High Performance Application-Oriented Operating Systems – the EPOS Approach*

Antônio Augusto Fröhlich[1], Wolfgang Schröder-Preikschat[2]

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

Abstract—

This paper presents the EPOS approach to bring object-oriented operating systems closer to high performance parallel applications. The gap between them originates from the complexity of assembling an operating system out of a complex collection of complex classes. EPOS aims to deliver, whenever possible automatically, a customized runtime support system for each application. In order to achieve this, EPOS introduces the concepts of *scenario-independent system abstractions*, *scenario adapters* and *inflated interfaces*. An application designed and implemented following the guidelines behind these concepts can be submitted to a tool that will proceed syntactical and data flow analysis to extract a blueprint for the operating system. This blueprint is then refined by dependency analysis against information about the execution scenario acquired from the user via visual tools. The outcome of this process is a set of *selective realize keys* that will support the generation of the application-oriented operating system.

Keywords— **Object-oriented operating systems, parallel operating systems, high performance computing.**

I. Introduction

Until some years ago, high performance was an attribute associated basically to platforms running scientific computations and databases. Nowadays, more and more applications demand for such platforms: virtual reality, Web servers and even embedded systems are pushing hardware and support software for parallelism. In this context, many research projects are trying to produce low overhead operating systems that do not impact applications as much as their all-purpose relatives.

Our experiences developing runtime support systems for parallel applications [SP94b] convinced us that adjectives such as "all-purpose" and "generic" do not fit together with "high performance" and "parallel", whereas different parallel applications have quite different requirements regarding the operating system. Even apparently flexible designs, like μ-kernel based operating systems, may imply in waste of resources that otherwise could be used by applications. These observations regarding parallel applications must also hold for any application demanding non-conventional support services. Therefore, each application must have its own runtime support system, specifically designed and implemented to satisfy its requirements (and nothing but its requirements).

This paper presents the EPOS approach to deliver a high performance, application-oriented operating system to each parallel application. The following sections describe the motivation for EPOS, its fundamentals, its design and its implementation. Afterwards, some preliminary results are presented together with an outline for the project continuation.

II. Motivation for EPOS

Automatic tailoring an operating system for a given application is a challenging task that starts with the fabrication of the building blocks that will be used to assemble the operating system. A straightforward approach to conceive building blocks is to take on object orientation and its corresponding tools. In this case, reusable operating system building blocks are implemented by classes and are stored in a repository (often a class library). This approach does not produce an operating system, but a collection of classes that can be specialized and combined to yield a variety of operating systems.

Although effective, the development of operating systems based on object-oriented building blocks brings along a new issue: how to put the building blocks together. The intrinsic nature of this approach also gives rise to a gap between that what the building blocks repository offers and that what the application programmers are looking for. Paradoxically, this gap grows proportionally to the system evolution, since the most the system evolves, the larger is the number of components in the repository and the more complex they are (due to an increase in the abstraction level).

Expecting an application programmer to browse a class

*This research has been partially supported by Federal University of Santa Catarina, by CAPES Foundation grant no. BEX 1083/96-1 and by Deutsche Forschungsgemeinschaft grant no. SCHR 603/1-1.

repository to select and adapt (by aggregation or inheritance) the classes that would conduct to the best, or at least to a good, operating system for his application is not realistic. EPOS main goal is to automate the process of selecting and adapting building blocks to yield an application-oriented operating system.

EPOS is actually an extension of the PURE [SSPSS98] family based operating system, since PURE supplies the building blocks that EPOS utilizes to assemble application-oriented operating systems. The approach followed by PURE is to understand an operating system as a *program family* [Par79] and to use *object orientation* [Weg86] as the fundamental implementation discipline. PURE building blocks are implemented as C++ classes and are designed to be portable and not to incur in unnecessary overhead. Therefore, PURE classes are ideal to construct any sort of operating system.

However, PURE has not been conceived to be used by application programmers. As an example of the complexity of generating an operating system out of PURE classes, let us consider a simple nucleus to support preemptive multithreading in a C 167 μ-controller: the nucleus would be comprised by more than 100 classes exporting over 600 methods [BGP+99]. Although the resulting nucleus would not be larger than 4 Kbytes, generating it is not a trivial task.

III. FUNDAMENTALS OF EPOS

In order to deliver application-oriented operating systems, EPOS adhere to the following guidelines:

- High performance: EPOS shall give each application its own runtime support system, which shall include only those components that are really necessary to support it. Operating system components shall be as adaptable as possible, thus granting the lowest possible overhead. Besides implying in tools to analyze and generate the operating system, this goal also demands for a comprehensive repository of system components.
- Invisibility: when requested to support the execution of parallel applications formerly implemented to run on a UNIX system, specially those conforming to the MPI standard for message passing, EPOS shall support them without being visible, i.e., without requiring any modification in the application's source code. Invisibility is achieved in EPOS by supporting some UNIX APIs, including runtime libraries (`libc`, `libm`, `libstdc++` and `libf2c`), POSIX file handling, and MPI. However, as EPOS does not share any development aspect with UNIX, most of its invisibility is gained by either porting libraries or implementing abstraction layers. For instance, POSIX is supported by stubs that redirect file operations to a file server running on an I/O node, and a subset of MPI is supported as an interface on top of EPOS communication abstractions. The little scientific character of this goal gives it a low priority.
- Static configuration: guided by the high performance goal of EPOS, we decided that static configuration will have priority over dynamic. This decision arises from the fact that very few dynamic reconfigurations, in a high performance scenario, pay off the overhead to support them. Even the adoption of a dynamic prototype that would collect information about an ideal static system configuration has been suppressed, since the intrinsic overhead of a dynamic system would distort the figures for the static one. Instead, EPOS shall take on profiling to enable static reconfiguration towards the optimal.
- Parallelism in distributed memory architectures: EPOS shall extend PURE to include abstractions to support parallel computing in distributed memory architectures. This is an open goal that starts with the definition of abstractions for processes, synchronization and communication and shall evolve with applications.

IV. DESIGN OF EPOS

EPOS has been designed to reduce the gap between PURE building blocks and parallel applications. However, differently from PURE, that adopts the *program families* design strategy and relies on object orientation solely as an implementation discipline, EPOS follows the fundamentals of object-oriented design as proposed by Booch [Boo94]. The design strategy of EPOS defines three main elements: *scenario-independent system abstractions*, *scenario adapters*, and *inflated interfaces*. The two first elements tackle the gap by hiding PURE building blocks and by supporting an efficient way to construct application-ready system abstractions; the third element exports the system abstraction repository in a fashion tractable by application programmers.

A. Scenario-independent System Abstractions

By observing PURE class repository, we concluded that several classes are not of interest to application programmers. Moreover, we concluded that, differently from an application programmer, a system programmer could easily configure a bulk of application-ready classes. In EPOS, we name these application ready classes *system abstractions* and we define that it is due to the system development team to construct them. This definition, besides establishing a clear boundary between PURE and EPOS, will render a system abstractions repository with fewer components than the respective PURE building blocks repository.

In turn, when we analyzed our first abstractions, we observed that those designed to present the same functionality in different execution scenarios are indeed quite similar. Moreover, abstractions conceived to support the same scenario often differ from each other following a pattern. For

instance, two *thread* abstractions, one targeting a single-task and the other a multi-task environment, present several similarities. Likewise, a *thread* abstraction targeting a multiprocessor scenario reveals synchronization mechanisms that can also be found in the *mailbox* abstraction, since invoking methods of both abstractions implies in synchronizing eventual parallel invocations (from different processors). In this way, we propose system abstractions to be implemented as independent from the execution scenario as possible. These adaptable and scenario-independent system abstractions would then be put together with the aid of some sort of "glue" specific to each scenario. We named these "glues" *scenario adapters*, since they will adapt an existing system abstraction to a certain execution scenario.

B. Scenario Adapters

Being able to design and implement scenario-independent system abstractions gives us a chance to considerably save development time, since many system abstraction can now be reused in different execution scenarios. However, writing aspect independent abstractions and adapting them to new scenarios is everything but trivial. So far, we succeeded in adapting system abstractions to specific execution scenarios by wrapping them with *scenario adapters*. Actually, scenario adapters are not restricted to wrap system abstractions; they can also wrap, when necessary, lower level building blocks. With this strategy we have implemented, for example, a *thread* abstraction that can be adapted to be used with single or with multiple address spaces, that can be linked to the application or integrate a μ-kernel, and that supports either local or remote invocation.

In general, aspects such as application/operating system boundary crossing, concurrent invocation synchronization, remote object invocation, debugging and profiling can be easily modeled with the aid of scenario adapters, thus making system abstractions, even if not for complete, independent from execution scenarios.

The approach to write pieces of software that are independent from certain aspects and later adapt them to a given scenario has been referred to as *Aspect-Oriented Programming* [KLM+97]. We refrain from using this expression because for EPOS, differently from AOP, factors such as languages to describe aspects and tools to automatically adapt components (*weavers*) are irrelevant. If ever present in EPOS, AOP would give means, not goals. Currently, scenario adapters are implemented in EPOS using the same language used to implement system abstractions, and most of them are implemented by hand.

C. Inflated Interfaces

The combination of scenario-independent system abstractions and scenario adapters reduces the number of components in the system abstraction repository, yields application-ready abstractions and enables the automatic generation of new abstractions. However, this is not enough to bring the process of operating system construction to the application programmer level. In EPOS, this task is due to a set of automatic tools, in such a way that application programmers are no longer requested to browse repositories and to specialize or combine classes. The concept of *inflated interfaces* enables these tools and gives programmers a better way to express their applications' needs.

An EPOS *inflated interface* embraces most of the consensual definitions for a system abstraction. It is inflated because it brings together not a single view of the abstraction it exports, but a collection of its most usual representations. Examples of inflated interfaces are *thread*, *task*, *address space* and *communication channel*. The inflated interface for the *thread* abstraction gathers several different views of it, including, for example, *pthreads* and native PURE *threads*. Multiple interfaces for an abstraction are only introduced when incoherent views have to be exported. EPOS inflated interfaces are extracted from classical computer science books and system manuals, nevertheless, our users, i.e., application programmers, are welcome to suggest modifications or extensions at any time.

The adoption of inflated interfaces for system abstractions enables the application programmer to express his expectations regarding the operating system simply by writing down well-known system object invocations (system calls in non object-oriented systems). It is important to notice that inflated interfaces are mere tools to export system abstractions. They are never implemented as they are seen by the programmer, i.e., as a single class, but as a set of scenario specific classes. When configuring the system, each inflated interface is bound to one of its scenario specific implementations.

D. Selective and Partial Realize Relationships

In order to support system design based on inflated interfaces, we propose two new object-oriented design notations: *partial realize* and *selective realize*. Both notations represent relationships taking place between an inflated interface and a class that realizes that interface. However, as the name suggests, a class participating in a partial realization implements only a specific subset of the corresponding inflated interface. In this scope, selective realization means that only one of several possible realizations is connected to the inflated interface at a time. These two design notations are depicted in figure 1.

Each class joining a selective realize relationship is tagged with a key. By defining a value for this key, a specific, usually partial, realization for that interface is selected. However, during system design, these keys are not supposed to assume any value, so that an inflated interface is considered to be

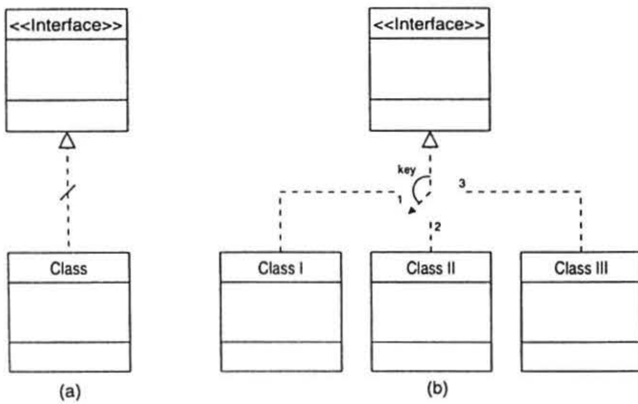

Fig. 1. Partial realize (a) and selective realize (b) relationships.

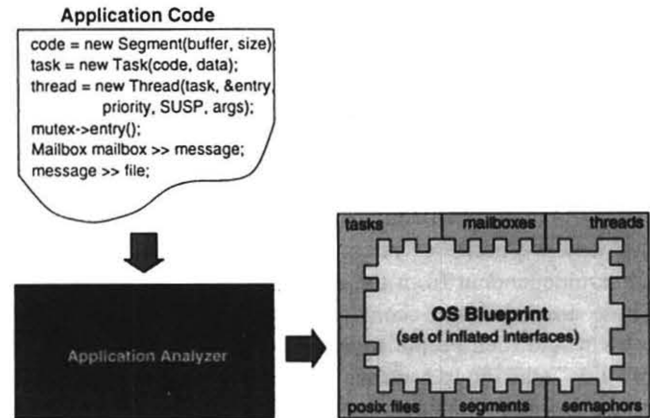

Fig. 2. Extracting an operating system blueprint.

bound to any of its realizations. The definition of selective realize keys are due the operating system generator and are not considered at design time.

Partial and selective realize design notations have their counterparts for system implementation, so that configuring an operating system can be done simply by defining values for selective realize keys. These keys are defined in a single configuration file and make conditional compilations and "makefile" customizations obsolete. Furthermore, the implementation of these relationships may be used to bind non object-oriented inflated interfaces to object-oriented implementations. This is useful, for instance, to bind an application written in Fortran or C to EPOS.

V. IMPLEMENTATION OF EPOS

With the design techniques described earlier, we can now consider the automatic generation of an application-oriented operating system. Our strategy begins top-down at the application, when the programmer implicitly specifies the operating system requirements simply by designing and coding his application while referring to the set of inflated interfaces that exports the system abstractions repository. An application designed and implemented in this fashion can then be submitted to an analyzer (figure 2) that will conduct syntactical and data flow investigations to determine which system abstractions are really necessary and how they are invoked. The outcome of this analysis is a blueprint for the operating system to be constructed, and will define, for instance, the use of multi-tasking instead of single-tasking, of multi-threading instead of single-threading, of protected address spaces instead of a single unprotected address space and so on.

Our primary operating system blueprint is, unfortunately, not complete, since there are aspects that cannot be deduced while analyzing the application. For example, the decision of whether the operating system will include support for multi-tasking or not, cannot be made based only on the application. The fact that the application does not show any evidence that multiple processes may need to run concurrently in a single processor does not necessarily mean that this situation will not occur. The multi-tasking support may be required because the application needs more processors than what is available. Several other relevant factors are often not expressed inside the application and therefore we still need user intervention to describe the application's execution scenario. However, in EPOS, the description of available resources is due to the operating system development team and the interaction with the user is done through visual tools.

Refining the operating system blueprint, by way of dependency analysis while taking in consideration the context information acquired from the user, renders a much more precise description of how the ideal operating system for a given application should look like. This refined blueprint can now be used to bind the inflated interfaces referred in the application to scenario specific implementations. For example, the inflated thread interface from the first step may have included remote invocation and migration, but reached the final step as a simple single-task, priority-scheduled thread for a certain μ-controller. The organization of an application-oriented operating system generated according to this model is depicted in the figure 3.

It is important to understand that, at the early stages of the operating system development, very often a required system abstraction will not yet be available. Even then, the proposed strategy is of great value, since the operating system developers get a precise description for the missing system abstractions. In many cases, a missing system abstraction will be quickly (automatically) adapted from another scenario using the scenario adapters described earlier.

Only if the operating system developers are not able to deliver the requested system abstractions in a time considered acceptable by the user, either because a system abstraction

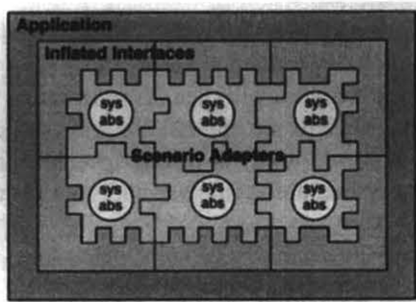

Fig. 3. Organization of an application tailored EPOS.

with that functionality have not yet been implemented for any scenario, or because the requested scenario is radically different from the currently supported scenarios, the user will be asked to select the best option from the available set of system abstractions (scenario adapters) and to adapt his program. In this way, our strategy ends where most configurable operating systems begin. Moreover, after some development effort, the combination of scenario adapters and system abstractions shall satisfy the big majority of parallel applications.

VI. PRELIMINARY RESULTS

So far we have implemented several system abstractions and scenario adapters that have been put together to assemble a few application-oriented operating systems. Perhaps, the most interesting example we can now cite is a communication channel implemented for our cluster of SMP PCs interconnect by a Myrinet network [FSP98]. Very often we face the assertion that moving communication to user level alone can bring the figures for communication close to the best. However, this affirmation is usually stated in disregard to the restrictions imposed by ordinary operating systems, like Unix and Windows NT. These systems always operate in multi-task mode, requiring the memory to be paged and avoiding the direct use of DMA to transfer a user message from host memory to the memory in the network adapter. A copy to a contiguously allocated buffer or the translation of addresses (for each memory page) has to be carried out.

However, if we consider parallel applications, which usually run on a single-task-per-node basis, the multi-task "feature" of the operating system turns into pure overhead. For multi-threaded applications, the situation is even worse, because the pipeline implemented by the most efficient user-level communication packages for Myrinet running on Unix [PT98, THIS97], which should hide the extra message copy overhead, loses its effectiveness when the pipeline stage responsible for the copy concurs with other threads for the memory bus.

We measured performance of the same communication abstraction in two execution scenarios: single-task and multi-task. The communication abstraction is the same in both cases, but a scenario adapter that performs a copy to a temporary buffer wraps it in the second case. The figures for sending messages from one node to another are depicted in figures 4 and 5, and show a difference, in favor of the single-task configuration, of about 22% for messages of 16 bytes and 46% for 64 Kbytes messages. Besides demonstrating the featherweight structure of EPOS, this example shows that it is worthwhile to give each application its own operating system.

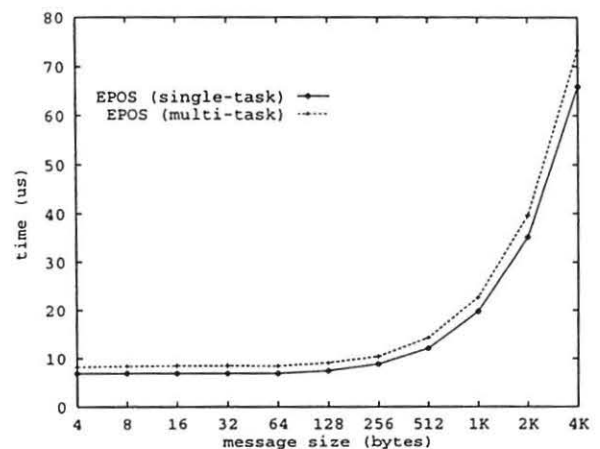

Fig. 4. Time to send a message in EPOS.

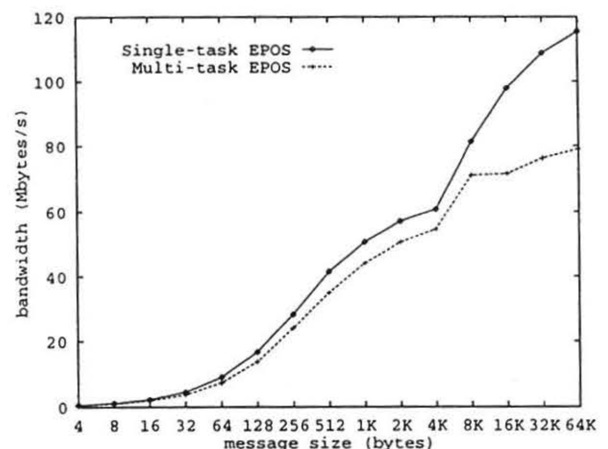

Fig. 5. Send bandwidth in EPOS.

VII. RELATED WORK

Several research projects aim to deliver operating systems that can be configured to better support a given application. They usually follow one of two strategies: kernel extensions or component based system construction. We discuss some of these projects according to the strategy they follow.

Operating system kernel extensions are usually accomplished by a μ-kernel, which implements a small set of functionality, and by a mechanism that enables applications to extend its functionality according to their needs. SPIN [BSP+95] supports system functions, written in a specific language, to be safely downloaded into the kernel from any application. VINO [SESS96] supports application code to run in the kernel address space and uses software fault isolation as a safety mechanism to avoid malicious extensions. EXOKERNEL [EKJ95] focuses on the separation of protection and management so that physical resources are securely exported to be managed by applications at user-level.

Projects in the second alternative, component based construction, usually relies on an object-oriented framework that supports system construction from a set of reusable classes. CHOICES [CJR87], one of the pioneers with this strategy, demonstrated the viability to build complex operating systems in an object-oriented framework. PEACE [SP94a] follows a family based design to implement an operating system family that comprise members to deal with specific classes of parallel applications. FLUX [FBB+97] abolishes the "core of functionality" and defines a framework in which a large set of components can be used to assemble an operating system.

EPOS approach is orthogonal to the monolith/μ-kernel/library organization, since a proper organization can be selected for each system. Just like in FLUX, the concept of a core of functionality is absent in EPOS. It differs from SPIN, VINO, EXOKERNEL and FLUX in the sense it aims to deliver application-ready operating systems, while these systems only support constructing them. Similarly to CHOICES and PEACE, EPOS defines an object-oriented framework, however, since it benefits from PURE fine-grain building blocks to implement its system abstractions, EPOS framework supports the construction of true application-oriented operating systems.

VIII. FURTHER WORK

The strategy to generate application-oriented operating systems proposed by EPOS can drastically improve application performance, because applications get only the operating system components they really need, and also because these components are fine-tuned to the aimed execution scenario. However, our strategy is not able to deliver an *optimal* operating system. Consider, for instance, the decision for a thread scheduling policy: several thread implementations, with different scheduling policies, may fit into the blueprint extracted by our tools, as long as they match the selected interfaces and satisfy the dependencies. Nevertheless, it is unnecessary to say that there is an optimal scheduling policy for a given set of threads running in a given scenario.

The decision of which variant of a system abstraction to select when several accomplish the application's requirements is, in the current system, arbitrary. Further development of EPOS shall include *profiling* primitives to collect runtime statistics. These statistics will then drive operating system reconfigurations towards the optimal. To grant an *optimal* system, however, would imply in formal specification and validation of our system abstractions, what is not in the scope of EPOS.

IX. CONCLUSION

In this paper we presented the EPOS approach to deal with the gap between object-oriented operating systems, specifically PURE, and high performance parallel applications. EPOS utilizes PURE building blocks to implement a set of *scenario-independent system abstractions* that can be adapted to a given execution scenario with the aid of *scenario adapters*. These abstractions are collected in a repository and are exported to the application programmers via *inflated interfaces*. This strategy, besides drastically reducing the number of exported abstractions, enables the programmers to easily express their application's requirements in regard to the operating system.

An application designed and implemented according to the strategy proposed in this paper can be submitted to a tool that proceeds syntactical and data flow analysis to extract a blueprint for the operating system to be generated. The blueprint is then refined by dependency analysis against information about the execution scenario acquired from the user via visual tools. The outcome of this process is a set of *selective realize* keys that supports the compilation of the application-oriented operating system.

The results obtained so far demonstrate the viability of constructing application-oriented operating systems and also the benefits an application can get by running on its own system. However, EPOS is now closer to its beginning than to its end: we have quite few scenario adapters implemented and the tools described in this paper are under construction.

REFERENCES

[BGP+99] D. Beuche, A. Guerrouat, H. Papajewski, W. Schrder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, St Malo, France, May 1999.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, USA, 1994.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the spin Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, USA, December 1995.

[CJR87] R. Campbell, G. Johnston, and V. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, 1987.

[EKJ95] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[FBB+97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[FSP98] A. A. Fröhlich and W. Schröder-Preikschat. SMP PCs: A Case Study on Cluster Computing. In *Proceedings of the 24th Euromicro Conference - Workshop on Network Computing*, pages 953–960, Västeras, Sweden, August 1998.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C.s Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP'97, Lecture Notes in Computer Science*, pages 220–242, Springer-Verlag, 1997.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering*, SE-5(2), 1979.

[PT98] L. Prylli and B. Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Proceedings of the International Workshop on Personal Computer based Networks Of Workstations*, Orlando, USA, April 1998.

[SESS96] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, 1996.

[SP94a] W. Schröder-Preikschat. PEACE - A Software Backplane for Parallel Computing. *Parallel Computing*, 20:1471–1485, 1994.

[SP94b] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice-Hall, 1994.

[SSPSS98] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, October 1998.

[THIS97] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High Performance Communication Library. *High-Performance Computing and Networking '97*, April 1997.

[Weg86] P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986.