Making Massively Parallel Systems Fast*

J. Nolte, J. Heuer, M. Sander,
F. Schön, W. Schröder-Preikschat

GMD FIRST
Hardenbergplatz 2, 1000 Berlin 12, FRG

ABSTRACT

Massively parallel systems are distributed memory architectures of a very large number of autonomously operating, interconnected nodes. The nodes cooperate through message passing via a high-speed interconnection network. The critical issue in such systems is communication latency. A major component of communication latency is the message startup time, i.e., the time it takes to execute the appropriate operating system kernel function for message passing. One of the necessary system optimizations therefore concerns the startup time minimization. Our approach to this problem is to provide different versions of the kernel for different modes of operation of the system. All versions have the same interface to the other modules of the distributed operating system and, thus, are interchangeable. This is accomplished by representing the message-passing kernel as an abstract data type.

1. Introduction

Massively parallel systems are distributed memory architectures consisting of a very large number of autonomously operating, interconnected nodes. The nodes cooperate through message passing via a high-speed interconnection network. Consequently, the operating system of such a machine is a distributed operating system [Tanenbaum, van Renesse 1985], [Schroeder, Gien 1989]. One of the features of a distributed operating system is *distribution transparency*, needed to reference objects (system entities as well as user tasks) "in a consistent manner regardless of such factors as access, location, migration and replication" [Herbert, Monk 1987].

The most important requirement for such a system is *efficiency*. The system may offer all the desired benefits such as resource sharing, parallelism, scalability, and fault tolerance, and yet it will not be acceptable if it fails on the side of performance.

One of the most crucial performance parameters is the message startup time. Future massively parallel systems must be capable of executing a message startup sequence – preparing, issuing, accepting and queuing of data transfer requests, along with interrupt handling, context switching and process scheduling – in the order of magnitude of 10 µsec [Mierendorff 1989].

* This work was supported by the European Commission under the ESPRIT-2 program and carried out as part of the GENESIS project, grant no. P2702

This paper deals with design aspects of message-passing kernels for high-performance, massively parallel distributed memory architectures. It presents first experiences made with the PEACE distributed operating system [Schroeder 1988]. PEACE extended well-established operating systems principles [Habermann et al. 1976] into the area of highly parallel distributed memory systems. The notion of program families [Parnas 1979] provided the framework for a system that caters to a variety of user requirements. Members of the PEACE operating system family consist of dedicated server processes that provide application-oriented services.

Basically, the PEACE approach is to use a small and efficient message passing kernel as foundation for all higher level system activities. In that respect PEACE is comparable to V [Cheriton 1984] and Amoeba [Mullender, Tanenbaum 1986], for example. However, it differs from the systems mentioned by a design that meets the needs of massively parallel systems. Highest consideration is given to the efficient implementation of the family of *PEACE nuclei*. All members have the same communication interface but different functionality and performance. The proper nucleus is then selected according to the application requirements.

The need for a message-passing nucleus family is evidenced by a case study that shows the potential disproportion between multi-tasking and communication. First, the case study hardware environment is briefly presented. Following a short PEACE performance analysis, the nucleus family is discussed. After that, the internal nucleus structure that conforms to the family approach is described. Finally it is explained how the PEACE nucleus optimally supports the GENESIS node architecture [Giloi 1990], achieving a symbiosis between hardware and software.

2. Case Study Environment

PEACE has been developed for SUPRENUM [Giloi 1988], a large scale MIMD supercomputer for numerical applications. A brief overview of the SUPRENUM architecture clarifies the hardware platform used for the PEACE kernel analysis. This is followed by a short description of major PEACE concepts and components referred to in the paper.

2.1. SUPRENUM

One of the unique features of SUPRENUM is its two-level structure consisting of *clusters* and *nodes*. A cluster consists of 16 *processing nodes* (PNs) and up to 4 *service nodes* (SNs), interconnected by a parallel bus system with a transmission rate of 320 Mbytes per second. The clusters form the lower level of the hierarchy. At the higher level, up to 16 clusters are interconnected by a torus (4-cube) structure. Figure 1 illustrates this architecture.

Designed primarily for double-precision, floating-point array processing, the PN features in addition to the Motorola MC68020-based CPU a vector processor with a peak performance of 20 MFLOPS for two chained operations and the Motorola MC68882 floating point unit. A well-matched memory structure, consisting of 8 Mbytes of DRAM (controlled by the Motorola paged memory management unit MC68851) and two 64 Kbyte vector registers, results in a high sustained vector performance for sufficiently long vectors. Reflecting the state of technology in 1986, the scalar performance, however, is rather weak. The MC68020 has little over

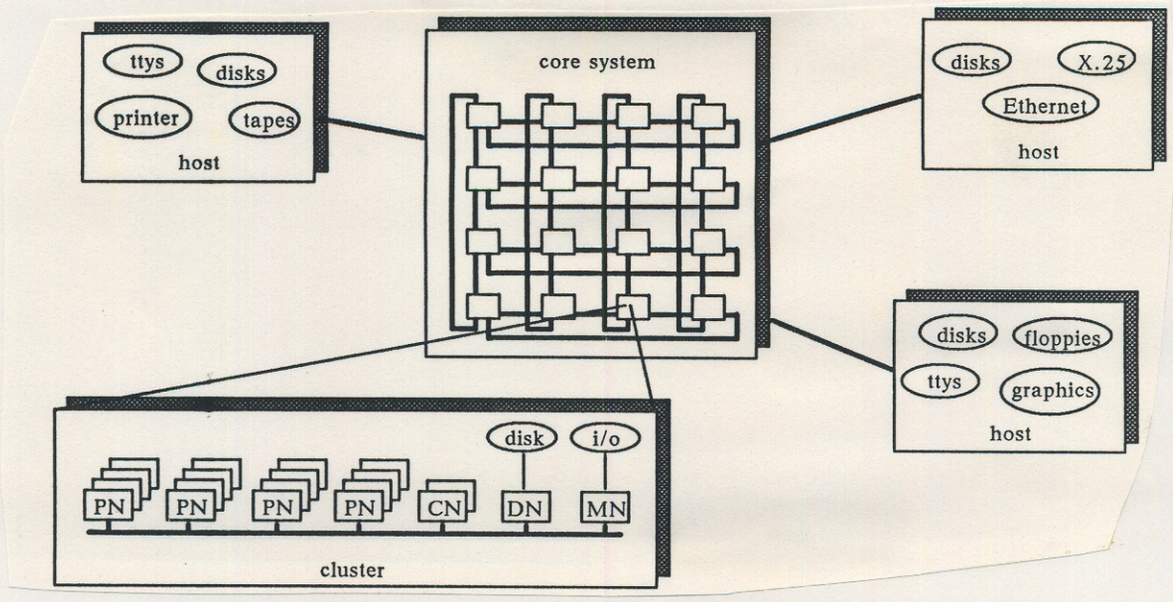

Figure 1: The SUPRENUM Architecture

2 MIPS. Random access to main memory has 3 wait states on read and 4 wait states on write. While this does not affect much the array processing performance of the node (in this case, the main memory works in the static column mode), it leads to relatively long execution times of sequential code (e.g., operating system routines).

Each cluster comprises two *communication nodes* (CNs) for interconnection with other clusters. Duplicating the CN in a cluster doubles the transmission bandwidth and makes the interconnection fault tolerant. Each CN has two physical links, a link being a 125 Mbits per second bit-serial token ring connection. The physical peak transfer rate between a pair communicating nodes is 40 Mbytes per second. Inter-cluster communication via the parallel bus system and intra-cluster communication via the links use the same fault tolerant protocol that is similar to the worm whole routing protocol. The fact that the inter-cluster interconnection has a lesser bandwidth than intra-cluster interconnection favors to some extend cluster locality of the application algorithms.

2.2. PEACE

PEACE has been designed to manage the SUPRENUM core system. The PEACE process model distinguishes between lightweight processes (*threads*) and heavyweight processes (*tasks*). Multiple threads of control that share the same address space and scheduling domain are called *teams*. Teams are used to implement user tasks and system services.

As indicated in figure 2, PEACE consists of three major building blocks. A more detailed system description can be found in [Berg et al. 1990].

Most of PEACE is executed in non-privileged user mode and defines an application-oriented *distributed operating system*. This component is constituted by a multitude of server teams that are site-independent and, thus, may be arbitrarily distributed over the parallel machine. The collection of teams provides typical services like naming, file handling, process management, memory management, exception

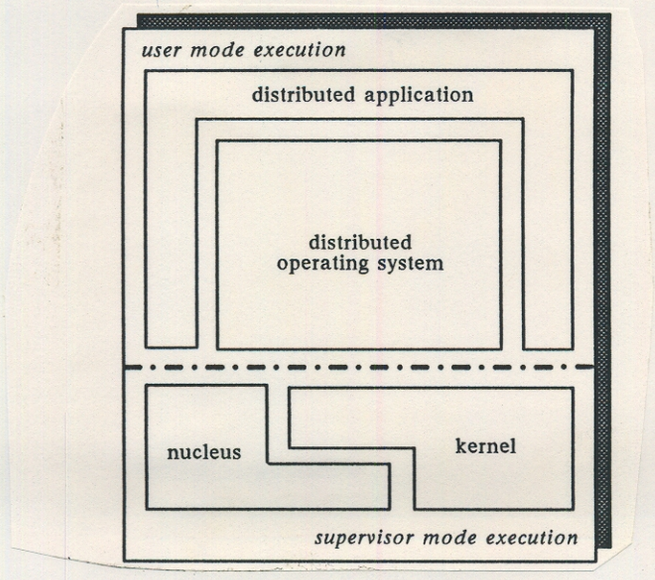

Figure 2: The PEACE Architecture

handling, inter-networking, loading, etc.

The distributed operating system is based on a *kernel* that implements an *abstract PEACE machine*. The kernel encapsulates all hardware-dependent functionalities. Thus, it contains device drivers, the MMU driver, and the trap and interrupt handler. In addition, the kernel provides services such as creating processes dynamically and associate them with system-wide unique identifiers and address spaces. Because of its proximity to the hardware, the kernel executes in supervisor mode. Internally, the kernel is represented as a multi-threaded team.

Services provided by the kernel and the distributed operating system are invoked by remote procedure calls [Nelson 1982]. This supports a system configuration that manages the parallel machine as a processor bank, thus associating individual nodes exclusively to user tasks.

The most basic component is the *nucleus*, whose sole function is to enable system-wide communication between tasks (teams or threads). Services of the nucleus are also invoked by remote procedure calls, which are handled in the usual way of system calls, namely as a local trap. The nucleus is also directly used by communicating tasks of the distributed application.

3. Rationale for the Family Concept

The primary requirement of massively parallel systems is very fast, system-wide inter-process communication. As pointed out in [Lantz et al. 1985], communication performance is less a problem of limited network bandwidth. Rather, it is a problem of system software overhead caused by node-bounded activities such as interrupt handling, buffering, context switching, scheduling and so on. A detailed analysis of typical message-passing kernels such as PEACE reveals that these activities are not necessarily required to support communication in a distributed environment. Rather, they are only required if one extends traditional multi-tasking into the distributed environment.

In addition to improving processor utilization in general, multi-tasking is a functionality that introduces scaling transparency into parallel, distributed applications. However, in cases where the application scales well with the actual number of nodes, it introduces nothing else but overhead and, thus, slows down overall application performance. Therefore, the main problem in the design and development of operating systems for massively parallel systems is to avoid that "artificial" bottleneck, without sacrificing multi-tasking altogether. Whether multi-tasking is really needed depends on the application and on the actual number of nodes available for execution. As a general guideline, applications should be structured so that there is exactly one task per node. This constitutes the majority of parallel (numerical) programs.

3.1. PEACE Nucleus Benchmarking

A multi-tasking PEACE nucleus has been benchmarked in a scenario where each node executes only a single task. Two versions of the nucleus were tested: one with non-protected and one with protected address spaces. All benchmarks were executed 100 000 times; from this the time for a single benchmark operation was computed.

Internal functions of the multi-tasking nucleus have been isolated and individually benchmarked to determine pure multi-tasking overhead. These functions will not be present in a single-tasking nucleus. They typically deal with process scheduling and dispatching and interrupt handling.

A SUPRENUM hardware platform consisting of two PNs has been employed for benchmarking. Both PNs resided in the same cluster. Note that multi-tasking implies node-bounded activities. This makes it irrelevant whether to use PNs from the same or from different clusters. In either case the same physical network interface, the same protocol and, hence, the same low-level communication software is used in SUPRENUM.

3.1.1. Rendezvous Time Parameters

Usually, a PEACE rendezvous means the exchange of a *send–receive–reply* sequence between a client and a server. In order to support remote procedure calls most efficiently, the PEACE nucleus provides an alternative interface for servers. This interface combines the corresponding *receive* and *reply* into a single operation *replace*, meaning to replace a rendezvous (remote procedure call) with another one. The mechanization first issues a *reply* and immediately afterwards performs a *receive*. Table 1 lists the timing for PEACE rendezvous based on *send–replace* sequences.

configuration	time (µ sec.)			
	local		*remote (with MMU)*	
	without MMU	*with MMU*	*phy. DMA*	*log. DMA*
1–to–1	360	420	784	867
2–to–1	312	361	537	566
4–to–1	284	331	525	554
8–to–1	270	315	525	554
16–to–1	262	305	525	562

Table 1: Rendezvous Time Parameters

Since the inter-process communication of PEACE is based on transferring fixed-size message packets of 64 bytes, 128 bytes are exchanged with each rendezvous. These

packets are buffered by the nucleus, resulting in two copy operations both at the client and the server site (a copy in/out with each *send* and *replace*).

The transfer of arbitrary sized messages is supported by a separate mechanism called *high-volume data transfer*. This mechanism allows for the system-wide exchange of memory segments without intermediate buffering. It is supported by "intelligent" DMA hardware in each node. Once having established the rendezvous, high-volume data transfer takes place. The critical factor for a user-level message startup time therefore consists in the rendezvous.

For the case of the local rendezvous, the performance figures of a nucleus with and without address space protection are shown. Address space protection is by means of a memory management unit (MMU). The different timing parameters show the MMU overhead in the process of copying data (arguments and message packets) between nucleus and user space. The same overhead exists in the case of remote rendezvous. Note that no task context switches took place, for only a single task was executed on a node.

For the case of the remote rendezvous, a SUPRENUM-specific aspect could be evaluated, namely the performance difference between logical and physical DMA during network-wide message transfers. Logical DMA means to perform MMU address translation with each data item (a 64 bit unit) being sent or received. In physical DMA mode, MMU address translation is usually done only once for the entire message packet before it will be sent or received. This hardware feature implies no additional software overhead, rather it must be either enabled or disabled at bootstrap time.

A *many–to–one* relation between client and server has been investigated. Up to 16 client threads issued a rendezvous request to a single server. Hence, the amount of overhead caused by server scheduling/dispatching and interrupt handling at the server site was revealed. Situations with more than one client were represented by a multi-threaded client team, whereas the server team always was single-threaded. In the case of having $n>1$ client threads requesting simultaneously a rendezvous, the resulting benchmark performance was divided by n. Hence, timing measurements for each configuration have been normalized to a single rendezvous.

3.1.2. Basic Time Consumed

For benchmarking multi-tasking overhead, dedicated nucleus instances were generated. The nucleus call interface was excluded. Table 2 shows the results.

Nucleus call overhead simply is the time it takes to request and deliver the process identification of the calling thread. It encompasses trap handling, monitor locking/unlocking, and argument passing between user space and supervisor (nucleus) space.

Segment validation is performed only as part of high-volume data transfers. This means to verify the validity of a given logical address with respect to the team address space that is involved in the data transfer. SUPRENUM housekeeping pertains to the locking and unlocking of the vector floating point unit in the case of task switching.

action		time (µ sec.)	
		with MMU	*without MMU*
nucleus call		21	19
message buffering		49	43
segment validation		255	0
SUPRENUM housekeeping		32	32
interrupt latency		88	86
team scheduling		68	68
dispatching	*thread*	38	38
	team	56	56
switching	*thread*	21	21
	team	40	25

Table 2: Basic Time Consumed

Team scheduling only takes place if several teams (tasks) need be executed by the node. Preemptive and timer-driven scheduling is performed. The time listed encompasses team switching, i.e. a complete context (address space) switch. Thread switching is concerned only with swapping CPU register sets. The difference between the times for team scheduling and team switching indicates the raw team scheduler overhead. Thread and team dispatching gives the raw overhead needed for ready list manipulation. Two operations are involved in a single dispatching activity, viz. setting a thread/team ready to run and selecting the next thread/team for execution.

Interrupt latency is the time needed to propagate hardware interrupts to threads. It encompasses the saving and restoring of CPU registers, interrupt handler invocation and return, and synchronization. The complete sequence is executed for remote thread communication.

3.1.3. Analysis

A remote PEACE rendezvous between two tasks based on *send-replace* sequences implies at least two nucleus calls, four message buffering activities, two team dispatchings, and two SUPRENUM housekeeping activities. In this model, team switching, i.e., complete task context switches, are omitted because only a single task was executed per node. If a task required to block, it enters the nucleus *idle loop* and starts polling on external events such as incoming messages.

In a *1-to-1* rendezvous the interrupt latency caused by the receive of (send and reply) messages is encountered twice. Thus, a total of 502 µsec with MMU and 472 µsec without MMU has been encounted for the individual rendezvous. Considering the entire rendezvous time of 784 µsec (867 µsec), approximately 64 (54) percent of it are caused by node-bounded activities. It should be noted that this portion means pure overhead for applications which have only a single task executed by the multi-tasking PEACE nucleus. None of the basic parameters considered above will be relevant for a single-tasking nucleus.

In a *2-to-1* rendezvous, a performance gain of 247 µsec (301 µsec) has been obtained for the physical (logical) DMA mode of operation. With higher load at the server site, i.e., with an increasing number of clients that are sending to the same server, a speed-up of the individual rendezvous was observed. In this scenario, the server always is busy because of pending client messages, i.e., its message queue will never be empty. Therefore, the server does not block on *replace*, which would

happen in the case of an empty message queue. This means that neither scheduling/dispatching nor SUPRENUM housekeeping activities must be performed, and that the server never enters the idle loop. Similar considerations hold for the case of local rendezvous.

3.2. Lessons Learned

The analysis confirms our conjecture [Schroeder 1987] that an increase of network bandwidth and interface hardware performance alone is not the answer to the performance bottleneck problem. It shows also, that the need for faster hosts [Lantz et al. 1985] is not the ultima ratio either. Rather, devising more efficient software solutions is as important as the speed-up of communication hardware.

The PEACE performance figures discussed above are very impressive, especially when comparing them with Amoeba [van Renesse et al. 1988], which also protects address spaces. Nevertheless, the performance is not good enough for massively parallel systems. Multi-tasking functionality creates an unnecessary performance bottleneck in all cases where applications do not require it.

In the scope of parallel computing there often are applications which scale well with the actual number of nodes and, thus, make multi-tasking superfluous. On the other hand, for reasons of fault tolerance and multi-user support, multi-tasking should not be totally sacrificed either. Rather, a solution must be found that lets the application pay only for those functions that are really required.

3.2.1. The Family Approach

Basically, the program family concept [Parnas 1979] means to identify a *minimal subset* as well as *minimal extensions* of system functions during the software design process. The minimal subset constitutes a common abstraction to all higher-level software modules. Ideally, it encapsulates solely *mechanisms* and not the *policies*. Policies should be introduced as minimal extensions. This leads to the step-wise functional enrichment of the system. Remarkably, this *bottom–up* construction is controlled in a *top–down* fashion: lower-level modules are introduced only when required by higher-level modules. Applying this approach recursively leads to a hierarchical system organization. The *uses relation* [Parnas 1979] between the modules then is an important instrument for associating modules to levels in the hierarchical system. Consequently, this leads to a pure application-oriented system structure, in which only those system functions (i.e., modules) are defined that are required by a given application.

Originally, this concept has been devised to improve maintainability of very large and complex software systems, e.g., operating systems. A program family design naturally results in a highly modular system structure, with each module being an *abstract data type* [Liskov, Zilles 1974]. It helps to concentrate on the pertinent facts during system development while providing a flexible framework for incremental construction, selective adaptation, and future extension of any kind of software.

3.2.2. Consequences for PEACE

The family concept is used in PEACE to realize an application-oriented distributed operating system. Teams represent abstract data types which implement system services. They are loaded on demand, when the application needing these services is loaded, more specifically, when the services are invoked for the first time.

The application-oriented family concept is to be applied to the nucleus design, in order to provide a well-defined framework for different instances of the nucleus. Presently, the nucleus family of PEACE consists only of the two nucleus instances discussed so far. This is because scaling transparency was considered the governing user requirement in SUPRENUM. As demonstrated by the case study, however, a more sophisticated family organization is required. There is a need for members ranging from a single-threaded communication library to a complete multi-tasking and multi-user implementation.

In this sense, the PEACE nucleus then provides the minimal subset of system functions, whereas kernel and distributed operating system provide minimal extensions. The difference between kernel and distributed operating system is that the kernel adds mostly mechanisms, whereas the distributed operating system introduces policies. The purpose of having a nucleus family is to have the minimal subset of system functions represented in an optimal fashion.

4. The Nucleus Family

For the design of a nucleus family for massively parallel systems, the distinction between single-tasking and multi-tasking support is too rigid. The PEACE development for SUPRENUM demonstrates that a more discriminating approach must be taken. One reason is that a large class of applications must be optimally supported. A second reason is that it is much less risky to develop an operating system incrementally rather than trying to implement the entire system in one shot.

In this sense, a multi-tasking nucleus will begin with supporting static scheduling, to be extended by preemptive scheduling and address space isolation, by memory protection and, finally, by multi-user security. The key aspect is that the nucleus design is required to be complete before implementation starts. All the intermediate steps towards the final implementation are considered of representing an autonomous member of the nucleus family in its own right, exhibiting different functionality and performance characteristics. Figure 3 illustrates this approach by means of a nucleus family tree.

As is discussed below, nucleus functionality increases from left to right and from top to bottom. By the same amount by which functionality is increased, the communication performance drops, i.e. the message startup time increases.

4.1. The Family Tree

At the root of the family tree, there are two major user requirements, namely single-tasking and multi-tasking. In case of single-tasking, a distinction is made into single-threading and multi-threading. Note, a task is a PEACE team and hence may be multi-threaded. In case that only a single thread of control must be supported on a node, the nucleus purely implements *communication*. This is the situation where a

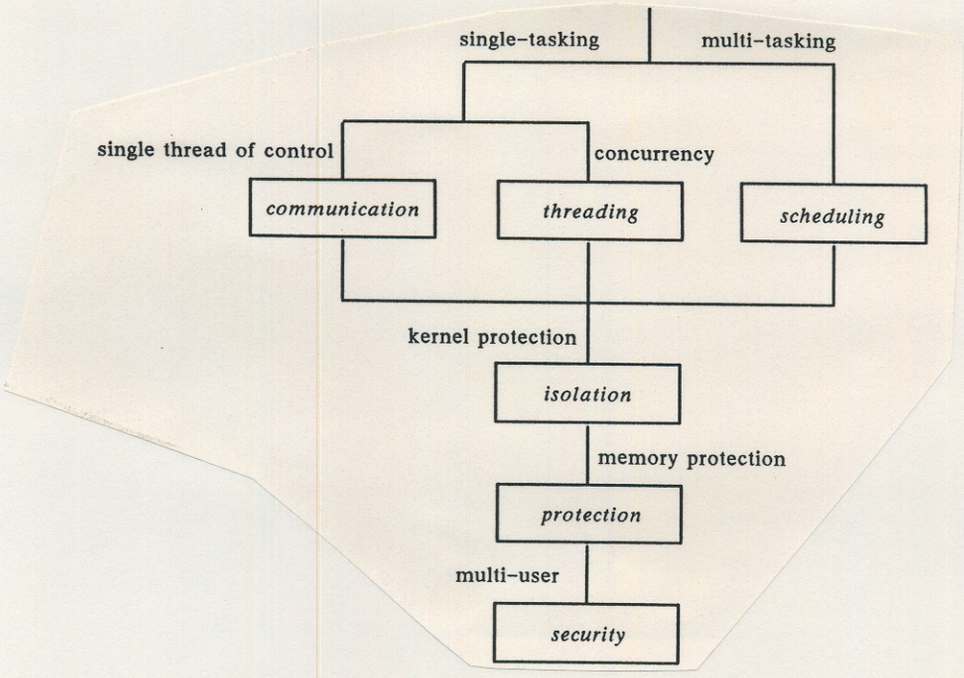

Figure 3: PEACE Nucleus Family Tree

one-to-one mapping between a node and a user task is feasible.

In PEACE, synchronous inter-process communication is considered as the fundamental communication paradigm, offering the highest possible end-to-end communication performance. *Threading* introduces concurrency relative to a common address space, Multiple threads of control constitute a team, specifying a one-level scheduling domain.

The need for multi-tasking at this stage increase only scalability and does not imply protected address spaces. Concurrency relative to several teams belonging to the same user application is supported. Thus, a non-preemptive *scheduling* strategy is introduced into the nucleus. This constitutes a switch from one-level to two-level scheduling, i.e. it extends thread scheduling by team scheduling.

In all these cases, the nucleus takes the form of a non-protected shared library that is directly linked to the application task. In addition, a node controlled by these nucleus instances is used exclusively for user task execution. Except for the nucleus, no other PEACE system components are located on these nodes. Application tasks (teams) are loaded as part of the node bootstrap procedure. Following the pattern of coroutines, merely dynamic creation/destruction of threads is supported. Since system services are invoked by means of remote procedure calls, PEACE server teams may reside elsewhere and are not required to share a single node with the user teams (i.e. tasks).

The requirement for preemptive scheduling implies a complete nucleus *isolation*. Now, interrupts must be handled, and the teams are given a limited time quantum for program execution. Dynamic creation/destruction of teams is introduced. However, team address spaces are not yet isolated, i.e., protected. In addition, the nucleus must be invoked through traps, thus introducing more overhead. Consequently, the nucleus

as part of the kernel is executed in privileged supervisor mode and protected against user mode tasks. Nucleus (kernel) protection may also be a case for the single-threading and multi-threading instances.

Applying address space isolation to user teams completes *protection* in its traditional sense. The kernel is extended by services to associate process objects with address spaces. Excepted of having explicitly requested segment sharing, teams are prohibited to directly access and manipulate peer address spaces. This makes the nucleus a true multi-tasking system, able to support the execution of multiple teams belonging to different applications. In this nucleus instance, high-volume data transfer, for example, implies explicit segment validation before accessing a team address space.

If a multi-user mode of execution is to be supported, memory protection alone is not sufficient for massively parallel systems. Rather, communication firewalls must be established such that threads belonging to different user teams are prohibited to communicate with each other. Thus, the principle of memory protection is extended into a distributed environment. Precautions for communication *security* are necessary. The nucleus is required to enforce integrity of the different distributed user applications running concurrently on the parallel machine.

4.2. Problem Orientation

The order in which we discussed the members of the nucleus family also defines the order of increasing system functionality. In the same order user-level communication performance drops. Applications which scale well with the given number of nodes are supported by either of the first two nucleus instances (communication and threading), depending on whether synchronous or asynchronous inter-process communication is required. A first step towards multi-tasking is to introduce the scheduling nucleus. This nucleus represents a compromise between scalability and protection. It is used for dedicated and mature applications.

A similar compromise can be made concerning nucleus isolation. However, there are two additional major issues to be addressed by the nucleus. First, dynamic nucleus reconfiguration must be feasible. Second, multi-tasking must be supported regardless of the availability of a MMU. This generally enlarges the applicability of the system.

Dynamic nucleus reconfiguration is needed in cases where system availability is the dominating issue. A single-tasking nucleus is used in all cases where tasks are mapped one-to-one to nodes. Node crashes, however, may require task redistribution and, thus, could cause the switching to a multi-user multi-tasking nucleus at some other node. This also may imply that some nodes are now to be shared between a number of tasks belonging to different user applications. In this case, the single-tasking nucleus need be replaced by the multi-tasking nucleus at run time.

5. Internal Nucleus Structure

Common to all nucleus instances discussed above is the communication aspect. Additional minimal subset of communication system functions. The actual structure of a nucleus instance reflects this aspect. How the internal organization of the PEACE nucleus looks like is illustrated in figure 4. This organization also supports a novel hardware architecture for massively parallel systems which is described in the

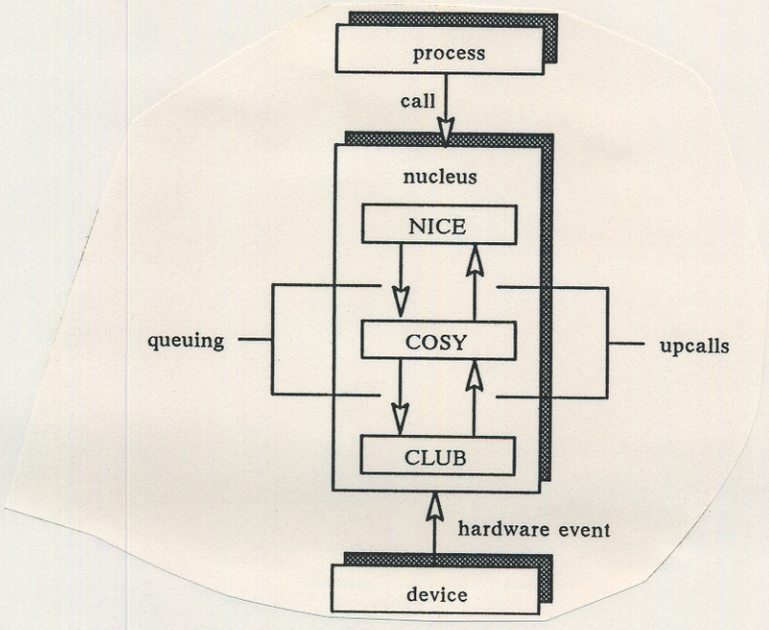

Figure 4: Nucleus Organization

following section.

At the top, we have a **n**etwork **i**ndependent **c**ommunication **e**xecutive, called *NICE*. This module implements system-wide synchronous inter-process communication, while asynchronous communication is supported by means of threads. The module encapsulates the different functions of the nucleus and manages rendezvous, threads, teams, and so on. That is, the nucleus family actually is a NICE family, and all members of this family inherit the same NICE interface. The two layers below NICE form the communication system, whose purpose is to exchange data streams between nodes in the most efficient manner. Rather than addressing processes (i.e. threads and teams), address space segments serve as origin and target for the message transfer.

The major task of the **co**mmunication **sy**stem (*COSY*) is to execute the proper communication protocol that copes with the problems given by the underlying network system. For example, depending on the type of network it performs segmenting/blocking of messages and introduces the view of virtual channels. The interface between NICE and COSY is asynchronous. In the downward direction it uses request queues, whereas the upward direction is implemented by *upcalls* [Clark 1985].

In order to keep most of the nucleus portable, a *CLUB* layer introduces an abstraction of the physical network interfaces. The CLUB layer encapsulates the low-level device drivers that control the network interface. It maps logical node addresses associated with threads onto physical node addresses as required by the hardware. Generally, it combines a group of related nodes into a club of homogeneous network hardware interfaces.

Again, the interface between COSY and CLUB is asynchronous, implemented by a request queue in the downward direction and upcalls in the upward direction. The CLUB hardware interface is assumed to be interrupt driven. Upcalls (to COSY and NICE) may be directly triggered by hardware interrupts.

With respect to different user requirements, different NICE modules are used for optimal support. Different network characteristics are covered by dedicated COSY modules, improving the portability of NICE by means of isolation from network details. Network hardware transparency is achieved by CLUB, i.e., there are different CLUB implementations for different network hardware interfaces. This generally improves portability by means of isolating nucleus instances from the details of network devices.

6. Symbiosis between Hardware and Software

So far, a pure software solution to the communication performance bottleneck problem of massively parallel systems has been discussed. Further improvements may come from dedicated support hardware. A stand-alone hardware solution is not feasible: rather a symbiosis between software and hardware solutions is required. This symbiosis will exist in GENESIS [Giloi 1990] by combining the PEACE nucleus family approach and a novel node hardware architecture into a powerful multicomputer system.

In the course of discussing the software/hardware symbiosis we consider a node that consists of at least of two CPUs that communicate by means of shared memory. The upper limit of the number of CPUs that makes technically sense for being plugged onto a node is a function of physical parameters such as CPU speed, memory speed and total memory bandwidth. There are no software constraints.

A GENESIS node will be configured with up to 4 CPUs. The node architecture is symmetrical, i.e., all CPUs have equal access to the physical network interface. Depending on the distribution of nucleus modules over the CPUs, different node configurations with different modes of operation are provided. The software configurations specify the node organization of being either functional dedicated or symmetrical.

6.1. Functional Dedicated Organization

A functional dedicated node configuration employs one CPU as *application processor* (AP) and a second CPU as *communication processer* (CP). The AP executes application tasks, while the CP is responsible for global inter-task communication.

With a GENESIS node consisting of 4 CPUs we will, for example, have two *logical nodes* on a single physical node. Each logical node consists of two CPUs, AP and CP. Both CPs then share the same physical network interface. Another scenario could be to configure the physical node with three APs and one CP.

In the following, we identify three different node representations, distinguishing between single-threaded and multi-threaded task execution. Two node configurations are provided in the case of multi-threaded tasks. In both configurations each thread is executed by a separate CPU, which results in the parallel execution of a task (team).

6.1.1. Functional Distributed Nucleus

The basic idea is to have COSY and CLUB completely executed by a dedicated processor, the CP. The processing of NICE is shared by both processors. This is based on the fact, that NICE knows about tasks (in the form of teams) and, therefore, is

responsible for exchanging the AP register set on performing a task context switch. The manipulation of ready lists for scheduling purposes, however, is done jointly by both the AP and the CP. In the case of incoming messages, for example, the CP directly dispatches the target thread by placing it onto the AP ready list. This way, interrupt overhead at the AP site caused by message-passing is avoided. Figure 5 shows this representation.

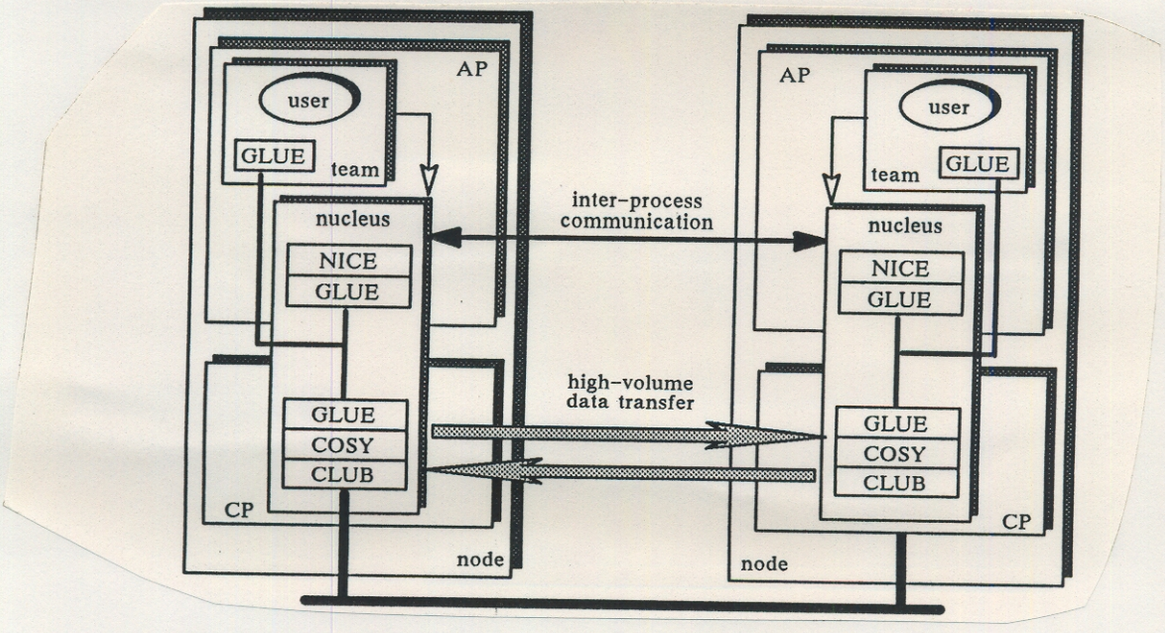

Figure 5: Functional Distributed Nucleus

In order to interface AP and CP, a *GLUE* layer is required for interconnecting NICE and COSY properly. By means of GLUE, user tasks then are directly interfaced to the CP without the need to trap via NICE. Because of the two distinguished processors, communication and computation of the same task can be performed in parallel, whereas overlapping by means of threading would always require AP multiplexing.

Typically, upon issuing a message-passing operation, the PEACE nucleus (on behalf of NICE) queues new data transfer requests into the COSY interface. Using GLUE, this queuing functionality is made available to user tasks. Hence, asynchronous communication is directly supported by hardware. Upon processing the requests, the CP (on behalf of COSY) performs high-volume data transfer between peer AP (team) address spaces.

The additional overhead implied by the GLUE layers is determined by the complexity of queue operations in a shared-memory multi-processor system. In the case of a GENESIS node being equipped with a 33 MHz Intel i860 [Intel 1989], the indivisible (high-level) queuing operation then performs approximately in less than 3 µsec (assuming 0 wait states memory).

The CP must be capable to support various types of applications with different communication styles, rather than supporting a single application only. Therefore, this configuration can only be a compromise and will be the appropriate one in cases where either tasks are mapped one-to-one onto nodes or where simple communication

patterns are given.

6.1.2. Threading within a Common Address Space

This configuration shows for two local NICE modules, one for the AP and another one for the CP. The CP is also configured with COSY and CLUB, thus it is configured with a complete nucleus that performs system-wide inter-process communication and high-volume data transfer. Figure 6 shows the resulting node organization.

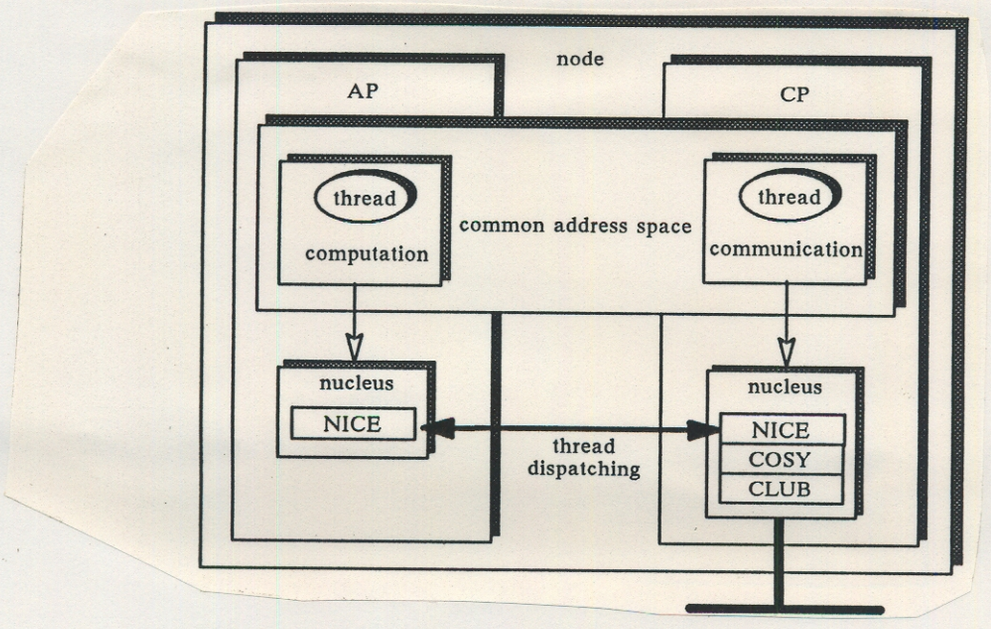

Figure 6: Threading within a Common Address Space

In contrast to the functional distributed nucleus, ready list manipulation is not jointly done by AP and CP. Each processor executes only threads of a specific type, either computing threads or communicating threads. In this model, the CP is responsible for the scheduling of communicating threads, and the task of the AP is to schedule computing threads. Both threads implicitly share a common address space. When the computing thread needs to communicate, it instructs its communicating thread to perform message passing instead. Overlapping of computation and communication of a task is achieved.

Typically, both threads are connected by two queues. The first (send) queue stores descriptors of outgoing messages, while the second (receive) queue stores descriptors of messages that have been already received. Thus, a mailbox-type of communication service is implemented. The computing thread is able to perform non-blocking sends, which may result in a local copy of each object being sent (dependent on the communication semantics). To receive a message, the computing thread inspects the receive queue. It may decide to block when the queue is empty. Queue synchronization is straightforward and performed in a task-related manner.

When the computing thread decides to block on a message to be received, it will be unblocked later on behalf of the communicating thread when the proper message arrives. For this purpose, the NICE at the CP site instructs its peer at the AP site to set

the computing thread ready to run. Vice versa, if the communicating thread decides to block because of an empty send queue, it will be unblocked later on behalf of the computing thread when the next send requested is issued.

The advantage of this configuration is that application-oriented mailbox implementations are optimally supported. This was not the case with the previously discussed configuration. Moreover, the degree of overlapping computation and communication is increased. This is because the complete user-level mailbox now is executed by the CP and not by the AP. The message startup time is significantly reduced by the fact that the computing thread does no longer execute any mailbox-specific code. Rather it merely notifies its communicating thread to generate a message that may be constituted by several user-level objects. This may imply memory management and the copying of user-level data objects. While these tasks are performed by the communicating thread, the computing thread continues execution and may poll on the event that a previously issued message has been sent out off the node.

6.1.3. Multi-Threaded Team

A single NICE module is shared by both the AP and the CP. Thus, the application task (team) as well as NICE are processed in parallel by two CPUs. Figure 7 depicts this model.

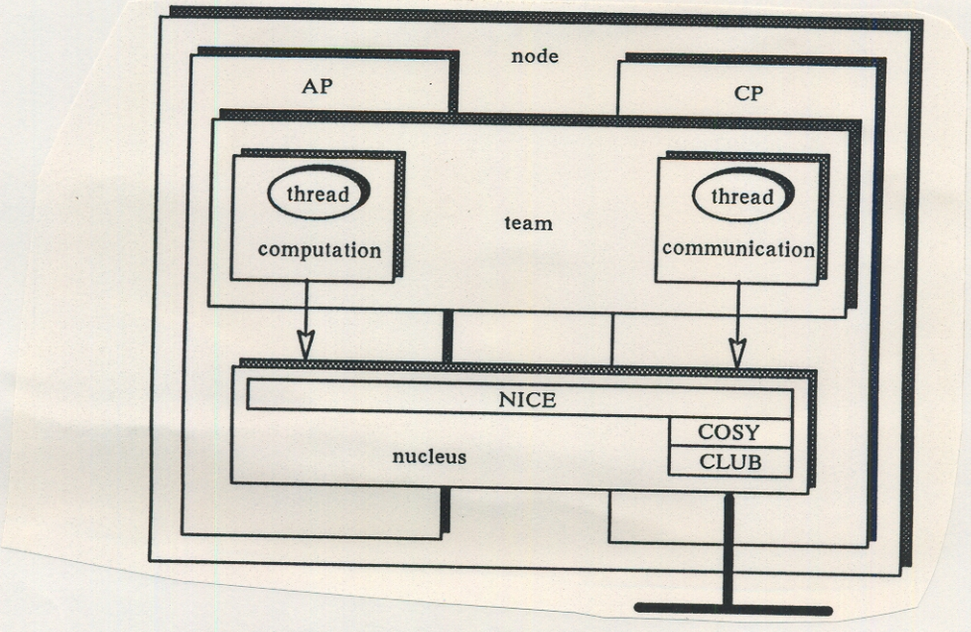

Figure 7: Multi-Threaded Team

There are no changes concerning the computing thread and the communicating thread at task (team) level. Changes only are dedicated to NICE, namely the manipulation of ready lists now is jointly performed by the AP and the CP. In the case of unblocking either of both threads, no explicit cooperation between two NICE instances is necessary. While a synchronized dispatch request queue is needed in the previous model, synchronized multi-processor scheduling is required now. The former model results in a shorter critical section during which the memory bus is locked when

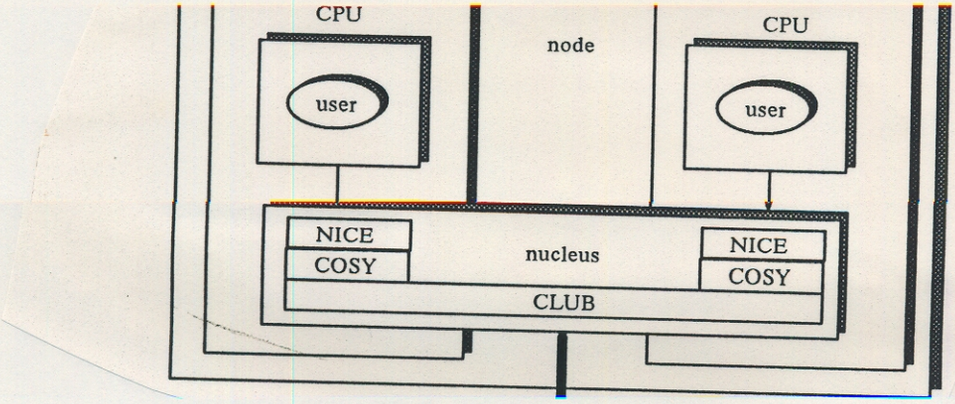

Figure 8: Local Scheduling

Access to the common network interface is by means of a dedicated CLUB module that is shared by all CPUs of the node. In this model, the CLUB module also performs multi-processor synchronization. However, the model still implements a single-

The model just presented is the appropriate one for a single-threaded mode of operation, and when the thread needs only synchronous communication. As was mentioned, the PEACE nucleus implements synchronous communication and supports asynchronous communication on a threads-basis. Asynchronous communication is still feasible with this model, however implies additional overhead caused by CPU multiplexing. As illustrated in figure 9, the entire nucleus is shared by all CPUs of the GENESIS node.

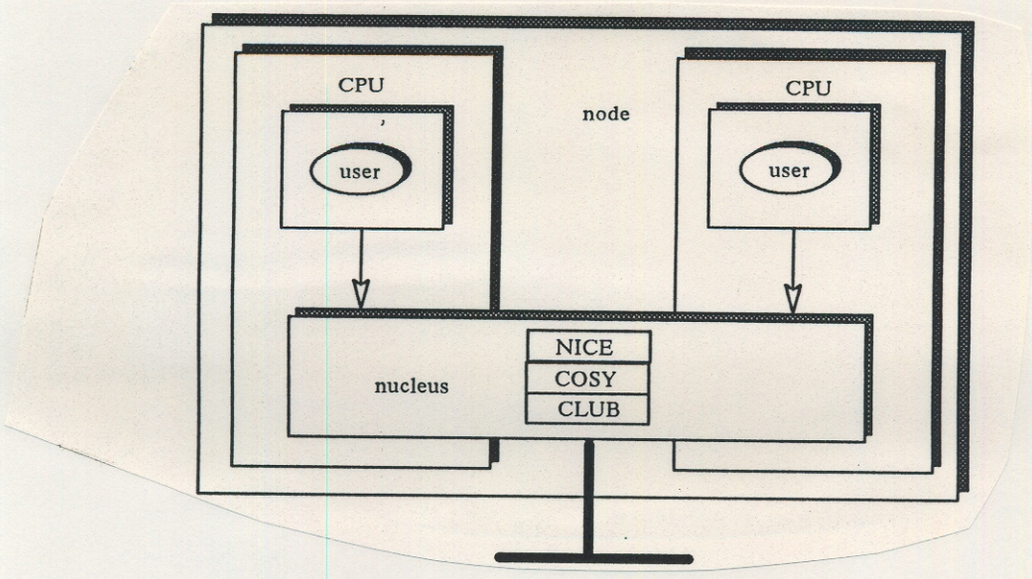

Figure 9: Global Scheduling

This model supports a mode of operation where a CPU executes any thread, independently of its vocation. While the functional dedicated organization associates computing threads with the AP and communicating threads with the CP, the symmetrical organization tries to keep the CPUs busy all the time. A much better node utilization is achieved and task throughput is increased. In the course of execution, threads change their location on behalf of NICE. As in the local scheduling model, CLUB is responsible for synchronizing access to the shared network device. In the global scheduling model it is the additional task of NICE to synchronize access to the ready list.

7. Concluding Remarks

As often required by the user community, support for massively parallel systems in general means to make the naked machine available to the parallel application program in order to obtain maximum performance. In contrast to that, distribution transparency often is required too. Both aspects significantly influenced the PEACE development and led to the idea of a nucleus family. The family approach was chosen because there cannot be a single solution for a couple of user requirements if the utmost communication performance is to be guaranteed.

For SUPRENUM, scaling transparency was a major user requirement. This led to the implementation of a nucleus family having only two members, providing protected and non-protected address spaces. Both members provide multi-tasking and were subject to benchmarking. The results show that a more sophisticated family design is required – address space protection is not the cause of communication performance problems but multi-tasking. In GENESIS, single-tasking mode of operation is a major user requirement, without sacrificing scaling transparency in general. Therefore, one must clearly distinguish between a single-tasking and a multi-tasking environment, and the nucleus family must reflect this.

we are grateful to Henning Schmidt for careful reading early manuscripts of the paper.

References

[Berg et al. 1990]
R. Berg, J. Cordsen, C. Hastedt, J. Heuer, J. Nolte, M. Sander, H. Schmidt, F.

[Cheriton 1984]
D. R. Cheriton: **The V Kernel: A Software Base for Distributed Systems**, IEEE Software 1, 2, 19-43, 1984

[Clark 1985]
D. D. Clark: **The Structuring of Systems Using Upcalls**, ACM Operating Systems Principles, Orcas Island, Washington, 1985

[Giloi 1988]
W. K. Giloi: **The SUPRENUM Architecture**, CONPAR 88, Manchester, UK., 12th-16th September, 1988

[Giloi 1990]
(Ed.), GMD FIRST, Berlin, FRG, 1990

[Habermann et al. 1976]
A. N. Habermann, L. Flon, L. Cooprider: **Modularization and Hierarchy in a Family of Operating Systems**, Comm. ACM, 19, 5, 266-272, 1976

[Herbert, Monk 1987]

Architecture, Cambridge, UK, 1987

[Intel 1989]
Intel Corporation: **i860 64-Bit Microprocessor Programmer's Reference Manual**, 1989

[Isle et al. 1977]
R. Isle, H. Goullon, K.-P. Löhr: **Dynamic Restructuring in an Experimental Operating System**, Technical Report 77-27, TU Berlin, Fachbereich 20 (Informatik), 1977

[Lantz et al. 1985]
K. A. Lantz, W. I. Nowicki, M. M. Theimer: **An Empirical Study of Distributed Application Performance**, Technical Report STAN-CS-86-1117 (also available as CSL-85-287), Department of Computer Science, Stanford University, 1985

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

[Liskov, Zilles 1974]
B. H. Liskov, S. Zilles: **Programming with Abstract Data Types**, SIGPLAN Notices, 9, 4, 1974

[Mierendorff 1989]
H. Mierendorff: **Bounds on the Startup Time for the GENESIS Node**, ESPRIT Project No. 2447, Internal Paper, GMD-F2.G1, 1989

[Mullender, Tanenbaum 1986]
S. J. Mullender, A. S. Tanenbaum: **The Design of a Capability-Based Distributed Operating System**, The Computer Journal, Vol. 29, No. 4, 1986

[Nelson 1982]
B. J. Nelson: **Remote Procedure Call**, Carnegie-Mellon University, Report CMU-CS-81-119, 1982

[Parnas 1979]
D. L. Parnas: **Designing Software for Ease of Extension and Contraction**, IEEE Transaction on Software Engineering, Vol. SE-5, No 2, 1979

[Schmidt 1990]
H. Schmidt: **Making PEACE a Dynamic Alterable System**, GMD FIRST, 1990

[Schroeder 1987]
W. Schröder: **A Distributed Process Execution and Communication Environment for High-Performance Application Systems**, Lecture Notes in Computer Science, Vol. 309 (1988), 162-188, Springer-Verlag, Proceedings of the International Workshop on "Experiences with Distributed Systems", Kaiserslautern (West Germany), Sept. 28 - 30, 1987

[Schroeder 1988]
W. Schröder: **The Distributed PEACE Operating System and its Suitability for MIMD Message-Passing Architectures**, CONPAR 88, Manchester, UK., 12th-16th September, 1988

[Schroeder, Gien 1989]
W. Schröder-Preikschat, M. Gien: **Architecture and Rationale of the GENESIS Family of Distributed Operating Systems**, ESPRIT Project No. 2447, Technical Report, GMD FIRST, Berlin, FRG, 1989

[Tanenbaum, van Renesse 1985]
A. S. Tanenbaum, R. van Renesse: **Distributed Operating Systems**, ACM Computing Surveys, Vol. 17, No. 4 (December), 1985

[van Renesse et al. 1988]
R. van Renesse, H. van Staveren, A. S. Tanenbaum: **Performance of the World's Fastest Distributed Operating System**, ACM Operating Systems Review, 22, 4, 1988