traps to invoke the nucleus and for address space isolation. This is the place where *cross domain calls* may

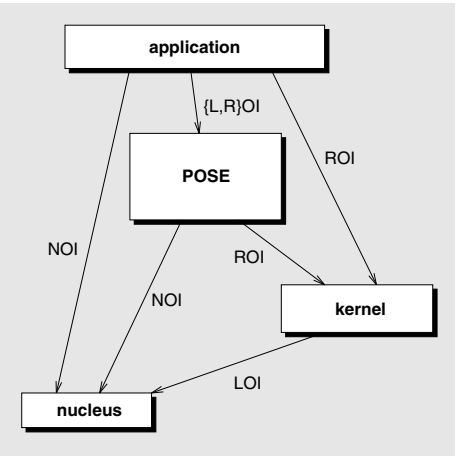

Fig. 2. Invocation scheme

happen. The nucleus is "nearby" the using entity. It shares with the entity the same node, but not necessarily the same address space segment. In addition to that, there are nucleus configurations which totally sacrifice the trap interface, defining a single address space that lodges possibly all four building blocks.

The kernel resides with the nucleus in the same address space. Together they constitute the *kernel entity*. The kernel therefore performs *Local Object Invocation* (LOI) to request nucleus services. Kernel services are made available via *Remote Object Invocation* (ROI)[14]. The ROI scheme always implies context switching, but not necessarily address space switching. A separate thread of control is used to execute the requested method (i.e., service). In contrast to that, NOI (logically) implies the activation/deactivation of the nucleus address space via local system call traps. The implementation of ROI takes advantage of the network-wide message passing services provided by the nucleus and, thus, is based on NOI.

Services of Pose are requested via LOI and ROI. The former scheme is used to interact with the Pose runtime system library whereas the latter is used to interact with the Pose active objects. In certain situations the Pose library directly transforms the issued LOI into one or more ROI requests to the kernel. The Pose service then is not provided by an active object, but entirely on a library basis (i.e., passive object).

Although the functional hierarchy assumes NOI for the interaction between application (Pose) and nucleus, the LOI scheme is used for those members of the Peace family that place their focus on performance. The entrance to the nucleus is represented as an abstract data type with two implementations. The first implementation sacrifices vertical and horizontal isolation. Thus, there is neither a separation between the user and supervisor modes of operation (verti-

cal isolation) nor a separation between competing tasks (horizontal isolation). In this case, NOI actually means LOI. Horizontal isolation means that user/system entities have a private address space, that is, operate in a private protection domain. The second implementation assumes complete (i.e., vertical and horizontal) isolation and requires a trap-based activation of the nucleus. NOI then becomes a cross-domain call. The variants basically distinguish between single-tasking (no isolation) and multitasking (isolation) modes of operation. They implement different members of the *kernel family*.

3 Dual Objects

Family based systems can be implemented conveniently by means of object oriented programming paradigms. Operating system services are implemented as classes then and users can extend and specialize these system classes by means of inheritance mechanisms. Thus both PEACE and VOTE are entirely implemented in C++. In VOTE inheritance is used to introduce specialized members of the family of consistency protocols and customize the building blocks to various user demands.

In theory this scenario is sound and straight forward but in practice the conceptual advantages of object orientation are *extremely* hard to exploit without suitable object models and language-level support for object-oriented implementation techniques in distributed contexts. In order to implement VSM semantics, instances of the VOTE classes are to be allocated to different nodes of a distributed-memory parallel computer. As a consequence, these instances need to be accessed network-wide.

When users extend and specialize e.g. PEACE or VOTE classes by means of inheritance mechanisms, class hierarchies need to be extended across address spaces as well as network boundaries and objects can be fragmented across address spaces. This in turn can lead to serious performance penalties caused by frequent remote invocations, when application classes closely interact with their system-level base classes. On the other hand it is obvious that client classes cannot have full access to system-level state information to avoid forgery and ease resource sharing amongst many clients. Implementing system services as fragmented objects[17] would have supported independence as well as encapsulation of object fragments allocated in different address spaces. Nevertheless we considered that model already too complex for those very lightweight system structures we were aiming at, because the fragmented object model partially relied on group communication mechanisms and did not support inheritance-based fragmentation.

Having such problems in mind we designed a general object model for system programming. This model transposes the classical coarse grained user/supervisor memory model of monolithic systems to very small objects of language-level granularity. So-called *dual objects*[20] implement system services and encapsulate both user-level and system-level state information in a single object context. Clients are allowed to control the user-level part of a dual object directly and

efficiently within their address spaces, whereas system servers have transparent access to both parts during service invocations. Thus much closer interactions between user and system classes are possible as in conventional object models and the model remains simple enough to be implemented efficiently.

3.1 Language-level Support

Dual objects are described by annotated C++ classes[1] that specify user-level and system-level class members. To retain a strong backward compatibility to original C++, all `public` and `protected` members are considered to be user-level, whereas all `private` members belong to the system-level. Thus the weak language-level protection of private class members in C++ is enforced by strong encapsulation of private data in the (remote) address-space of the server.

Fig. 3 shows a much simplified example of the dual `Adviser` class that controls consistency protocols in the VOTE system. Dual classes are identified using a `/*!dual!*/` annotation following the closing bracket of the class declaration. The private member slots belong to the system-level whereas all `protected` members are user-level data members. Thus the methods `indicateWS()`, `setMode()`, `setProt()` and `setCap()` can be executed directly within the client's address space as indicated by the `/*!local!*/` annotation whereas methods like `handle()` are executed remotely under the control of the server using remote object invocation techniques (ROI)[14]. During invocations the user-level members are made available for the server. Thus the `handle()` method has transparent access to the actual `cap`, `mode`, `protocol`, `high` and `low` members.

Methods and parameters are annotated in an IDL-like style to specify parameter functionality and remote invocation semantics. Parameters to remote methods are passed by value by default. Pointers have to be annotated to specify e.g. input (`in`) or output (`out`) parameters[2].

Generic bulk data is not automatically handled through language-level parameter passing but by explicit remote `fetch()` and `store()` operations provided by the PEACE nucleus. These two primitives provide fast end-to-end data transfers between (remote) address spaces. An address-space capability (`Ticket`) is required to apply these methods. Thus in the `Adviser` example, `handle()` uses the user-level `cap` member to perform bulk data transfers between the client's and the server's address spaces.

Dual classes can be composed by (even remote) inheritance mechanisms and thus important C++ features like multiple inheritance are retained. These classes are fed to a *dual object generator* (DOG) to transform them into functionally

[1] Basing on annotated C++ was more a pragmatic than a conceptual decision. At the time starting the PEACE development, it was fairly hard to convince users to better employ C++ rather than Fortran for parallel programming. It would have been even harder, if not impossible, to come up with some sort of interface definition language different from C++ and Fortran. Thus, annotating C++ was seen to be the best possible compromise at that point in time.

[2] For a complete description of annotations see[19,20].

```
class Adviser : ... {
    private:                // system-level
        Actor** actors;
        Actor*  my_actor;
        int num_actors;
    protected:              // user-level
        Ticket   cap;
        AccMode  mode;
        VSMProt  protocol;
        Addr     low;
        Addr     high;
    public:
        // local methods
        void indicateWS (int high, int low) /*!local!*/
        {
             this->high = high;
             this->low = low;
        }

        void setMode (AccMode mode) /*!local!*/
        {
             this->mode = mode;
        }

        void setProt (VSMProt prot) /*!local!*/
        {
             this->protocol = prot;
        }

        void setCap (Ticket cap) /*!local!*/
        {
             this->cap = cap;
        }

        // remote methods
        int  handle (int page, Addr addr, AccFault type);
        ...
}/*!dual!*/;
```

Fig. 3. The Dual Adviser class

enriched C++ classes capable of dealing with the distribution aspects of dual objects. Client and server representations of a dual object are generated. The same holds for the stub methods used for remote invocation. An additional preprocessor version of the DOG integrates the dual object model into standard C++ to ease the implementation of distributed and parallel system services (as well as C++ applications).

3.2 Resource Sharing

A client (or a server) can grant access to its resources by creating clones of its dual object instances and transferring these clones to other processes. Whenever a dual object is cloned, only the user-level part is copied, whereas the system-level part is (transitively) shared amongst all clones. Thus in the `Adviser` example (fig. 3) the private system-level members are shared amongst all instances transitively cloned from the same origin. In contrast all `protected` user-level members are independent copies that will be manipulated or initialized independently from each other (fig. 4). The user-level part of a dual object then

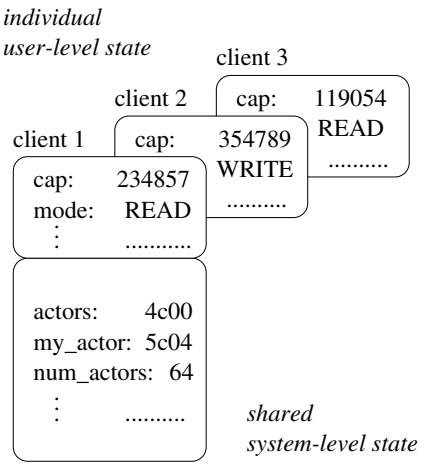

Fig. 4. Resource Sharing with Dual Objects

becomes a client-specific context that is implicitly provided to servers during service invocations. Thus the `cap`, `mode`, `protocol`, `high` and `low` members (fig. 4) will always refer to the user-level part of the client actually calling a specific `Adviser` instance.

As a result servers are not enforced to maintain client-specific data themselves. Robustness of services is therefore enhanced and system-level data can be shared conveniently and economically amongst many clients. In fact, these resource sharing facilities are comparable to delegation based models[16,25] with the major difference that the sharing facilities of dual objects are statically defined through classes whereas common delegation schemes allow dynamic sharing.

3.3 Runtime Model and Implementation Issues

Dual objects have two representations at runtime, one for clients and one for servers. We call instances of the client's representations *likenesses* and instances of the server's representation *prototypes*. A likeness reflects the public interface of a dual class and consists of public and protected members only, whereas the prototype consists of all members. Furthermore, the likeness holds a (remote) reference to its prototype. Thus a likeness is both an object that can be manipulated locally as well as a proxy[23] for a remote prototype.

Prototypes are passive C++ objects that are kept in so-called *domains*[18]. Domains are a concept for local object spaces that are managed by active server objects we call *clerks*. These clerks are able to instantiate new prototypes upon request and control access to all objects within their domains. Many domains may share an address space or may have separate address spaces either on the same machine or somewhere in a network to constitute a global distributed object

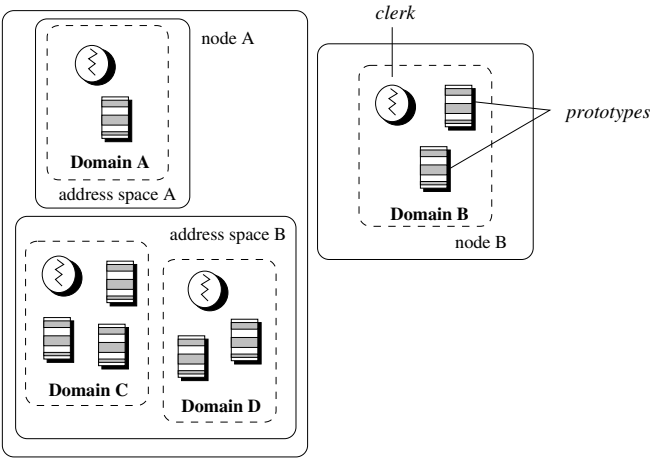

Fig. 5. Domains

space (fig. 5). Furthermore, a domain may either be sequential or concurrent. Sequential domains are monitors and allow exactly one object in the domain to be manipulated at a time. Concurrent domains manage a dynamic thread pool and implement read/write monitors on single object instances.

When a dual object is created, an instantiation request is sent to the clerk of the domain selected to host the new prototype. Domains are selected either by name contacting a name service or by a *unique identifier* denoting the communication address of the clerk controlling the domain. The clerk in turn creates the prototype and executes its constructor. After initialization all user-level parts of the prototype are extracted and sent back to the requesting client. Here a likeness is initialized with the user-level data and the remote reference to the newly created prototype. In fact, any time a client declares a new likeness instance which is not a clone of an existing likeness, the instantiation procedure described above is transparently executed.

Methods that access user-level data only are executed locally on the likeness leaving the prototype untouched. All methods that are executed on the likeness that involve system-level data will in fact be remote object invocations on a remote prototype. Since the user-level part may have been changed by previous local calls, it is transmitted along with the arguments of the call (fig.6). The receiving clerk then will update the prototype by means of the actual user-level data before the method is executed[3]. When the method returns the user-level part is extracted and sent back to the client along with the results of the method. The likeness in turn is updated with the actual data. This protocol causes some additional overhead for those methods which access both user-level and system-

[3] This is necessary because we use standard C++ compilers as back-end. Otherwise user-level members could be referenced differently from system-level members.

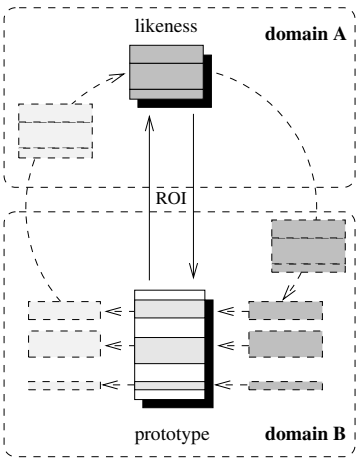

Fig. 6. Remote Invocation

level data simultaneously. Since that overhead is comparable to implicit parameter passing it can be neglected if the transfer costs of user-level state information is small compared to the execution time of the method (fig. 3). Other methods are either not affected or even executed locally at the client site if no system-level data is accessed at all.

4 The VOTE System

A shared-memory program running on top of VOTE is executed in a multiple reader/single writer (MRSW) sequential consistency model[12] to ensure architectural transparency. At any time, the program may change the consistency maintenance, keeping on execution by virtue of a different memory consistency model. VOTE supports a number of performance enhancement techniques. These techniques help avoiding sequences of read/write memory access faults, pre-paging and releasing address ranges, and to provide support for one-sided communication in order to propagate data to each process in a specified set of destinations. A fine-grained multiple writer model allows (within a page) modifying accesses with a subsequent restoration scheme[7] in order to unify a sequential consistent result. Furthermore, VOTE also supports message-passing communication functions which operate in coexistence to the demand paging of sequential consistency.

A detailed description of the VOTE system, its performance enhancement techniques, and studies about efficiency and scalability of parallel VSM applications can be found in[8]. This paper concentrates on the VOTE core only to show how dual objects have been exploited to construct an efficient VSM system.

4.1 Functional Units

VOTE is distinguishing three functional units responsible for handling consistency maintenance and raising memory access faults. These three functional units are called *catcher*, *actor*, and *adviser* (Fig. 7). For the sake of clarity, the terms *requesting site*, *knowing site*, and *owning site* will be used in the following. The *requesting site* is the process causing the memory access fault. The *knowing site* is the process implementing the consistency maintenance, whereas the *owning site* is the process actually owning the requested memory page. In specific situations, the *knowing site* may also play the role of the *owning site*.

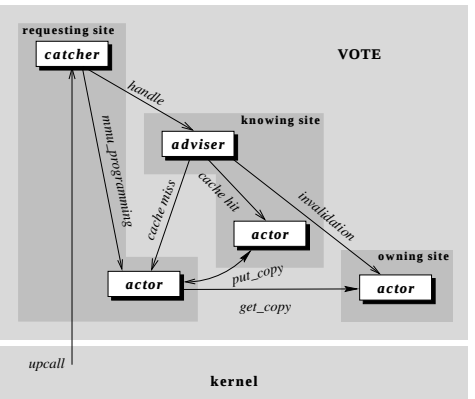

Fig. 7. VOTE building blocks

An application process using the global address space of VOTE will be associated with an exception handler (*catcher*). In case of a memory access fault, the *catcher* is invoked (via an upcall) by the PEACE kernel. The *catcher* determines the consistency maintenance instance relevant for handling the access fault. For this purpose, a pre-determined (user-directed) mapping from memory pages to consistency maintenance objects is used. Upon this information, the *catcher* calls the *adviser* at the *knowing site*.

The *adviser* object implements the consistency maintenance for the requested memory page. This instance maintains directory information about the distribution of the memory pages it has taken responsibility for. Each *adviser* is supplemented by a set of so called *actor* objects. When a VSM variable is declared to be maintained by a specific *adviser*, an associated *actor* will be created in the address space of the declaring application process. Thus, using several *adviser* objects in order to maintain different memory pages will result in the same number of *actor* instances associated to each of the application processes. The *adviser* itself creates an *actor* for the purpose of caching VSM pages and to optimize access fault handling.

The purpose of an *actor* is to provide to its *adviser* an interface controlling the memory management and the movement of VSM memory pages. The functional encapsulation of, on the one hand, consistency maintenance by the *adviser* and, on the other hand, memory management and data transfer by the *actor* enables VOTE to make use of idle processors or dedicated system processors. State-of-the-art consistency maintenance operates with a dynamic distributed ownership-based protocol. Due to the ownership approach, this scheme is restricted to operate only on processors running application tasks. Because of this disadvantage, VOTE rejects the dynamic distributed consistency maintenance and uses the highly optimized fixed distributed scheme instead.

4.2 Interactions of the Functional Units

If an application causes a memory access fault, kernel-based trap handling saves the current context and activates the *catcher* being in charge of raising the memory access fault. The *catcher* allocates a memory region, determines the responsible *knowledge site* and calls `handle()` (Fig.3) with the requested page number, address of the allocated memory region, and information about the access fault type (i.e. protection or access violation) as arguments. When an *adviser* receives the request to handle a memory access fault, the access fault type, the directory information, and client specific information determines the further processing.

The most simple case of consistency maintenance is a protection violation, that is the *requesting site* already owns a copy of the memory page with a read-only access permission. Then the *adviser* sends invalidation messages to the set of *owning sites*, updates the directory information and replies to the *requesting site* in order to upgrade its memory access permission.

If the access fault was an access violation, the *knowing site* checks if the requested memory page has been cached. In case of a cache hit, the page is transfered directly to the *knowing site* using `put_copy()`.[4] If the access violation was due to a write access, the *knowing site* invalidates the set of copy holders before the directory information is updated. In contrast, a read access violation requires only adding the *requesting site* to the set of copy holders. Afterwards, the *requesting site* is sent a reply for local MMU programming.

The final two cases handle access violations if a cache miss for the requested page was detected at the *knowing site*. In this situation the *adviser* calls the *actor* of the *requesting site* with the information where to get a copy of the memory page. Then, data is copied from the *owning site* to the *requesting site* using `get_copy()`. At the *requesting site*, the desired access permission is defined. If the page has read-only access permission, a copy of the page is sent to the *knowing site* as well (using `put_copy()`). As described above, caching these pages at the *knowing site* results in a prompt handling of future read access faults with respect to the same pages. Finally, control returns to the *adviser*. If a write access

[4] The description is simplified for reasons of clarity. To ensure sequential consistency, VOTE uses a distributed locking protocol.

violation is to be handled, invalidations are sent to the set of *owning sites*. The directory information is updated and the *requesting site* is sent a reply.

5 Exploiting Dual Objects

The entire VOTE software architecture is implemented using the high-level object paradigm of dual objects. The three function units *catcher*, *actor* and *adviser* are implemented as dual classes. Hence, the interactions taking place between the instances of these classes (i.e. the dual objects) are to be carried out on a ROI-basis mostly.

```
class Catcher : public Guardian {
    private:
        Adviser* adviser;
        ...
    public:
        Catcher (/*!in!*/Adviser* adviser);
        void exception(LogAdr loc, LogAdr pc, MemFault cause)
        {
            ...
            adviser->handle(loc.Index() ...); // remote call
            ...
        }
}/*!dual!*/;
```

Fig. 8. The Catcher class

Fig. 8 shows the definition of the dual `Catcher` class that reacts on page-faults and propagates these to an appropriate `Adviser` instance for VSM protocol handling. The `Catcher` class is derived from a dual `Guardian` class. This class gives access to low-level memory management of the PEACE kernel and propagates traps (even across the network) to higher-level abstractions. Method `exception()` redefines a virtual `Guardian` method and implements application-specific fault handling. For performance reasons we place `Catcher` instances in a local domain sharing the local address space of the process, whose traps need to be caught. Nevertheless, in principal these faults could also be handled at any place in the network transparently to the nucleus detecting the fault because dual objects provide location transparency.

Upon an access fault, the `exception()` method of a `Catcher` instance will be called by the nucleus and executed in the domain where the `Catcher` actually resides. The `exception()` method then performs local VSM processing such as memory allocation before it contacts the `Adviser` instance responsible for the memory region in question. This call is carried out on a ROI-basis invoking the `handle()` method of the dual `Adviser` object (fig. 3). The *catcher* is unaware of the actual location of its *adviser* and the `Adviser` instance can be located anywhere in the network.

The same holds for instances of the dual class `Actor` that implements methods like `put_copy()` and `get_copy()` for remote page transfers as well as methods for remote MMU handling (fig.9).

```
class Actor : ... {
    private:
        VSMInfo*   info;
    public:
        ...
        // remote methods for page transfer
        int  put_copy (int page, /*!in!*/VSMPage* data);
        int  get_copy (int page, /*!out!*/VSMPage* data);
        ...
        // remote methods for address space handling transfer
        int  invalidate (int page);
        ...
}/*!dual!*/;
```

Fig. 9. The Actor class

The complete dual `Actor` class sketched here very briefly implements in fact more than 35 different methods for remote page and MMU handling, that optimize several access cases and allow remote control of address space management. Although these remote methods could have been implemented manually by means of low-level message passing operations, these numbers indicate that language-level ROI support can improve software productivity significantly. This is especially true during prototyping phases when interfaces quite often change and experimentation consumes a lot of the total development time. We estimate that a manual implementation would have consumed at least several hours up to a few days of additional error prone work that was performed reliably by an automatic tool in a fraction of a second.

The dual `Actor` class basically uses the ROI capabilities provided by dual classes. The dual `Adviser` class (fig. 3) intensively applies the extended resource sharing model provided by dual objects. The private system-level part contains references to a set of `Actor` instances (data members `actors` and `num_actors`). Data member `my_actor` points to the local *actor*. This variable is used to optimize the handling of read access faults in case of a cache hit for the requested VSM page. The user-level data members (`protocol`, `low`, `high`, and `mode`) are used to customize the consistency maintenance with respect to the specific memory access characteristics of an application. Each process belonging to the parallel application gets a clone of the dual `Adviser` instance controlling a VSM segment shared by these processes. Recall that in the case of dual objects only user-level parts are cloned, whereas the system-level part is implicitly shared amongst all clones (refer to section 3.2). Therefore it is possible to individually specify the current working set of an application process defining the members `low` and `high` to minimize page-faults and false sharing, when several processes operate on the same page but on disjoint locations within the page. Furthermore, the `mode` and

`protocol` members provide information about the required memory consistency models and expected page-access patterns. Thus each process belonging to the parallel application can customize its individual access patterns and consistency requirements through efficient local operations, that only manipulate user-level state. Upon a memory access fault, the *catcher* invokes the *adviser* thereby automatically providing the customized user-level state information of the faulting process.

Without the dual object model we either would have to perform costly remote invocations to optimize page access or store this information locally in a different kind of object and pass this information explicitly as additional parameters upon any access fault. The first case would be extremely inefficient and virtually discards its intended optimization effect completely. The second case would require clumsy workarounds and unclear interfaces since the user-level data would to be passed as extra arguments to the methods of the `Adviser` class. The dual object model keeps all related data in a uniform context and effectively hides distribution aspects.

6 Performance Analysis

The VOTE performance was measured on a multi-node MANNA system. Every dual-processor MANNA node consists of two 50-MIPS RISC processors (i860XP) sharing the memory and the I/O units attaching peripherals to the node; they also share the bidirectional communication link. This link connects the node with a byte-wide 16×16 crossbar switch, providing a throughput of 2×47.68 MB/s. The effective memory access rate is 381.47 MB/s.

6.1 Dual Objects

The high-level dual object model eases the design and implementation of the VOTE system. Reusability, extensibility, and efficient use of system resources are enhanced to a high extent. Table 1 shows the basic timings for elementary communication costs as well as remote object invocation times. The PEACE nucleus provides synchronous and asynchronous packet-based mechanisms (64 bytes) for inter process communication (IPC) as well as primitives for end-to-end bulk data transfer (`fetch()`, `store()`). The ROI mechanisms are built upon these mechanisms. When the argument size (including the user-level part of a dual object) exceeds an IPC-packet, bulk data transfer primitives are automatically applied to transmit bigger messages.

The typical ROI overhead is in the vicinity of 8μsec and drops to less than 5μsec in case of local communication that implies a better cache utilization[5]. Notably we cannot measure significant differences with varying parameter sizes up to the size of a packet. The DOG generates statically typed message formats

[5] The generated code as well as the code in the ROI runtime system is the same in both the local and the remote case.

Operation	Time (μsec)
Inter Process Communication (IPC)	157
Async. IPC	18
bulk data transfer (fetch)	122
bulk data transfer (store)	28
ROI with argument size not exceeding an IPC packet	165-168
async. ROI	ca. 26
ROI with implicit bulk data transfer	308
remote object creation	266

Table 1. Performance of basic operations

for most methods and thus marshaling costs for simple data types and aggregate types are *extremely* low. Only array data types and aggregate data types that contain arrays require significant improvement (see also table 2).

Table 2 shows the overheads of memory allocation and copying when executing `get_copy()` (refer to Fig. 7). The benchmarks were run using dual objects of different utilization, with variable sized data sets, and exploiting different compilers (Metaware, Portland).[6]

Size (KB)	Metaware compiler					Portland compiler				
	Copy granularity				Basic	Copy granularity				Basic
	char	int	double	memcpy	costs	char	int	double	memcpy	costs
4	1793	807	607	463	369	1545	691	552	478	376
8	3325	1350	946	658	483	2834	1107	816	667	489
16	6477	2503	1743	1071	701	5539	2084	1504	1100	701
32	13572	4995	3610	1936	1151	10999	4061	2922	1978	1166
64	26969	9807	7115	3691	2058	21848	7986	5702	3724	2065

Table 2. Overhead of memory allocation and copy efforts (μs)

The execution of `get_copy()` requires calling the *owning site* and transferring a page containing the requested data. Since `get_copy()` is a method applied to a dual `Actor` object, the call is carried out on a ROI-basis. In this particular case, (1) a remote rendezvous between client and server takes place and (2) a data

[6] The Metaware compiler is a product of MetaWare Inc, version Rel 2.1e. The Portland compiler is a product of The Portland Group, Inc., version Rel 3.1-1. The Metaware compiler is strong achieving a better MIPS rate, whereas the Portland compiler results in better FLOPS rates. Consequently, an operating system is best supported by the Metware compiler while "number crunching" applications are best supported by the Portland compiler.

transfer is initiated to get the requested page in place. These two steps make up the basic communication costs of remote memory-to-memory paging in VOTE. Depending on the compiler in use, Table 2 lists the sum of both times.

As indicated in Table 2, VOTE cannot make full use of the parameter passing mechanisms of the dual object model to come close to the basic communication costs. Rather, VOTE allocates and transfers the data itself. The effects of implicit parameter passing are shown in the columns `char`, `int`, and `double`. In these cases, following the typical use pattern of a C++ VSM programmer, the requested memory page is a typed array of the plain data types `char`, `int`, and `double`, respectively. Many of these pages may constitute some sort of matrix used by the (numerical) application program.

Invoking `get_copy()`, the ROI stubs linearize the typed page into a typed message before communication becomes effective. Similar holds for the receiving site, where a typed page is rebuilt from the received typed message. For each page, this causes a local memory-to-memory copy overhead in addition to the basic (page) data transfer. As shown, the copy overhead varies depending on the actual plain data types of the page. This is because copying is performed in a (high-level language) loop created by the DOG and containing assignments to either of `char`, `int`, or `double` instances.

Table 2 shows that the copying overhead dominates the total execution time of the data transfer and is not acceptable for performance critical services. Even the use of a highly optimized memory-to-memory copying routine (`memcpy()`) considerably defects the overall performance. Compared to the basic communication costs, in the best case the VOTE consistency maintenance would increase by 25.5 % (using the Metaware compiler and a `memcpy()` of 4 KB pages) and in the worst case the runtime would grow as much as 1210.5 % (using the Metaware compiler and performing a `char` copy loop to transfer 64 KB pages). Here are the limits of straightforward language-level support for parameter passing. Although the DOG is perfectly able to pass objects with multi dimensional arrays as parameters, this feature could not be used for performance reasons. Thus pages needed to be treated like bulk data and needed to be handled manually by means of `fetch()` and `store()` methods that do not require internal copies. On the other hand, most remote synchronization and consistency protocol handling could be performed flawlessly and transparently using dual objects and the language-level support provided by the DOG. Therefore the most complex parts of the VOTE system could be designed and implemented with a minimum of development overhead. After this work was completed, the manual optimization of the page transfers was straightforward and easy to accomplish by means of the PEACE data transfer primitives.

In VOTE, access violation has been especially optimized with respect to a read access fault on a page cached by the *adviser*. The optimization took advantage from specific data-transfer features provided by the PEACE nucleus, in particular the asynchronous write of an arbitrarily sized data segment into remote memory[22].

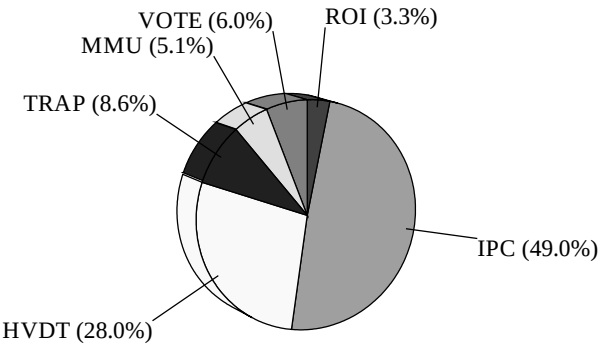

Fig. 10. Percentages of system functions during consistency maintenance

The breakdown into the basic activities of trap handling, packet communication (IPC), dual object overhead (ROI), data transfer (HVDT), MMU programming, and maintenance activities in VOTE shows that communication is the dominating issue (Fig. 10). About 80% of the time spent by VOTE for access fault handling is devoted to communication, that is ROI, IPC, and HVDT. Therefore, a high-performance communication facility is of great importance for VOTE. Dual objects make the handling of the VOTE classes much easier and, used in an appropriate way, add an negligible overhead to the basic message-passing mechanisms.

6.2 Related Works

Considering the implementation of VOTE and the resulting performance, the dual object paradigm proved to be the right decision for the design and development of the VSM subsystem of PEACE. Making a cost/profit analysis, the additional overhead of dual objects is very low and can be neglected when running real-world applications. As outlined by Table 3, the end-user performance of VOTE compared to other VSM systems applying very low-level message passing layers for implementation appears to be very good.

System	Platform	CPU	Network	Time (ms)
Mether	SunOS4.0	25 MHz MC68020	1.2 MB/s	70-100
Munin	V	25 MHz MC68020	1.2 MB/s	13-31
Myoan	OSF/1 + NX	50 MHz i860XP	200 MB/s	4.068
VOTE	PEACE + dual objects	50 MHz i860XP	47.68 MB/s	0.667

Table 3. Comparison of read-fault handling in different VSM systems

Mether and Munin both run on rather old hardware. Nevertheless, a comparison of Myoan[3] and VOTE is fair because of the same CPU foundation in both systems. Myoan runs on the Intel Paragon machine with a network throughput which is more than four times better than the throughput of the MANNA communication network. Yet the performance of VOTE is more than six times better than handling a read access fault in Myoan, although communication and data transfer are responsible for about 80 % of the total costs.

One of the main advantages of VOTE compared to Myoan is the specialized operating-system kernel, which appeared in the just discussed case to be "only" a communication and thread library rather than a microkernel with additional user-level and problem-oriented communication support. In order to partly overcome the performance problems, Myoan is even using the low-level NX communication library for inter-node communications bypassing the underlying microkernel. This reduces the communication time (for 8 bytes) from 1909 μs, when exploiting the IPC functions of the OSF microkernel, to about 329 μs.

In VOTE we hadn't to go down to such low levels mainly because dual objects both provide a suitable high-level system programming paradigm and a very efficient implementation at the same time. Thus implementation issues are significantly improved and the lightweight system structure of PEACE is not compromised by complex middleware layers.

Other high-level approaches from the distributed systems area such as CORBA[9] tend to "eat up" the performance of lower layers for the sake of convenient heterogeneous computing and interoperability issues. In the high performance systems area more recent parallel C++ versions such as CC++[5], ICC++[6], Mentat[1] and MPC++[11] seem to be promising. Nevertheless, system programming still needs significantly more control over runtime issues than languages designed for application level programming usually provide. MPC++ and EUROPA C++[26] provide powerful meta-level programming facilities that could have beneficial impact on system programming in the near future.

7 Conclusion

Family based operating system services such as the VOTE VSM system are hard to implement without suitable high-level paradigms and language-level support. Design and implementation of VOTE, as well as PEACE, was strongly influenced by the concept of dual objects. This object model transposes the classical coarse grained user/supervisor memory model of monolithic systems to very small objects of language-level granularity. Therefore dual objects encapsulate both user-level and system-level state information in a single object context to encourage much closer interactions between user and system classes as in conventional object models. Servers are not enforced to maintain client-specific data themselves. Robustness of services is therefore enhanced and system-level data can be shared conveniently and economically amongst many clients.

By providing a highly abstract and yet efficient object-oriented programming environment, the (system) programmer is relieved from dealing with the peculiarities of distributed-memory parallel machines. The VOTE performance figures show that high-level object paradigms do not impose a general bottleneck for both complex and performance critical operating system services.

References

1. A.Grimshaw. Easy-to-use parallel processing with Mentat. *IEEE Computer*, 26(5), May 1993. 304
2. B. N. Bershad and J. M. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, 1991. 285
3. G. Cabillic, T. Priol, and I. Puaut. Myoan: An Implementation of the Koan Shared Virtual Memory on the Intel Paragon. Technical Report 812, Irisa, Rennes, 1994. 304
4. J. B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. PhD thesis, Rice University, 1993. 285
5. K. M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993. 304
6. A. Chien, U.S. Reddy, J.Plevyak, and J. Dolby. ICC++ – A C++ Dialect for High Performance Parallel Computing. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software, ISOTAS'96*, Kanazawa, Japan, March 1996. Springer. 304
7. J. Cordsen. Basing Virtually Shared Memory on a Family of Consistency Models. In *Proceedings of the IPPS Workshop on Support for Large-Scale Shared Memory Architectures*, pages 58–72, Cancun, Mexico, April 26th, 1994. 286, 295
8. J. Cordsen, Th. Garnatz, A. Gerischer, M. D. Gubitoso, U. Haack, M. Sander, and Schröder-Preikschat. VOTE for PEACE — Implementation and Performance of a Parallel Operating System. *IEEE Concurrency*, 5(2):16–27, 1997. 295
9. Object Management Group Document. The Common Object Request Broker: Architecture and Specification 2.0. Technical report, OMG. 304
10. A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976. 287, 288
11. Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jörg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Reflection '96*, 1996. 304
12. L. Lamport. How to make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C–28(9):241–248, September 1979. 295
13. D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 19–21, 1992. 285
14. Henry M. Levy and Ewan D. Tempero. Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software—Practice and Experience*, 21(1):77–90, January 1991. 289, 291

15. K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, 1986. 285
16. H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object–Oriented Systems. In *Special Issue of SIGPLAN notices*, volume 21, pages 214–223. ACM, November 1986. 293
17. Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Structuring Distributed Applications as Fragmented Objects. Rapport de recherche 1404, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), January 1991. 290
18. O. M. Nierstrasz. Active Objects in Hybrid. In *Special Issue of SIGPLAN notices*, volume 22, pages 243–253. ACM, December 1987. 293
19. J. Nolte. Language Level Support for Remote Object Invocation. Arbeitspapiere der GMD 654, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany, June 1992. 291
20. J. Nolte and W. Schröder-Preikschat. An Object-Oriented Computing Surface for Distributed Memory Architectures. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume 2, pages 134–143, Maui, Hawaii, January 5–8, 1993. IEEE Computer Society Press. 286, 290, 291
21. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2), 1979. 286
22. W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3. 286, 302
23. M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, MA, 1986. 293
24. P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986. 286
25. Peter Wegner. Dimensions of Object–Based Language Design. *Special issue of SIGPLAN Notices*, 22(12):88–97, October 1987. 293
26. The Europa WG. EUROPA Parallel C++ Specification. Technical report, http://www.dcs.kcl.ac.uk/EUROPA, 1997. 304