INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

Design Principles of Parallel Operating Systems —A PEACE Case Study— *

Wolfgang Schröder-Preikschat †

TR-93-020

April 1993

Abstract

Forthcoming massively parallel systems are distributed memory architectures. They consist of several hundreds to thousands of autonomous processing nodes interconnected by a high-speed network. A major challenge in operating system design for massively parallel architectures is to design a structure that reduces system bootstrap time, avoids bottlenecks in serving system calls, promotes fault tolerance, is dynamically alterable, and application-oriented. In addition to that, system-wide message passing is demanded to be of very low latency and very high efficiency. State of the art parallel operating systems design must obey the maxim not to punish an application by unneeded system functions. This requires to design a parallel operating system as a *family of program modules*, with parallel applications being an integral part of that family, and motivates *object orientation* to achieve an efficient implementation.

*This report is a condensed version of the author's habilitation and will also appear in the proceedings of the "International Summer Institut on Parallel Architectures, Languages, and Algorithms", Prague, Czech Republic, July 5–10, 1993 (Lecture Notes in Computer Science, Springer-Verlag).

†On leave from the German National Research Center for Computer Science, GMD FIRST, Rudower Chaussee 5, D-1199 Berlin, Germany, wosch@first.gmd.de

1 Introduction

Compared to the classical operating systems area the design and development of parallel operating systems is a quite new discipline. At the end of the eighties, this discipline became important with the breakthrough of parallel computer systems based on distributed memory architectures. In the nineties, it will be even more important in the realm of massively parallel systems. Massively parallel systems consist of hundreds or thousands of processing nodes. Recent studies [8] already present the vision of millions of cooperating nodes. The preferred paradigm to construct parallel machines of that scale relies on distributed memory. At the hardware level, the view of a common global memory is sacrificed. This view will be re-introduced, if at all, by software virtualizing the physically distributed memory space. A very high-speed message passing network then serves as the "backbone" to interconnect the nodes. Every node operates more or less autonomously. A *Multiple-Instruction/Multiple-Data* (MIMD) operation principle is established. In the following, the focus is on the logical design of parallel operating systems for these MIMD computer architectures.

Traditionally, parallel applications call for extremely high performance of both hardware and software. Very low *communication latency* and, thus, very high *communication performance* are still the predominant user requirements that must be fulfilled by parallel operating systems for MIMD computer architectures. While in the past shared memory architectures did play the major role, it is very well accepted now that innovative parallel computers will be *tightly-coupled distributed systems*. In order to improve programmability and, thus, not only user acceptance but also maintainability and applicability, *transparency* [14] cannot be sacrificed. Transparency, however, can be achieved only at the cost of *efficiency*. Transparency in the context discussed here means to hide from parallel applications problems coming up with distributed systems. Efficiency means to reduce the message startup time for a given application to an absolute minimum. The former implies system software overhead, whereas the latter demands to generally avoid this overhead. A "vicious circle" parallel operating systems are assigned to break up.

As a solution to this problem, two major aspects will be discussed in the following investigations. Firstly, the design of a parallel operating system should lead to a *program family* [25] and, secondly, the implementation should follow the paradigm of *object orientation* [32]. The goal is to try to answer the question of whether distributed (microkernel-based) operating systems are suited for distributed memory parallel computers or specific system software structures are required. If there is the need for specific system software structures, the questioning of how much these structures differ from, or will have in common with, distributed (microkernel-based) operating systems is investigated.

The outline of the paper is as follows. Section 2 briefly discusses the characteristics of parallel operating systems. In Section 3 the functional decomposition made in design process of the PEACE [29] parallel operating system is considered. PEACE will serve as a case study system. Following that, Section 4 explains various system configurations emerging from problem-oriented arrangements of the PEACE building blocks. Conclusions are presented in Section 5.

2 Parallel Operating Systems

Almost any new (commercial) operating system that appears at the market is based on the microkernel architecture [10]. The most recent example of this fact is WINDOWS-NT^{TM} [6]. At first sight, this seems to suggest exploiting a microkernel as the common platform on top of which the parallel operating system should be built. It is quite feasible to even port microkernel-based UNIX^{TM} systems onto distributed memory parallel computers [34], with every node executing the microkernel, a subset of system servers, and at least one application task. But the question comes up whether this really is the right approach. Adopting portable operating systems to parallel machines and providing specific, portable operating system support for parallel computing on these machines are two different matters.

The purpose of this section is to show that a microkernel-based architecture is not the ultima ratio for building parallel operating systems. First, a brief survey of the current state of the art of parallel operating systems takes place. After that, the two most crucial design constraints are discussed. These constraints give the arguments to seek for a platform different than a microkernel. Finally, the design principles standing behind the alternative approach are presented.

2.1 State of the Art

For many years, parallel programming was closely related to parallel computing in a *shared memory multiprocessor system* environment. This understanding was also a dominating issue in operating systems development over the past decade. Multiprocessor operating systems such as *Mach* [33] emerged. But even with these specifically designed operating systems scalability problems arose when the number of processors significantly increased [2]. In many cases these problems were due to an inappropriate kernel structure.

In shared memory systems, the granularity of program units that can be executed in parallel is determined by the performance of the process model. The most important aspect hereby is the overhead for process creation, synchronization, and switching. Crucial points always are the vertical interactions between the (non-privileged) application process and the (privileged) operating system kernel. These interactions are extremely heavyweight when compared to local program activities. Most recently, approaches have been made aimed at reducing kernel interactions in the course of context switches to an absolute minimum [18]. However, completely bypassing the kernel is not feasible. Both, monolithic and microkernel-based operating system architectures have in common that user and kernel space are physically isolated from each other. The separation is necessary in traditional multi-tasking and timesharing systems. But it is not a predominant requirement in the context of (massively) parallel computing.

A well-known fact is that shared memory computer architectures are no longer able to keep pace with the dramatically increasing performance requirements of parallel applications. Well established computer manufacturers identified this problem and decided to go the alternative way also, namely to refer to distributed memory architectures. Faced with this situation, parallel operating system design must not place the focus on "old-fashioned" shared memory computer architectures—although the design must not completely ignore these architectures.

A parallel operating system still must take care for the management of a (possibly) very large number of nodes, be able to tolerate node failures, provide I/O services, enable virtualization of nodes by implementing a proper process and address space model, and ease the programming of parallel applications. A site-transparent execution of application tasks must be supported, such that (static/dynamic) load balancing becomes feasible. This calls for access transparency, e.g. on files, I/O devices and other system services, but also on application processes. Bootstrapping the system and network-wide loading of processes must be supported. In this context, one of the major problems will be to synchronize global system activities and to detect the minimal functioning of the system. Centralized approaches are to be rejected as they will not only raise fault-tolerance problems but also result in a serious performance bottleneck. Last but not least, traditional system services are required such as process, memory, and address space management. Thus, parallel operating systems are requested to provide many services distributed operating systems typically provide. However, they should provide these services only on *extant-on-use* basis and not at all times as this will imply a potential performance bottleneck and an increase of system complexity.

The major difference to distributed (microkernel-based) operating systems is that quite a large number of applications accommodate their degree of parallelism to the degree of parallelism provided by the hardware. Hence, the operating system is not in all cases required to multiplex a single node between several tasks or even threads of control. Standard distributed operating systems, however, are supposed to do so. Of course are multi-tasking operating system kernels able to process "well-shaped" parallel applications whose tasks can be mapped in one-to-one correspondence with the nodes. In particular, this also holds for microkernel-based systems. However, all the microkernels presently available at the market fail to efficiently support the above mentioned type of parallel application [28]. The main problem is the artificial boundary between user and kernel. This boundary is due to the microkernel approach and not necessarily demanded by the parallel application. Even if the hardware supports direct network access from user mode and, thus, allows to bypass the kernel when global communication has to take place, yet overloading processing nodes with unneeded system functions does not only waste local memory resources[1] but also limits scalability in general.

For all these reasons discussed above and due to the lack of properly designed parallel operating systems, parallel computer manufacturers developed their own system software platforms. These platforms, however, cannot always be regarded as operating system but rather runtime environment. They were specifically developed for a certain distributed memory parallel computer, having a certain set of applications in mind. Examples are *CS/Tools* [21], *PARIX* [26], and *UBIK* [30]. Except the former mentioned one, these all are systems developed for Transputer-based architectures. The *iPSC/2 hypercube* with

[1] Virtual memory on the processing nodes is in almost every case sacrificed, not only for technical reasons. Data parallel programming of distributed memory machines typically makes very large local address spaces superfluous. Moreover, an "unlimited" local address space is in contradiction to the MIMD principle, namely to avoid the *von Neumann bottleneck* to the memory. Note that traditional virtual memory has only little in common with virtual shared memory [19]. The latter was not introduced to solve the problem of memory over-allocation, but rather to present a view of a global, common address space that is physically distributed. In particular, for performance reasons, this view can (and should) be implemented without relying on paging but on compiler and runtime system support.

its 128 nodes is controlled by *NX/2*, a parallel operating system providing virtual shared memory [20]. The operating system for the CM-5 [5], called CMOST, is a SunOS variant. This variant, however, is only executed on the control processors. Processing nodes are run by a runtime executive and, normally, will be subjected to single-tasking mode of operation. Nevertheless, there is also support for multi-tasking. But this means that the control processor, i.e., CMOST, preempts all tasks belonging to the same application remotely "at the same time". Multi-tasking is not an autonomous feature of the processing nodes[2].

These systems have two things in common: highly efficient network-wide communication and poor functionality. They all suffer from a design concept that makes it impossible to scale up with the manifold demands of parallel applications. Similarly, state of the art distributed (microkernel-based) operating systems suffer from a design concept that makes it impossible to scale down kernel functionality accordingly. This is where an innovative parallel operating system has its niche. The design is required to support scalability in the two directions. Functional enrichment must be possible. The design of parallel operating systems is assigned to provide a framework for the composition of application-oriented parallel computing platforms.

2.2 Design Constraints

Parallel computing defines its own "book of rules". No matter which principle is used for designing a parallel operating system, the resulting implementation must never hide performance. One of the most crucial performance limiting factors is embodied by the communication system and addresses the message startup time problem. Another important factor is the complexity and, thus, immense difficulty of programming massively parallel systems. This calls for new programming models and still involves large efforts in theory, language design, and compilation techniques. It is well known, that programming theses systems must be "*liberated from the von Neumann style*" [1], but there still are no satisfying programming models available. Compared to that, the operating system can only provide a very modest contribution to cope with the complexity of a machine consisting of thousands of interconnected nodes.

2.2.1 Message Startup Time

The communication performance problem in massively parallel computer architectures is dominated by the message startup time. Basically, the message startup time determines how many processor cycles are lost to local application program processing by performing remote interprocess communication. The loss of "number crunching" power caused thereby is not only due to the communication protocol overhead but also a question of the actual operating mode of the node [28]. A single-tasking mode of operation, e.g., does not imply any form of address space isolation. There is no need to protect either the tasks (since there is only one per node) or the kernel (since it only has to keep track of the resources of a single task): neither horizontal nor vertical (address space) isolation is required. Thus, the kernel is nothing but a communication library, sharing with the application task the

[2]This somewhat strange scheduling strategy is the result of providing direct user-level access onto the network hardware interface. However, the problem is not in granting the access but the lack of a properly designed hardware interface that supports multi-tasking on the node and enables user-level access.

same address space. Of course, this is inconceivable if a multi-tasking mode of operation is demanded by the application and, hence, must be supported on the node.

The choice of the proper operating mode depends on constraints defined by the application and the kernel architecture. There are a number of parallel applications that scale very well with the actual number of nodes. These applications call for single-tasking support on the nodes. The microkernel architecture promotes the encapsulation of system services by user-mode tasks. This approach demands multi-tasking support on the nodes even if only a single application task must be locally processed. In this case, the microkernel does not work for but against the parallel application if the (parallel) operating system was assigned to reduce the message startup time to an absolute minimum.

The message startup time mainly is determined by the *communication latency* of a single network-wide message passing operation. Latency is caused by all "vertical activities" needed at the source and destination node to send, receive, and deliver a message request (and not the message contents) to a process and all "horizontal activities" between the nodes to execute the communication protocol. That is to say, communication latency is the product of *node latency* and *network latency*. In massively parallel systems, node latency primarily is due to software overhead, i.e., the message passing kernel, whereas network latency depends on the hardware, i.e., the interconnection network. With the scalar performance of a 40 MIPS processor (RISC technology) in mind, communication latency must be below 10 μsec. As discussed in [22], only then a balanced ratio between the per-node computing power and network performance is achieved.

2.2.2 System Complexity

Programming and management of thousands of interconnected nodes is a non-trivial task—and it does not get better with millions of nodes. It is not only the problem of keeping track of all node resources and to associate nodes with application tasks. This still assumes that a proper programming model exists, enabling to generate that many tasks or threads of control. Also bootstrapping an operating system enters completely new dimensions. Getting an operating system instantaneously loaded over all the nodes is not only utopia but will also significantly slow down system startup time. In addition to the difficulties arising from the complexity of the computer architecture, the main software problem is the complexity of the operating system and the procedure that must be executed during the initialization process.

Basing on a *dynamically alterable operating system*, bootstrapping only must take care of the absolute minimal subset of system functions that need to be operable at any time. In addition to that, relying on *incremental loading* features, the consequence will be to solely bootstrap a single node and to continue bootstrapping of other nodes on demand. The nodes (and so are operating system services) must only be in operation when they are needed by some application. In final consequence the application always takes responsibility (i.e., is the reason) for node bootstrapping. It turns out that, on the basis of the adequate operating system structure, the majority of nodes of a massively parallel system must not be bootstrapped at all—incremental loading encompasses incremental bootstrapping. In such a context, *garbage collection* becomes indispensible as well. Just as system functions are loaded on demand, namely when invoked for the first time, they must also be deleted

automatically when their presence is no longer required.

These considerations lead to a situation in which emphasis for the system design must rest on the ability of being almost arbitrarily configurable. It thus tenders to model operating system services by ("medium-grained") objects and trying to reduce the configuration problem to the question of how to generally map objects onto a parallel machine. Programming massively parallel systems then mainly becomes a configuration problem. This is true for both the user application and the parallel operating system. In order to ease configuration, an object model must found the basis of programming. In this sense, massively parallel systems and distributed systems share a very common basis. Approaches stemming from the distributed systems area [15] become transferable to the area of massively parallel systems to aid the mapping of at least the operating system. Thus, the least minimum requirement in the development of parallel operating systems is to design a structure that eases configuration. The operating system must show an actual representation that logically consists of a number of "*transient objects*".

2.3 Design Principles

A solution to the problems discussed in the previous sections is to understand a parallel operating system as a *program family* and to use *object orientation* as fundamental implementation discipline. The former concept (program families) helps to prevent to design a monolithic system organization and the latter concept (object orientation) enables the efficient implementation of a highly modular system structure.

2.3.1 Program Families

The program family concept distinguishes between a *minimal subset of system functions*[3] and *minimal system extensions*. It does not prescribe 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* [13], with each new level being a new minimal basis, i.e., *virtual 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 far 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 get to be the final system extensions. The traditional boundary between application and operating system becomes indistinct. The operating system extends into the application, and vice versa. An incremental system design also promotes a dynamically alterable system structure. This approach, thus, is the key to success to overcome the bootstrap problem discussed earlier.

2.3.2 Object Orientation

Applying the family concept in the software design process leads to a highly modular structure. New system features are added to a given subset of system functions. Because of the

[3] The term "*minimal basis*" is used as synonym too.

strong analogy between the notions "program family" and "object orientation", it is almost natural to construct program families by using an object-oriented framework [4]. Both approaches are in a certain sense dual to each other. The minimal subset of system functions in the program family concept has its counterpart in the superclass of the object-oriented approach. Minimal system extensions then are introduced by means of subclassing. Inheritance and polymorphism are the proper mechanism to allow that different implementations of the same interface may coexist at the same time. Code reuse is significantly enhanced, increasing the commonalities of different family members:

> "We consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them" [25].

For the development of parallel operating systems it is sensible to chose, above all, the program family concept as fundamental design principle. Object orientation should primarily be used as implementation and not as design instrument. Inheritance makes it much easier to implement and maintain an incremental system design, but is not mandatory for it.

3 Functional Decomposition

The maxim for the design and development of operating systems for distributed memory parallel computers is to reduce the number of site-dependent components to an absolute minimum. Hereby, a site denotes either of cluster, node, processor, or address space. Whether or not a system component is sharing with other system components a specific site should be a matter of configuration and not implementation. A highly modular system structure is required. This structure must enable, but not enforce, the decoupling of functional units such that a high degree of decentralization becomes possible. In order to design a structure that meets the needs of (massively) parallel systems, the following three aspects give some directions:

- *The hardware architecture relies on a message passing system.* This requires some kind of message passing kernel acting as a software backplane to interconnect all the entities (i.e., user and/or system tasks) that are going to be executed by the parallel machine.
- *The parallel machine is of limited scalability.* Applications that were designed to operate in an environment with a larger (or even unlimited) number of nodes must be supported by a parallel computing platform virtually offering any degree of parallelism. Instead of dealing with physical processors (or nodes), precautions for a virtual processor model must be made.
- *The computing platform is a functionally dedicated system.* Not all nodes are equipped with the same set of peripherals and are executing the same software. Most of the nodes are used for number crunching purposes and not every node is directly connected to a mass storage device, for example.

This suggests a system organization, in which each aspect is covered by a separate building block, leading to three major subsystems. Two basic rules motivate this step. The first rule is that splitting up a system into subsystems leads to a general division of complexity and, thus, makes the problem of constructing a parallel operating system more understandable and manageable. The second rule is that a modular structure of well-defined subsystems reduces the number of node-bounded system components to an absolute minimum.

With these aspects in mind, since entities must be able to cooperate with each other, the software backplane has to act as minimal subset of system functions. Providing the notion of virtual processors is been done by introducing minimal extensions to the minimal basis, since not every application will acquire more nodes than are physically available. Similarly, services that depend on the availability of specific devices are qualified as system extensions. The same holds for services that are only used in the context of specific applications.

3.1 Macrostructure

Every family member is constructed from three major building blocks (Figure 1). These building blocks are the *nucleus*, the *kernel*, and POSE, the *Parallel Operating System Extension*. 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 blocks over the nodes of the parallel machine.

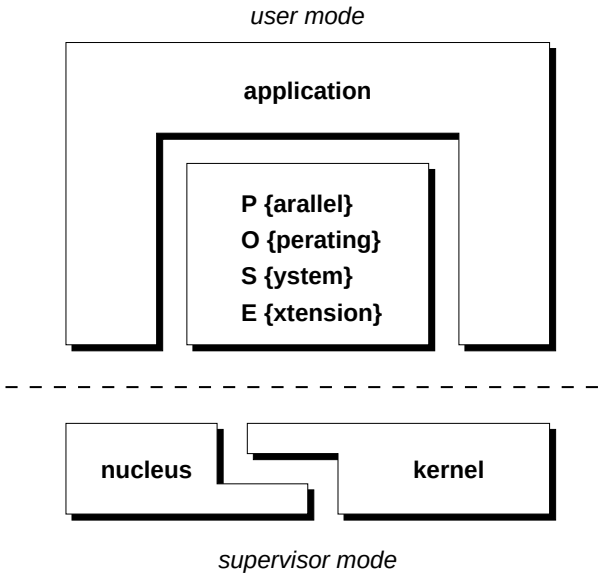

Figure 1: Building blocks

The nucleus implements system-wide interprocess communication and provides a run-time executive for the processing of threads. It acts as minimal basis and 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, load balancing, and (inter-) networking to provide some host access are performed by POSE.

Kernel and POSE services are built by active objects implemented by lightweight processes. In contrast, the nucleus is an ensemble of passive objects that schedule active objects. Each service that is provided by both POSE and kernel is implemented by an entity that may consist of several active objects, enabling concurrent service processing. An entity provides a common execution domain for active objects and defines the *unit of distribution*, while an active object is the *unit of execution*.

Nucleus and kernel constitute the *kernel entity*. This entity defines an abstract processor for the execution of possibly multi-threaded tasks. The sole purpose of the kernel entity is to provide hardware abstractions that make (parallel) applications and POSE independent from the physical characteristics of a given processing node. This also includes providing the concept of "logical nodes".

3.2 Extensibility and Configurability

Entities are system extensions. They are loaded on demand and (in most cases) can be arbitrarily distributed over the nodes of the parallel machine. The need for having a specific entity running on a particular node is expressed either by the application or by a system administrator. In the former case, the entity gets to be loaded when the service it represents is called for the first time. The entity vanishes when this service is no longer needed. This may happen, for instance, upon termination of the application entity that demanded the service. Note that in this model, application entities are not distinguished from system entities. In other words, provided that the application is properly structured, incremental loading of parallel applications is free of charge.

In the system administrator case, an initial configuration description specifies which entity must be initially loaded (i.e., bootstrapped). This procedure loads "opaque" entities from some bootstrap device (usually, disk or network) and stores them into the local memories of the nodes. With understanding parallel programs as a group of such related entities, bootstrapping also can be applied to get applications running. The purest form of batch processing is achieved.

Either way, POSE is a dynamically alterable building block. The design specifies that POSE alters its shape with respect to the applications that are to be executed by the parallel machine. Thus, applications determine the actual complexity of the parallel operating system. In final consequence, they are responsible for bringing (i.e., bootstrapping) the operating system on the machine.

Incrementally loading the operating system does not mean to place POSE entities onto the node where the demanding application task is residing. Since these entities are qualified as site-independent, they can be loaded on any appropriate node. In order to achieve an "optimal" mapping, expressing some nearest neighbor relationship between the entity

and either of higher-level application or lower-level hardware/software component may be sensible. This is where the configuration aspect comes in. The entity is to be attributed properly and the attributes must be interpreted by some low-level load balancing service.

3.3 User and Supervisor Mode

Logically, application and PoSE are assumed to be executed in the non-priviledged processing mode (*user mode*) of the underlying processor. In contrast, nucleus and kernel are assigned for execution in privileged mode (*supervisor mode*). Thus, the logical design defines an artificial boundary to isolate these two components from other entities. This does not mean that the actual system representation always must show for isolated system components. Rather, the purpose of introducing the boundary is to guarantee downward scalability.

The system design must be complete for being able to later integrate the version of a system component of significantly lesser (i.e., scaled-down) functionality. Therefore, it is of importance to clearly identify those components that, under certain circumstances, must be executed in privileged mode and others that can be executed in non-privileged mode. Note that supervisor mode components also constitute the set of (logically) node-bounded entities. The complexity of this set has significant impact on how long it takes getting the parallel machine started. By designing a system structure that exhibits a fluent boundary between user and supervisor mode the number of node-bounded entities becomes dependent on the application that must be supported.

This design especially promotes address space sharing between all entities of the same node, provided that the structure of the application allows to do so. In a single-tasking mode of operation, e.g., application, PoSE, kernel, and nucleus all may execute within a single address space. However, it must be clear that only in the presence of secure programming languages and in the absence of temporary node failures the model of a single address space can be considered as being safe. Nevertheless, this model still is applicable for mature applications—and, of course, for mature system components—to prevent per-node overhead in crossing subsystem boundaries. As a consequence, since it is not possible to safely protect user domains, the entire parallel machine then must be operated in batch mode.

3.4 Functional Hierarchy

The functional hierarchy defined between PoSE, the kernel, and the nucleus makes possible a very high degree of decentralization (Figure 2). All components are encapsulated by (active/passive) objects, which requires object invocation schemes to request service execution. Well-defined interfaces enforce a clean separation of all building blocks. Different styles of calling sequences, to pass the interface, then basically distinguish between single and multiple address space system configurations.

3.4.1 Invocation Schemes

Nucleus services are made available to the application and PoSE via *Nearby Object Invocation* (NoI). The logical design assumes a separation of the nucleus from the application

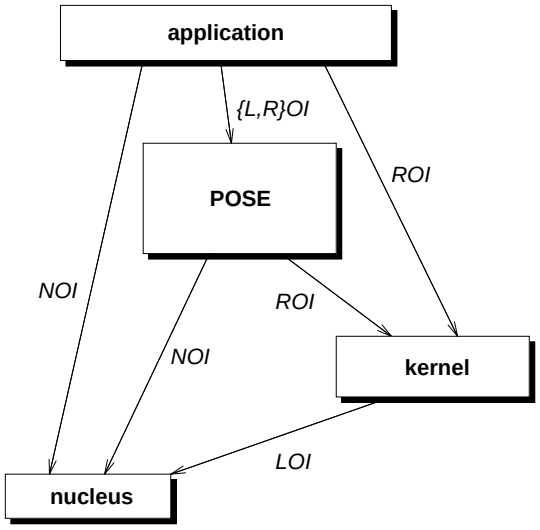

Figure 2: Problem-oriented invocation schemes

(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 happen. The nucleus is "nearby" the using entity. It shares with the entity the same node, but not necessarily the same address space segment. Since kernel and nucleus together reside in the same address space, the kernel performs *Local Object Invocation* (LOI) to request nucleus services.

Kernel services are made available via *Remote Object Invocation* (ROI). The ROI scheme always implies context switching, but not necessarily address space switching [24]. 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 logically based on NOI. However, since nucleus and kernel together define a single program, nucleus primitives invoked by the kernel to either issue or accept ROI requests hence will not be activated via NOI but LOI.

Services of POSE are requested via LOI and ROI. The former scheme is used to interact with the POSE passive objects (i.e., 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 POSE object but entirely on a library basis.

3.4.2 Actual Structure

From the design point of view neither the kernel nor POSE need to be present on every node, but only the nucleus. In a specific configuration, the majority of the nodes of a massively parallel machine is 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 all not 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. The building blocks have been designed by considering the various schemes of object invocation (Figure 2). However, it depends on the actual operating system family member whether these schemes become effective as specified by the design or can be replaced by a more simple and efficient alternative.

Although the functional hierarchy assumes NOI for the interaction between application (POSE) and nucleus, the LOI scheme is used for those members of the kernel family which 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 user and supervisor mode of operation (*vertical isolation*) nor a separation between competing tasks (*horizontal isolation*). In this case NOI actually means LOI. 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. Both variants basically distinguish between single-tasking (no isolation) and multi-tasking (isolation) mode of operation. They implement different members of the kernel family.

A single-tasking mode of operation implies that only a single address space is supported on the node. As was pointed out above, NOI then takes the form of LOI. This also means that all interactions with the local residing kernel happen via LOI too. Thus, instead of ROI, as specified by the design (see also Figure 2), the more efficient alternative is applied in that case. Note that this configuration sacrifices the address space boundary between higher-level entities and the nucleus. Since the nucleus is part of the kernel address space, sacrificing nucleus isolation also implies sacrificing kernel isolation. As the kernel "uses"[25] the nucleus, and thus depends on its correctness, it makes little sense to isolate the kernel in this situation. Nevertheless, kernel services still can and must be provided via ROI. In other words, there are different invocation schemes existent at the same time to interact with the kernel: LIO is employed by co-located entities whereas ROI serves to execute kernel requests issued by remote residing (user/system) entities.

3.5 Microstructure

The minimal subset of system functions defined so far is a compromise between scaling transparency and efficiency. In fact, the compromise was made only with respect to the interface specification of the abstract data type representing this subset. The internal behavior of the abstract data type is manifold and basically distinguishes between single and multi-tasking mode of operations. That subset defines a "minimal but perfect basis" for distributed memory parallel applications.

Offering dedicated single-tasking and multi-tasking kernel implementations enables the user to deal with the tradeoff between performance and functionality on an individual basis. Parallel applications whose tasks can be mapped in one-to-one correspondence with the nodes are not punished by multi-tasking overhead, as these applications will be supported by a single-tasking kernel. Experiences show that this overhead takes up to 74 % of the message startup time when executing a user-level *send-receive-reply* sequence [28]. Note, a

state of the art microkernel however supports only multi-tasking and, thus, unnecessarily drains computing power from a single-tasking application.

The minimal subset of system functions is represented by a *kernel family* that implements four different operation modes (Figure 3). Each operation mode is represented by a subfamily. The entire family tree shows different nucleus versions, with the root (top) being the most simple and the leaf (bottom) being the most complex instance. As functionality and complexity increases, node and, thus, communication latency increases.

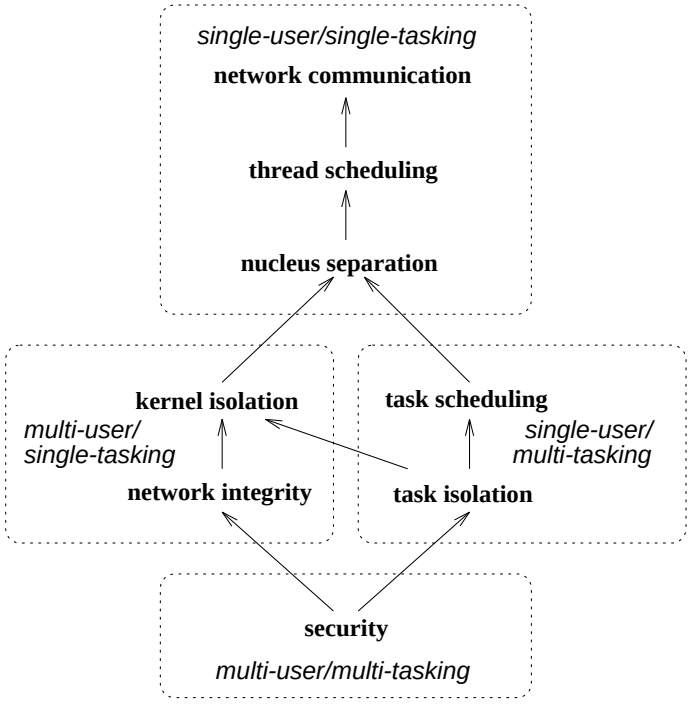

Figure 3: Nucleus family tree

This organization of the nucleus (i.e., the kernel entity) is one of the major differences to state of the art microkernel designs. The kernel entity defined is no single microkernel, but a "*microkernel family*". It can be adapted to the individual needs of parallel and distributed applications by providing family members that exhibit different system characteristics regarding functionality and performance.

3.5.1 Single-User/Single-Tasking

The root of the family tree shows three different nucleus instances supporting single-user/single-tasking mode of operation, with tasks possibly being multi-threaded. This operation mode allows only one user application at a time and merely supports the processing of one task per node. A single address space model is implemented. Consequently, the entire parallel machine is subjected to batch processing. This enables the most efficient execution of the parallel program and at the same time leads to the most inefficient utilization of the

parallel machine, since the machine cannot be shared between a number of applications.

Processing of non-dynamic parallel applications is readily supported by the *network communication* instance. At this stage, thread dispatching is non-preemptive and completely under the control of the application task. This nucleus representation is extremely lightweight and supports the notion of *featherweight processes*. Featherweight processes are a scaled-down version of lightweight processes. Each of them implement the purest form of a unit of execution, without consideration of any protection and security measures.

Thread preemption is the minimal extension to network communication. This means *thread scheduling* on a timer basis, actually introducing a second scheduling level. This level implements CPU protection [27], since a single thread (or task) is no longer able to seize the processor. The additional level knows the *bundle* as scheduling unit. A bundle consists of one or more threads, with non-preemptive scheduling of threads of the same bundle. Preemptive scheduling only happens between bundles. Bundles are thus autonomous scheduling units and are given a limited time quantum during which execution proceeds without scheduler intervention. As a consequence, threads execute no longer under the control of the application task but the system (i.e., nucleus). This enables to safely integrate kernel-level lightweight processes that implement the service access point for ROI. These threads then are combined into a *kernel bundle*, while all other threads form the *application bundle*. By that means, kernel service processing works independently from the processing of the application task. It ensures that remotely requested services are processed latest when the thread scheduler gains control and selects a *clerk* [24] for execution—provided that the application task sharing with the kernel entity the same address space is "well-behaved"[4].

The next nucleus instance does not really introduce additional functionality, but provides the platform for multi-tasking and/or multi-user mode of operations. These operation modes call for isolation and protection measures. Thus, the *nucleus separation* instance introduces as minimal extensions to thread scheduling NOI and ROI patterns in order to request services from the nucleus and the kernel, respectively. This kernel family member provides a nucleus trap interface. Higher-level entities are physically uncoupled from nucleus code. Since every nucleus and kernel instance takes the form of an abstract data type, these entities are also logically uncoupled from kernel-level data. This supports dynamically changing the operating mode of a node by replacing the actual kernel entity by a different implementation. However, note that only code separation is supported, but not memory protection. As a consequence, the passing of complex data structures between the kernel entity and higher-level entities is straightforward and involves no MMU programming.

3.5.2 Multi-User/Single-Tasking

In a distributed memory parallel machine, multi-user mode of operation is feasible even if only a one-to-one correspondence between tasks and nodes is supported. The entire multi-node machine can be allocated to different users at the same time. For this purpose, different *user partitions* are provided, with every user application executing within a private

[4]With scheduling bundles of threads that reside within the same address space, in a shared-memory multiprocessor environment, several processors then may be in charge of executing different bundles. In this model, an application task consists of several bundles to take advantage of preemption and of the shared-memory processor architecture.

partition. Obviously, this does not require local ("on-board") security measures to protect the tasks from each other, but it requires to protect the network interface. By direct network access the user task is able to intrude the network and, thus, tasks mapped to different user partitions.

In order to provide a multi-user function, the kernel entity must be entirely isolated. Memory protection is to be introduced as minimal extension to nucleus separation. This results in a new family member, providing *kernel isolation*. Because the nucleus is part of the kernel entity, applying memory protection to the nucleus also implies the isolation of parts of the kernel address space. In this case, nucleus invocation, i.e., crossing the trap interface, usually does not cause increased overhead. However, the passing of data structures gets more heavyweight. It mainly depends on the MMU and on the (kernel) address space model whether or not additional overhead is introduced[5]. Anyway, the increase of nucleus functionality is encompassed by a potential loss of communication performance.

Once the kernel entity is protected, tasks are no longer capable to intrude lower-level software components and, thus, bypass protection domains. However, in order to communicate, tasks need to know communication endpoint (i.e., thread) identifiers, and these identifiers must be passed down to the nucleus along with a communication request. In a multi-user configuration, tasks mapped to different user domains must therefore be prevented from being able to successfully apply communication endpoint identifiers of each others domain. This is the purpose of the minimal extension to kernel isolation, namely *network integrity*.

In order to model and enforce protection domains, a capability-based approach [7] is used. This approach grants object access only if a thread (i.e., subject) is in the possession of that object or one of its proxies. An object must be created and bound to a system-wide unique identifier before it can be used. It is assumed that a unique identifier cannot be deduced [23]. In order to achieve this, the nucleus generates a random number which, combined with a global hash key, is used to make identifiers system-wide unique. Note that this procedure works autonomously and needs not be controlled by a central system component. The creator implicitly possesses the newly created object and henceforth acts as its owner. It is the autonomous decision of the owner to make the object either globally known, by exporting its unique identifier, or even accessible, by exporting its *proxy object* [24]. Thus, the access domain of an object may be extended only through the owner. As a consequence, the nucleus must not verify that communication to an active object is allowed but only whether this object exists—knowing the capability of this object enables communication.

3.5.3 Single-User/Multi-Tasking

The scheduling paradigm which was introduced by the thread scheduling instance does not distinguish between multiple autonomous program entities. Merely execution of a single application program (i.e., task) is supported. With having introduced nucleus separation, the processing of several application programs becomes possible by slightly extending the

[5] Usually, the address space of the currently executing thread is mapped into the kernel address space. This enables direct access onto its user address space when the thread temporarily executes in kernel space, unless the MMU requires, though, to explicitly classify each access as a user mode address space operation. In that case, simple memory load/store instructions may easily expand to heavyweight read/write procedures.

scheduling functions. The nucleus can be easily shared by a number of application tasks, as it exploits a trap mechanism to provide a common entry point for independently generated programs.

A task is represented as a *team* of lightweight processes, with the team possibly exhibiting several thread bundles that can be autonomously scheduled. The team owns resources such as message buffers, memory space for stacks, process descriptors; and *task scheduling* then means to additionally keep track of the resources owned in order to select the next task ready to run. A scheduling discipline may be favorable that takes care of the current load and, thus, is trying to improve global system utilization. This further calls for minimal scheduler extensions to those already provided by nucleus separation.

At this stage, multi-tasking can be readily supported even if private address spaces are not provided. A private address space serves for two basic purposes. On the one hand it implements memory protection, isolating programs from each other. On the other hand it defines a logical address space to execute a program, enabling code/data relocation at runtime. Being relocatable is also a property of *position independent* code, which then needs to be generated by a compiler. In addition, the use of secure programming languages supports program isolation without the necessity of address space protection hardware.

For all these reasons an intermediate step was made. This step introduces the task scheduling instance as a platform for multi-tasking. A minimal extension to this platform is introduced with *task isolation*, encorporating full address space protection. This extension is used to generally improve system availability and in those cases where neither the programming language nor the compiler provides adequate support for a single address space multi-tasking mode of operation. Task isolation extends not only task scheduling but also kernel isolation. The latter family member already provides vertical isolation to protect the kernel entity from higher-level user entities. With task isolation, the nucleus is also functionally enriched by measures of horizontal isolation to protect user entities from each other.

3.5.4 Multi-User/Multi-Tasking

By combining the functions of network integrity and task isolation, a new nucleus instance emerges. This instance introduces multi-user *security*. Multi-user security in a distributed environment demands communication firewalls and protected address spaces. The former is achieved by ensuring network integrity and the latter is achieved by enforcing task isolation. Obviously, the peak of system functionality has been reached, however only at the cost of system performance.

3.6 Communication System

The nucleus performs network-wide message passing between processes and address spaces and needs a communication system for this purpose. The communication system is asked to (1) keep track of interactions between remote residing processes (i.e., active objects), (2) execute some sort of transport protocol, and (3) control the network hardware interface. A communication system emerges that exhibits three *problem-oriented protocol layers* (Figure 4).

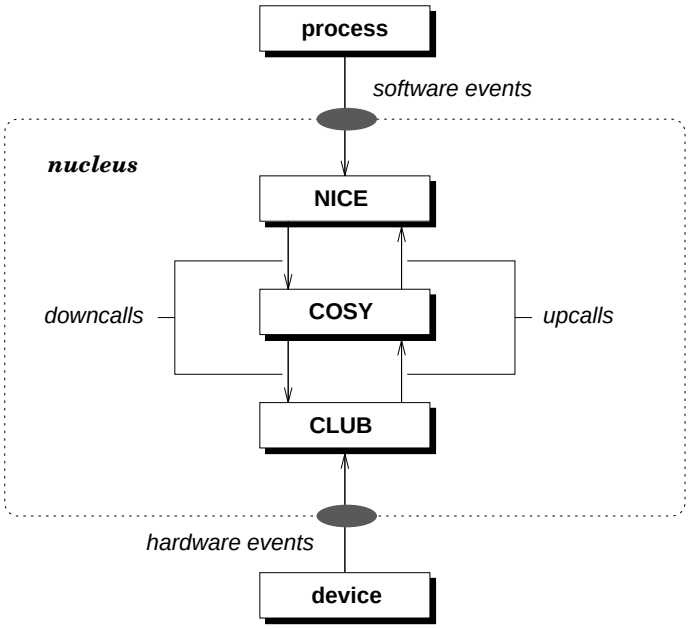

Figure 4: Communication system architecture

Interactions between the layers happen via *downcalls* and *upcalls* [3]. Queues are used where possible to decouple the different flows of control. Calls in either direction are virtually asynchronous, since it depends on the actual load and on the nucleus configuration of whether message transfer requests must be queued or can be immediately carried out. The communication protocols implemented by these three layers are also referred to as NC^2, which stands for "NICE–COSY–CLUB".

3.6.1 Inter-Nucleus Protocol

The nucleus supports network-wide communication between both processes (i.e., threads) and address spaces (i.e., teams). Interprocess communication is synchronous and *packet-oriented* whereas the communication between entire teams works *segment-oriented* and is asynchronous. In order to communicate, the locality of processes (active objects) and/or team address spaces (passive objects) must be known. This is the task of the top layer of the communication system. It determines the correlation between active/passive objects and the nodes where these objects are residing.

Network-wide communication between objects is supported by NICE (*Network Independent Communication Executive*). The NICE layer implements the *inter-nucleus protocol*. It extends into a network environment local functions dealing with the control of processes and address spaces. Network communication is not been done by NICE, but by some lower level communication system (i.e., COSY). Instead, state transitions of processes and address spaces are controlled to logically enable end-to-end data transfers without the need for intermediate buffering.

The base function of the inter-nucleus protocol is to verify the presence of communication endpoints. This measure is to ensure availability of address spaces that are either source or destination of a data transfer. The basic inter-nucleus protocol primarily encapsulates resource management strategies. It does not implement security measures. Since not in every configuration the parallel machine is run in a multi-user mode of operation, security is implemented as minimal extension to a minimal subset of NICE functions.

3.6.2 Transport Protocol

Data transfer is accomplished by COSY (*COmmunication SYstem*). This layer encapsulates *transport protocol* functions and provides an abstraction from actually given network capabilities. Depending on these capabilities, COSY is more or less complex. It provides a *protocol family* and not only a single implementation for all possible system configurations.

Logically, COSY takes responsibility for a secured data transport of arbitrarily sized messages. However, "logical" does also mean a configuration in which COSY is not in charge of any network activities. That extreme situation arises when the network hardware itself is capable of transferring message streams in a manner required by parallel applications. In those cases, COSY simply passes through all requests from and to NICE, without interpretation.

COSY supports the concept of *virtual channels*. Downcalls from NICE, that have network relevance and thus imply a message transfer, are furnished with logical addresses of the nodes to which these calls are directed. A logical connection between the sending and the receiving node is established and made known by generating a virtual channel identifier. In reality, a virtual channel is nothing but a number that is used as hash key to locate protocol objects that manage a specific connection. Upcalls from CLUB deliver virtual channel identifiers as part of the COSY packet that was just received.

The protocol objects COSY implements distinguish between different data transport functions. They are tuned with respect to the given class of network, however without having knowledge of how this network is really accessed. Access transparency in this case is rendered possible by CLUB. Transport protocol functions for message segmenting and reassembly, e.g., are provided when it is required by the network or even by the application.

3.6.3 Network Device Protocol

Abstraction from the physical network interface is been done by CLUB (*CLUster Bus*). The bottom layer of the communication system, thus, encapsulates the network device and attaches the nucleus physically to the network. This layer implements the *network device driver*.

CLUB provides the view of an abstract network device that may have several physical representations. The CLUB abstraction makes COSY independent from the network device actually used, no matter of whether this device is a physical or a logical one. By that means, the portability of COSY protocols is supported. It also provides a framework for the development of communication protocols, as it becomes possible to test new implementations under the control of some host operating system instead of being compelled to use the bare hardware.

Another important aspect of CLUB is to allow coexisting COSY entities to share a single network device. A sending COSY entity provides additional demultiplex information used by the receiving CLUB instance to select the proper transport protocol entity for the incoming message[6]. Which pair of COSY entities becomes effective is an attribute of the message sending process. In principle, CLUB performs *dynamical binding* of upcalls to COSY across the network. A message object that is going to be transmitted by means of CLUB encodes the upcall handler (i.e., method) to be used for the delivery of that object to the peer COSY entity—a CLUB message is some sort of "*Active Message*" [31].

3.7 Communication Latency

The most crucial aspect of MIMD systems is the message passing performance, in particular the message startup time. As was mentioned earlier, the message startup time depends on the *communication latency* of a network-wide message transaction. The communication latency depends on (1) the *node latency* for sending and receiving a message to/from the network and (2) the *network latency* for actually transferring the message to the peer node. Node latency is mainly a software issue, whereas network latency primarily depends on the communication hardware. Thus, in order to reduce the message startup time, the node latency should be kept as small as possible—and this calls for a proper design and implementation of the nucleus and the embedded communication system.

The "coarse grain" approach is to rely on a kernel family. That is to say, node latency is reduced by letting applications only pay for the service functions really needed. This approach is refined basing on a communication paradigm by means of which arbitrarily sized messages are exchanged (1) in a pipelined manner and (2) directly between the address spaces of the communicating threads without intermediate buffering. The transfer of a memory segment basically means to send a message header immediately followed by a message trailer. At the receiving site, the message header is used to announce the arrival of a message segment and to enable the delivery of the incoming data to the destination address space. The basic idea is the following:

- overlap the header transmission with the procedure locally executed to setup the message trailer;
- overlap the trailer transmission with the procedure remotely executed to deliver the message segment.

Trying to overlap the above mentioned communication phases is motivated by the "short-term buffering" capabilities of the network. Typically, there are two categories of buffers. These are the "communication wire" and the sending and receiving FIFO registers of the network hardware interface. If DMA is exploited in the transfer procedure, an overlap may be given only if the memory bandwidth is not fully utilized in order to send the message out of the node. As long as the CPU can run memory cycles and/or primarily performs code/data cache accesses, the fetch-execute-cycle is not delayed and operates with full speed.

[6] This feature is required if the same network is the only means by which node bootstrapping is possible. In this case, "normal" messages must be distinguished from "bootstrap" messages.

This makes the processing of CPU instructions (in particular, nucleus software) appear in parallel to the network-wide transfer of memory cells.

The scenario of trying to overlap message setup and transmission phases is illustrated in Figure 5, which shows a fine-grain breakdown of the communication latency. Network latency basically consists of two parts: *packet latency* (i.e., message header transmission) and *segment latency* (i.e., message trailer transmission). It depends on the network architecture whether both parts will accumulate to the same delay. Note that the network may be tuned to specifically support the transfer of small and fixed size packets [5]. Node latency is the sum of *sender latency* and *receiver latency*. The former is due to the header and trailer setup procedure, referred to as *header latency* and *trailer latency*, respectively. The latter is due to (1) signaling and handling a communication interrupt, (2) receiving a packet from the network, (3) forwarding this packet to NICE, (4) verifying the destination address space segment, and (5) accepting the message segment from the network. The last four steps determine the *delivery latency* of a segment and are the most crucial factors in trying to reduce the message startup time. In principle, these steps are performed by the portable part of the nucleus. The first step listed relates to the non-portable nucleus part and is highly processor-dependent. The play for tuning the receiver latency, thus, is the fraction made up by the delivery latency.

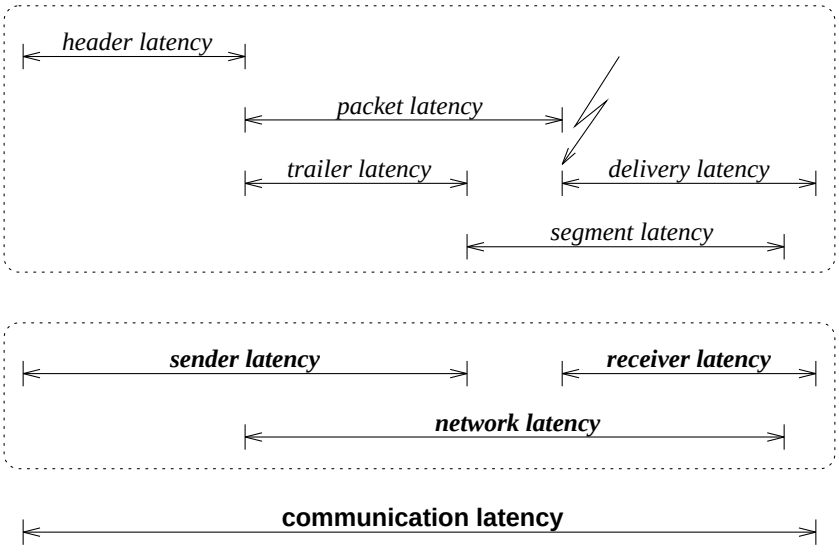

Figure 5: Message delivery latency

Achieving message delivery on time is a question of the receiver latency. The longer the execution path is between signaling the communication interrupt (in order to receive the message header) and accepting the message segment, the longer the receiver latency is. The length of this path particularly depends on the processor architecture and on the operating mode of the receiver node. State of the art RISC processors may cause a prohibitively large

performance decrease of interrupt-driven communication systems[7]. A single-tasking mode of operation calls for significantly less nucleus functions that must be processed in the receive procedure than a multi-tasking mode. Consequently, delivery latency of a single-tasking node is smaller.

In order to overcome the receiver latency problem two solutions are possible. The first solution applies a different communication pattern. This pattern, e.g., will cause the COSY entity at the receiver site to explicitly signal when to transfer the message trailer, which requires to execute a *handshake protocol*. Similarly, the COSY entity at the sender site only transmits the header and then waits on the signal to start the trailer transfer. The second solution is to still rely on the original communication pattern but introduce a *delay slot* between the transfer of message header and trailer. The delay slot is computed according to the following formula:

$$\begin{aligned} T_{packet} &= L_{packet} + L_{delivery} \\ T_{segment} &= L_{trailer} + L_{segment} \end{aligned}$$

$$T_{delay} = T_{packet} - MIN(T_{segment}, T_{packet})$$

However, introducing delay slots is a meaningful undertaking only if message delivery within a known and fixed time frame can be guaranteed. This calls for *real-time capabilities* of all nucleus instances involved in network-wide high-volume data transfer. In particular, it requires NC^2 to implement a (scaled-down) *real-time communication protocol* [9]. In addition to that, the network hardware must provide guarantees for the allocation of packet transfer slots. There must be a fixed and known (worst-case) delay from which the occurrence of the remote communication interrupt due to a transmitted header packet can be deduced. This delay is the packet latency.

4 System Composition

A number of *problem-oriented system configurations* become possible by combining the building blocks nucleus, kernel, and POSE in a manner required by the hardware architecture and/or demanded by the parallel application. In the following, a distinction between a native operating system for distributed memory parallel computers and a parallel computing platform for (high-performance) distributed systems is made. It is demonstrated that the designed system architecture is applicable in completely different environments.

4.1 Native Operating System

A *native operating system* runs on the bare hardware. It is assigned to control all hardware resources without assistance of some other operating system. Being of native type mainly is

[7] For example, the PEACE trap handler for the i860 is about 2000 lines of C code and has to take into account a number of obstacles defined by the pipeline architecture of this processor. Assuming 40 MIPS performance, a single interrupt may cause a delay of approx. 50 μsec before the first (C++) interrupt handler statement is executed. This is about three times slower than in the case of a 2 MIPS mc68020, which is a CISC processor. Effectively, there is a factor of 55 in interrupt handling startup time decrease, although the i860 has a 20 times higher scalar performance than the mc68020.

the question of how to isolate and implement the hardware-dependent system components. Hardware dependency also means site dependency, namely that a device driver, e.g., has to reside on the node the device to be controlled is attached to. As discussed later on, replacing carefully isolated components may not only imply the portage onto completely different hardware platforms but also to transform a native operating system into a user-level runtime executive.

There are two system components encapsulating site-dependent functions. These are the nucleus and the kernel. The nucleus is site-dependent because it acts as the minimal basis needed to interconnect cooperating active objects of a parallel/distributed application. Since the nucleus supports network-wide communication and preemptive scheduling, hardware-dependent modules are used as abstractions from a clock device and a network device. In addition to that, the nucleus shows for hardware-dependent modules that implement processor abstractions needed to perform process switching and interrupt synchronization.

The kernel is site-dependent because it encapsulates additional device driver functions used to support specific application and/or system configurations. Examples are the MMU driver and the disk driver, the former necessary to implement multiple address spaces or virtually shared memory and the latter required for file handling purposes. In addition to these hardware-dependent features, the kernel also is assigned to implement a dynamical process model. This calls for process management functions that are not necessarily hardware but site-dependent. Process creation, e.g., means allocation of a process control block on a specific node and this can be done only by kernel modules residing on that node.

POSE is classified as site-independent and, thus, hardware-independent in terms of portability. Although providing memory management and file handling functions, e.g., POSE is hardware-independent since it bases on hardware abstractions introduced by the kernel. Both examples depend on abstractions from low-level hardware capabilities and are not required to be co-located with the hardware devices that effectively carry out the management functions, namely to setup some address space segment or read/write a disk block.

For the reasons mentioned above, POSE is arbitrarily distributable across the parallel machine. Since the presence of the kernel is only required in the context of specific application or system structures, nodes may also be equipped with the nucleus solely. This leads to a number of possible node configurations as illustrated in Figure 6 (shaded boxes intimate the application building block).

4.1.1 User-Specific Nodes

A user-specific node is equipped with a minimal subset of system functions only. Focus is on application processing and this means to entirely sacrifice on POSE. That is to say, POSE services are provided elsewhere, but never on a user-specific node. These services are requested on ROI basis and, therefore, transparently accessible from remote.

With this background, only nucleus and kernel are provided as local system support. However, the kernel is required only under the following circumstances:

- the node has globally accessible devices attached to,
- the node is to be managed/controlled from remote, and/or
- the application calls for *scaling transparency*.

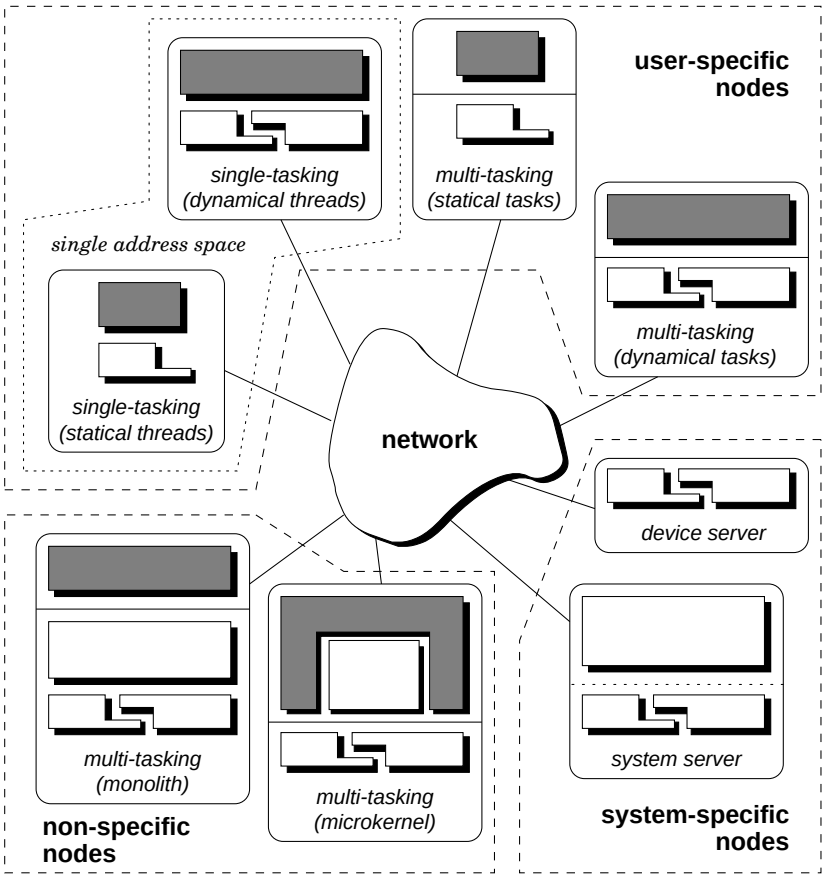

Figure 6: Operating mode variety

In the first case, the device driver simply exhibits a ROI interface. The second case works similar and, for instance, enables remote residing system components to force a "warm startup" and to construct/destruct active objects and/or address spaces. The third case assumes support for dynamical process management to enable many-to-one correspondence between application tasks and nodes.

Applications that scale well with the actual number of available nodes only call for a single-tasking mode of operation. In this mode, a further distinction is made between configurations supporting *statical threads* and *dynamical threads*. In the statical case, multiple threads of control are established as a "side-effect" of bootstrapping. These threads then exist as long as the node is executing the single application task. As a consequence, only the nucleus is required for application processing. In the dynamical case, multiple threads of control are dynamically created and destroyed. This feature is provided by the kernel. The nucleus remains unchanged and the kernel adds minimal nucleus extensions used for thread management.

The main characteristic of the statical and dynamical threads model, however, is the *single address space* on a node. Thus, nucleus and kernel take the form of a runtime library

that is directly linked to the application task. All local system functions are requested on a LOI basis. The kernel additionally offers a ROI interface to remote residing system components. This single address space approach reduces the message startup time to an absolute minimum, since protection domains are not implemented and, therefore, must not be bridged in the course of processing communication primitives. The single address space, thus, is the most important prerequisite to make lightweight processes even featherweight.

Scaling transparency calls for multi-tasking mode of operation. This implies *multiple address spaces* per node. A further refinement leads to the distinction between *statical tasks* and *dynamical tasks*. This is a similar distinction as the one made in the case of single address spaces. A library implementation of nucleus and kernel, as in the single address space variant, is no longer feasible. Rather, the kernel family member providing nucleus separation (refer to Section 3.5) is to be used as minimal basis on that node.

4.1.2 System-Specific Nodes

A system-specific node is used as server station and provides globally accessible operating system services. These nodes provide services that either have been off-loaded from user-specific nodes or are required to control devices. That is to say, system-specific nodes may host site-dependent as well as site-independent services.

One scenario in establishing a system-specific node is to have only nucleus and kernel running, with the kernel encapsulating one or more node-specific device drivers. In this case, a *device server* node is configured. This node provides remote device access on ROI basis. The nucleus is used to enable cooperation with the device driver (an active object) residing on that node. A typical example is to introduce a disk node. The main purpose of the kernel-level disk driver then is to implement disk block caching on that node and to provide cache read and write operations via ROI. In this case, file system functions are provided elsewhere and may even be available on a library basis, only. Since ROI is used to interact with the device, applications are unaware of the fact that disk I/O is not really performed by a user-specific node, for instance. This also is true with file management functions taking the form of a user-level library, thus hiding the disk driver (i.e., cache) interface.

The disk driver example explained above may cause problems with several application tasks going to access the same disk partition, i.e., file. In order to prevent data inconsistencies, a file cache consistency protocol is to be executed. This can be readily achieved on a library basis as well. Another and more straightforward approach, however, is to add a file manager to the device server node. This introduces a *system server* node that does not only make devices globally accessible but is also assigned to provide general (site-independent) operating system services.

File management is only one example to establish a system server node. Naming, process management, address space management, paging (e.g., in a virtually shared memory context), trap handling, incremental loading, load balancing, networking are further examples—and the list could be continued. That is to say, almost any POSE service is a candidate for being off-loaded from user-specific nodes and, thus, establishing a system server node. All these services are provided on ROI basis and are therefore accessible crossing node boundaries.

The device server configuration also implies a single address space model. Solely the kernel entity (i.e., nucleus and kernel) is executed on the node. In the system server case, address space isolation may be required. However, whether or not this really needs to be done depends on the number and type of PoSE services being executed by the node. Separation between user and supervisor mode could be an issue, just as address space protection. Logically, a system server node implies a multi address space model.

4.1.3 Non-Specific Nodes

A non-specific node executes user as well as system tasks. That is to say, in addition to nucleus and kernel, non-specific nodes also exhibit the PoSE and application building block. With PoSE being executed in user mode, a configuration emerges that compares to a typical *microkernel* organization. In this case, nucleus and kernel provide a (supervisor mode) platform on top of which microkernel-based user and system applications can be processed. For obvious reasons, vertical and horizontal isolation measures are required. That is to say, the kernel entity needs to be isolated from higher-level (user and PoSE) tasks and the tasks need to be isolated from each other. A distinction between user and supervisor mode of execution is demanded.

The second configuration of a non-specific node is to have only application tasks executing in user mode. PoSE, kernel, and nucleus then will be subjected to supervisor mode execution. In addition to that, operating system services are provided by passive objects. This leads to a traditional *monolith*, i.e., a procedure-based operating system in which all service functions are mapped into the same (supervisor mode) address space. Both, the RoI paradigm used to interact with PoSE and/or kernel and the NoI paradigm used to interact with the nucleus are of procedure-based nature anyway. It merely is a question of how these calling principles are carried out by an actual system representation.

The duality of process-oriented and procedure-oriented operating system structures [16] makes possible to present a design that can be implemented either way. However, having a process-oriented implementation (by means of active objects) in mind is the right approach in order to prevent possible performance bottlenecks. It is important to clearly identify system components that are possibly associated with an autonomous thread of control, protected by a separate address space, executed on a different node, and, therefore, accessed via RoI. On this basis, it becomes a matter of configuration to generate an actual system representation that really exhibits active objects as service providing entities. The other way round, namely assuming that all components will share the same address space, may end up into a highly inefficient implementation if decentralization needs to be achieved in a later development phase. This even may hold although, from the software technical point of view, either representation is possible.

4.1.4 Configuration Interplay

All configurations discussed above are able to coexist. The nucleus is used as common communication platform. Since PoSE and kernel services are provided by means of RoI, user-specific nodes are able to request services executed by either system or non-specific nodes. Vice versa, system-specific nodes are used to manage the parallel machine in terms of a *processor bank*, allocating user-specific nodes for the processing of parallel applications.

Whether or not user-specific nodes can be run by a combination of the configurations shown in Figure 6 (and explained above) actually depends on the application. For example, a load balancer (residing on a system server node) may decide to allocate as many tasks as possible in one-to-one correspondence with the nodes. If the number of tasks exceeds the number of available nodes, some and not all of the nodes need to be driven in multi-tasking mode of operation. This assumes that the number of tasks is already known at loading time. A configuration description (in the sense of a batch job specification) can be used to instruct the load balancer accordingly. That way, parallel applications are tried to be mapped onto a multi-node machine with the goal of keeping the message startup time for the majority of tasks as small as possible. Note, one-to-one correspondence between tasks and nodes also implies a single address space model and, thus, results in a low message startup time.

4.2 Parallel Computing Platform

The previous subsection discussed a system configuration that was classified by a monolithic representation of POSE, kernel, and nucleus. All three building blocks share the same address space and will be subjected to supervisor mode of execution. The other view, however, may also be sensible, namely to let the building blocks execute in user mode. In this configuration, a so-called "*guest level*" implementation of the native operating system on top of some other (host) operating system emerges. In fact, if the designer decides to construct a highly portable and configurable operating system, a guest level implementation is feasible in almost every case. It is mainly a question of how to isolate hardware-dependent system components—and this, at all, is the key question in order to achieve portability.

A guest level implementation of a parallel operating system is not only sensible but also mandatory at all. Consider the case in which a parallel machine is attached to some host computer on top of which a quite different operating system is likely to be executed. In that case, software packages need to be available for not only interconnecting the two different hardware components but also coping with the heterogeneity issues raised by two different operating systems. Three approaches for host interconnection can be distinguished. These approaches differ in the way a parallel application is going to be mapped onto the entire hardware complex. They also exhibit different techniques to physically interconnect both host and parallel computer.

4.2.1 Asymmetrical Coupling

A straightforward way to achieve the host coupling is to mainly provide *utility programs* that help users to utilize the parallel machine for their own purposes. These programs run under control of the host operating system. They are started by some sort of command interpreter. The entire complex defines a *hosted system* (Figure 7).

The parallel operating system then has to exhibit (1) a communication service that relies on protocols dictated by the host operating system and (2) service functions implementing the bridge to the host-resident utility programs. For example, in the UNIX case TCP/IP needs to be run by some node of the parallel machine. Typically, the coupling is implemented by a system-specific node. This node also may be assigned to execute software that enables the interaction with the utilities controlling the parallel machine. In that scenario, the host

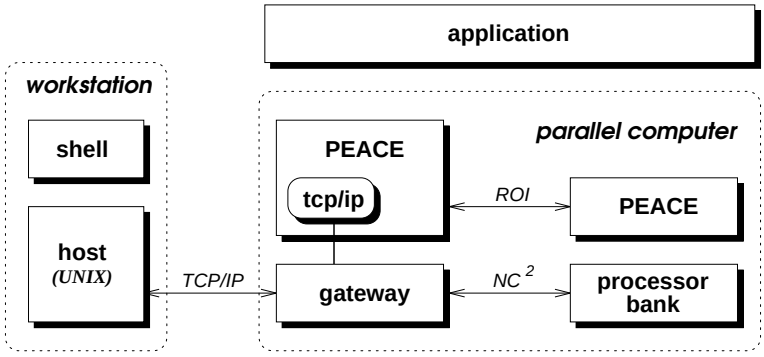

Figure 7: Hosted system

computer acts as *frontend* only. Application tasks are only executed by the multi-node machine.

Although straightforward and undoubtedly applicable, the pure frontend approach has its limits for quite a reasonable number of (distributed memory) parallel applications. Many of these applications distinguish between a *host task* and a *node task*. There will be one host task only, but a number of node tasks. The host task typically creates the node tasks, distributes initial parameters, and awaits final results that have been jointly computed by these subtasks. Node tasks actually represent the computational aspect of the parallel application, while the host task corresponds to the management aspect. This is also referred to as "*host/node programming*". In fact, the parallel program is to be distributed not only over the nodes of the parallel machine, but also over the host.

4.2.2 Logically Symmetrical Coupling

In order to overcome the limits given with an asymmetrical coupling between host and parallel computer, a host task must be supported by the same software platform used to process node tasks. This can be achieved by following up two ideas:

- make changes in the host operating system, i.e., add new kernel functions that mirror those provided on the nodes;
- let the kernel unchanged and provide a user-level software package that simulates the nodes of a parallel machine.

Following an approach that relies on host operating system kernel changes to achieve the interconnection extremely limits applicability and availability of the parallel machine. Even in the kernelized approach an efficient coupling is not guaranteed. For these reasons, the more promising way is to entirely rely on user-level software packages.

This is where the guest level implementation of a parallel operating system demonstrates its full power. In is not only a means by which parallel computing is made feasible by relying on local area network technology, for instance. Rather, this implementation defines

an *abstract parallel machine*, hiding from the application the different hardware subsystems. Basically, a *self-hosted system* emerges as shown in Figure 8.

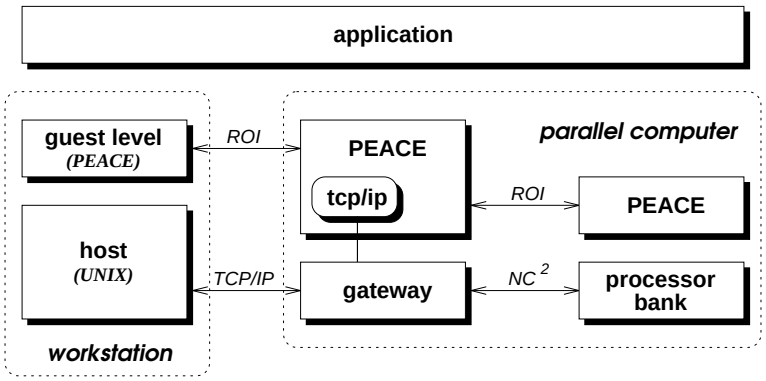

Figure 8: Self-hosted system

The tasks of the parallel application all see the same abstractions. The host task is able to cooperate with the node tasks on ROI basis. In particular, the host task can directly exploit the services provided by the parallel operating system. Vice versa, the parallel operating system is enabled to take advantage from the services provided by the host operating system. The host-residing part of the guest level implementation takes care for the mappings.

Nevertheless, the coupling between host and parallel machine is only logically symmetrical. This is because the interconnection still bases on two different networking schemes. The host is reachable via standard technology, such as Ethernet or FDDI and TCP/IP. The nodes of the parallel machine are networked by means of a problem-oriented communication system, e.g. the NC^2 suite. As consequence of these quite different networking techniques, a system-specific node as gateway still is indispensable. The gateway function then is to translate TCP/IP messages into NC^2 messages, and vice versa.

4.2.3 Physically Symmetrical Coupling

The gateway node of the parallel machine may easily become a serious bottleneck if large traffic is encountered. For example, this is the case when only the host provides special output devices in order to adequately display the results of a parallel computation. A typical device is the graphics display. Such a display usually is shared by a number of different applications and not necessarily been used by parallel applications only. Its proximity to the host, thus, is quite reasonable.

Overcoming the potential I/O bottleneck defined by the gateway node, however, has far-reaching consequences. First, additional hardware support is required, namely to directly attach the host computer to the network that interconnects the nodes of the parallel machine. Second, standard communication protocols as used by the host operating system for LAN environments must be sacrificed.

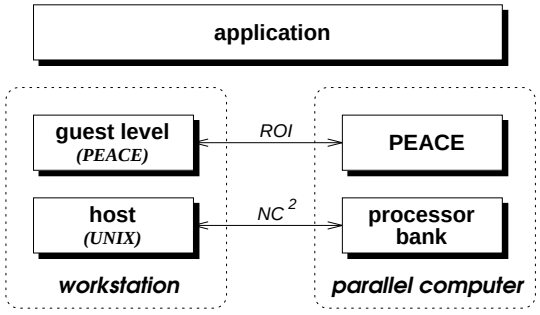

Figure 9: Self-hosted integrated system

This approach leads to a fully *integrated system* (Figure 9). It does not require to re-implement (or even port) parts of the parallel operating system, excepted the COSY protocols. The guest level implementation still may be a user-level software package. However, instead of accessing TCP/IP sockets, e.g., raw network device access is required. The guest level then must have been extended by the same COSY implementation that is actually been used by the native nuclei.

5 Conclusion

A massively parallel computer, by its nature, is a *tightly-coupled distributed system*. Parallel applications being run by this machine, thus, are specific types of distributed applications. These applications demand system-wide message passing with very low latency and very high efficiency. It is this performance issue which, from the operating system design point of view, is one of the most significant characteristics to distinguish a parallel application from a distributed application. Distributed operating systems, even those basing on a microkernel, are by far too overhead-prone. Rather, specifically designed parallel operating systems are required, which also show for distributed operating systems characteristics.

The major goal in parallel operating system design, therefore, is to combine transparency with efficiency. *Transparency* means to hide from parallel applications typical problems coming up with distributed systems—but without enforcing transparency in all cases. This requires to provide almost every system service a distributed operating system provides—but not necessarily providing these services at all times. *Efficiency* means to reduce the message startup time for a given application to an absolute minimum. Message passing is a fundamental system capability to support inter-node communication in massively parallel systems and, hence, must be provided by a proper platform—but without all applications being obliged to use the same platform. Thus, system services are to be provided on *extant-on-use* basis.

State of the art parallel operating systems design must obey the maxim not to punish applications by unneeded system functions which are unexploited but still "used"[25]. This includes the entire spectrum of computer resources, ranging from memory space to processor cycles. An open, application-oriented operating system structure is required, with the

application and not the operating system deciding which functions must be supported and which must not. An approach should be followed in which an operating system is being understood as a *family of program modules* and not as a monolith of more or less related components. In such a context, the parallel application is an integral part of a *family of parallel operating systems* and *object orientation* then is the natural choice to design and develop such a family. Thus, an application becomes the final system extension.

The paper presented rationale and concepts of the design of parallel operating systems for distributed memory MIMD machines. PEACE was used as a case study system to exemplify the design concepts. The PEACE system is running as UNIX guest level on a cluster of workstations and as native operating system on a 16 node (32 processor, i860) MANNA system [12], 64 node (T800) Transputer system, and 320 node (mc68020) SUPRENUM system [11]. It provides a common, object-oriented and scalable "software backplane" for parallel computing [17].

Acknowledgement

I would like to thank Joachim Beer for his helpful comments on the manuscript of this report.

References

[1] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

[2] R. Bryant, Hung-Yang Chang, and B. Rosenburg. Experience Developing the RP3 Operating System. *Computing Systems, The Journal of USENIX Association*, 4(3):183–216, 1991.

[3] D. D. Clark. The Structuring of Systems Using Upcalls. *Operating Systems Review*, 19(5):171–180, 1985.

[4] J. Cordsen and W. Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (I-WOOOS '91)*, pages 24–28, Palo Alto, CA, October 17–18, 1991. IEEE 91TH392-1.

[5] Thinking Machines Corp. The Connection Machine CM–5 Technical Summary. System Referance Manual, October 1991.

[6] H. Custer. *Inside WINDOWS-NT.* Microsoft Press, ISBN 1-55615-481-X, 1993.

[7] R. S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[8] Feasibility Study Committee of the Real-World Computing Program. The Master Plan for the Real-World Computing Program, March 2, 1992. Draft.

[9] D. Ferrari. Real–Time Communication in an Internetwork. Technical Report TR-92-001, International Computer Science Institute, Berkeley, CA, 1992.

[10] M. Gien. Micro-kernel Architecture – Key to Modern Operating System Design. Technical Report CS/TR-90-42.1, Chorus systèmes, Paris, 1990.

[11] W. K. Giloi. The SUPRENUM Architecture. In *Proceedings of CONPAR 88*, pages 10–17, Manchester, UK, September 12–16, 1988. Cambridge University Press.

[12] W. K. Giloi and U. Brüning. Architectural Trends in Parallel Supercomputers. In *Proceedings of the Second NEC International Symposium on Systems and Computer Architectures*, Tokyo, August 1991. Nippon Electric Corp. appears as book.

[13] 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.

[14] A. J. Herbert and J. Monk. *ANSA Reference Manual.* Advanced Network Systems Architecture, Cambridge, UK, 1987.

[15] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *Transactions of Software Engineering*, SE–11(4), 1985.

[16] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, April 1979. Reprint of Proceedings of the 2nd International Symposium on Operating Systems Structures, IRIA, October, 1978.

[17] J. Lennon. Give PEACE a Chance. The Plastic Ono Band – Live PEACE in Toronto, Apple Records, 1969.

[18] H. M. Levy. Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, Oct. 13–16, 91, Asilomar Conference Center, Pacific Grove, CA, USA. ACM Operating Systems Review, Vol. 25, No. 2, Special Issue, 1991.

[19] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, 1986.

[20] K. Li and R. Schaefer. A Hypercube Shared Virtual Memory System. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 1, pages 125–132, 1989.

[21] Meiko Ltd., Bristol, UK. *CS/Tools Programmer's Manual*, 1990.

[22] H. Mierendorff. Bounds on the Startup Time for the GENESIS Node. Technical report, GMD F2.G1, Bonn, Germany, 1989. ESPRIT Project No. 2447.

[23] S. J. Mullender and A. S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4):289–299, 1986.

[24] J. Nolte. Language Level Support for Remote Object Invocation. Arbeitspapiere der GMD 654, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany, June 1992.

[25] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering*, SE–5(2), 1979.

[26] Parsytec Computer GmbH, Aachen, Germany. *PARIX Programmer's Manual*, 1991.

[27] J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley Publishing Company, ISBN 0-201-06079-5, second edition, 1985.

[28] W. Schröder-Preikschat. Overcoming the Startup Time Problem in Distributed Memory Architectures. In Veljko Milutiniovic and Bruce D. Shriver, editors, *Architecture and Emerging Technologies Tracks*, volume 1 of *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, pages 551–559, Kauai, Hawaii, January 8–11, 1991. IEEE Society Press, IEEE 91TH0350-9.

[29] W. Schröder-Preikschat. *P_EACE—The Logical Design of Parallel Operating Systems*. 1993. In preparation.

[30] Telmat, France. *UBIK Programmer's Manual*, 1990.

[31] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. Technical Report UCB/CSD 92/675, University of California, Berkeley, CA, 1992.

[32] P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986.

[33] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, volume 21 of *Operating Systems Review*, pages 63–76, Austin, Texas, CA, 1987. ACM.

[34] R. Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the USENIX Winter Technical Conference*, pages 449–468, San Diego, CA, USA, January 25-29, 1993.