On Component-Based Communication Systems for Clusters of Workstations

Antônio Augusto Fröhlich
GMD-FIRST
Kekuléstraße 7
12489 Berlin, Germany
guto@first.gmd.de

Wolfgang Schröder-Preikschat
University of Magdeburg
Universitätsplatz 2
39106 Magdeburg, Germany
wosch@ivs.cs.uni-magdeburg.de

Abstract

Most of the communication systems used to support high performance computing in clusters of workstations have been designed focusing on "the best" solution for a certain network architecture. However, a definitive best solution, independently of how well tuned to the underlying hardware it is, cannot exist, for parallel applications communicate in quite different ways. In this paper, we describe a novel design method that supports the construction of run-time systems as an assemblage of components that can be configured to closely match the demands of any given application. We also describe how this method has been deployed in the development of a communication system in the realm of EPOS*, a project that aims at delivering automatically generated application-oriented run-time support systems. The communication system in question has been implemented for a cluster of PCs interconnected with Myrinet, and corroborates the effectiveness of the proposed design method.*

1. Introduction

The parallel computing community has been using clusters of commodity workstations as an alternative to expensive parallel machines for several years by now. The results obtained meanwhile, both positive and negative, often lead to the same point: inter-node communication. Consequently, much effort has been dedicated to improve communication performance in these clusters: from the hardware point of view, high-speed networks and fast buses provide for low-latency and high-bandwidth; while from the software point of view, *user-level communication* [1] enables applications to access the network without operating system intervention, significantly reducing the software overhead on communication. Combined, these advances enabled applications to break the giga-bit-per-second bandwidth barrier.

Nevertheless, good communication performance is hard to obtain when dealing with anything but the test applications supplied by the developers of the communication package. Real applications, not seldom, present disappointing performance figures. We believe the origin of this shortcoming to be in the attempt of delivering generic communication solutions. Most high performance communication systems are engaged in a "the best" solution for a certain architecture. However, a definitive best solution, independently of how well tuned to the underlying architecture it is, cannot exist, since parallel applications communicate in quite different ways. Aware of this, many communication packages claim to be "minimal basis", upon which application-oriented abstractions can (have to) be implemented. Once more, there cannot be a best minimal basis for all possible communication strategies. This contradiction between generic and optimal is consequently discussed in [14].

If applications communicate in distinct ways, we have to deliver each one a tailored communication system that satisfies its requirements (and nothing but its requirements). Of course we cannot implement a new communication system for each application, what we can do is to design the communication system in such a way that it is possible to tailor it to any given application. In the *Embedded Parallel Operating System* (EPOS) project [5], we developed a novel design method that is able to accomplish this duty. EPOS consists of a collection of components, a component framework, and tools to support the automatic construction of a variety of run-time systems, including complete operating systems.

The particular focus of this paper is on EPOS communication system, which has been implemented for a cluster of PCs interconnect by a Myrinet high-speed network. In the next sections, the *Application-Oriented System Design* method will be introduced, followed by a case study of its applicability to design a communication system. The implementation of this communication system is later discussed, including a preliminary performance evaluation. The paper is closed with authors' conclusions.

2. Application-Oriented System Design

Application-Oriented System Design (AOSD) is a novel operating system[1] design method that, as the name suggests, has a strong compromise with applications. Its main goal is to produce run-time support systems that can be tailored to fulfill the requirements of particular applications. Accomplishing this task begins with the decomposition of the operating system domain in abstractions that are natural to application programmers. This is exactly the decomposition strategy promoted by *Object-Oriented Design* [3] and may sound obvious to application designers, but most system designers simply neglect the problem domain and let implementation details, such as target architecture, programming languages, and standardized interfaces, guide the design process [11]. Application programmers, not seldom, get run-time systems that barely resemble the corresponding domain.

The next step is to model software components that properly represent the abstractions from the decomposed domain. Extensive components, that encapsulate all perspectives of an abstraction in a single entity, are not an alternative, since we want components to closely match the requirements of particular applications. A more adequate approach would be to apply the commonality and variability analysis of *Family-Based Design* [10] to yield a family of abstractions, with each member capturing a significant variation and shaping a component. Nevertheless, this approach has the inconvenient of generating a high number of components, thus increasing the complexity of the composition process. We handle this drawback by exporting all members of a family through a single *inflated interface*. In a system designed accordingly, adequate members of each required family could be automatically select by a tool that performs syntactical analysis of the corresponding application's source code.

Another important factor to be considered while modeling abstractions is scenario independence. When a designer realizes, for instance, that a communication mechanism may have to be specialized in order to join a multithreaded scenario, he has to choose between modeling a new family member and capturing this scenario dependency in a separate construct. Allowing abstractions to incorporate scenario dependencies reduces their degree of reusability and produces an explosion of scenario-dependent components. Therefore, an application-oriented design should try to avoid it, only allowing those variations that are inherent to the family to shape new members. The resulting *scenario-independent abstractions* shall be reusable in a larger variety of scenarios, some of them unknown at the time they were modeled.

Scenario specificities, in turn, can be captured in constructs like the *scenario adapters* described in [6]. Because scenario adapters share the semantics of collaborations in *Collaboration-Based Design* [13], one could say that an abstraction collaborates in a scenario. This separation of abstractions and scenarios is also pursued by *Aspect-Oriented Programming* [7], nevertheless, though it provides means to support this separation, it does not yet feature a design method.

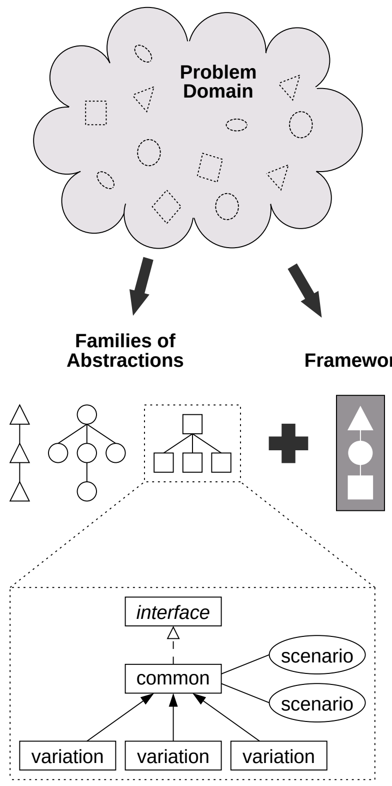

Figure 1. An overview of Application-Oriented System Design.

After decomposing the problem domain in scenario-independent abstractions and scenario-adapters, organizing the solution domain accordingly becomes straightforward. *Inflated interfaces* hide most details of the solution domain by exporting all members of a family of abstractions, as well as the corresponding scenario adapters, through a single interface. Since these interfaces emanate directly from the problem domain, application programmers should feel

[1]The term "operating system" is used here in its broadest meaning, encompassing all kinds of run-time support systems.

comfortable to use them. What is missing to deliver a true application-oriented run-time system is a way to assemble components together correctly and efficiently. By correct assembly we mean preserving the individual semantics of each component in the presence of others and under the constraints of an execution scenario. By efficient assembly we mean preserving their individual efficiency in the resulting composite.

One possibility to produce the desired compositions is to capture a reusable system architecture in a *component framework*. A framework enables system designers to predefine the relationships between abstractions and therefore can prevent misbehaved compositions. Furthermore, a framework defined in terms of scenario adapters can achieve a high degree of adaptability. Efficient composition can be accomplished if the framework uses *Generative Programming* techniques [4], such as *static metaprogramming*. Since static metaprograms are executed at compile-time, a statically metaprogrammed framework can avoid most of the overhead typical of traditional object-oriented frameworks. It is also important to notice that, though component composition would take place at compile-time, nothing would prevent components from using dynamic reconfiguration mechanisms to internally adapt themselves.

In brief, *Application-Oriented System Design* (figure 1) is a multiparadigm design method that supports the construction of customizable run-time support systems by decomposing the system domain in families of reusable, scenario-independent abstractions and the corresponding scenario adapters. Reusable system architectures are modeled as component frameworks that can guide the compilation of the target system. Application programmers interact with the system through inflated interfaces, without having to know details about the organization of families or scenarios.

3. The Design of an Application-Oriented Communication System

We applied Application-Oriented System Design to develop a communication system for clusters of workstations in the realm of project EPOS. By decomposing the domain of high-performance cluster communication, we obtained two families of abstractions: `Network` and `Communicator`. The first family abstracts the physical network as a logical device able to handle one of the following strategies: *datagram*, *stream*, *active message* (AM), *asynchronous remote copy* (ARC), or *distributed shared memory* (DSM). Since system abstractions are to be independent from execution scenarios, aspects such as access control and sharing are not modeled as properties of `Network`, but as "decorations" that can be added by scenario adapters. EPOS family of `Networks` is depicted in figure 2.

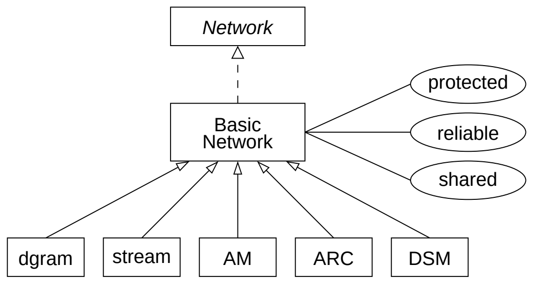

Figure 2. The Network family.

For most of EPOS abstractions, architectural aspects are also modeled as part of the execution scenario, however, network architectures vary drastically, and implementing portable abstractions would certainly push performance bellow acceptable levels. Consider, for instance, the architectural differences between Myrinet and SCI: a portable active message abstraction would underestimate Myrinet, while a portable asynchronous remote copy abstraction would misuse SCI. Therefore, the family of `Network` abstractions will be individually designed and implemented for each desired network architecture. Some family members that are not directly supported by the architecture will be emulated, because we believe that, if the application really needs (or wants) them, it is better to emulate them close to the hardware.

The second family of abstractions deals with communication end-points. These are the abstractions effectively used by applications to communicate with each other. EPOS family of `Communicators` is shown in figure 3 and has the following members: *connection*, *port*, *mailbox*, *active message handle*, *asynchronous remote copy segment*, and *distributed shared memory segment*. Again, scenario dependencies such as multitasking and multithreading are modeled as scenario adapters.

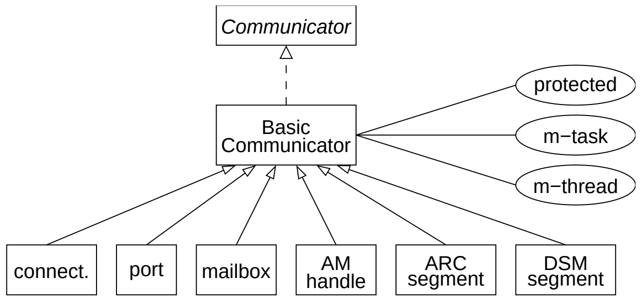

Figure 3. The Communicator family.

These two families, when completely implemented for a variety of network architectures, will yield a large number of components that will be stored in a repository together with several other subsystems. With such a large number of components, selecting and configuring the right ones in order to produce an application-oriented system may become a defying activity, even when assisted by visual tools. Hence, Application-Oriented System Design proposes all members of a family to be exported through a single, inflated interface. In this way, application programmers can design and implement their applications referring to fewer interfaces and ignoring the particular properties of each component. Actually, the programmer catches a comprehensive perspective of the family, as though a super-component was available, and uses the operations that better fulfills his requirements[2]. As an example, the inflated interface of the `Communicator` family is depicted in figure 4.

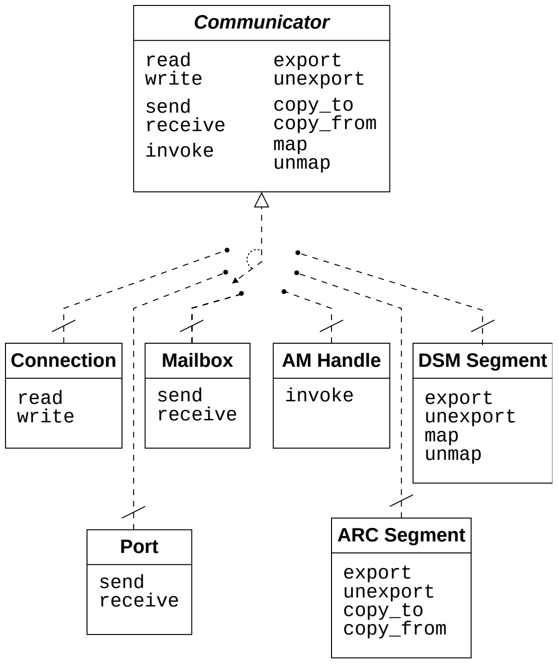

Figure 4. The Communicator inflated interface and its realizations.

The process of binding an inflated interface to one of its realizations can be automated if we are able to clearly distinguish one realization from another. In EPOS, we identify realizations through the signatures of their methods, so that syntactical analysis of application source code can identify which of the realizations are needed. If two realizations present the same set of signatures, as with `Port` and `Mailbox` in figure 4, syntactical analysis may not be enough to decide for one of them, and user intervention may be required. Nevertheless, although `Port` and `Mailbox` differ only semantically[3], the syntactical analysis of other components may render one possibility invalid. For example, if the application is known to execute on a single-task-per-node basis, a scenario with multiple receivers is not possible, breaking the tie in favor of `Port`.

The set of selected family members, in addition to information obtained from the user, defines the execution scenario for the application. As proposed by Application-Oriented System Design, scenario peculiarities are applied to abstractions by means of scenario adapters. In EPOS, a scenario adapter wraps an abstraction as to enclose invocations of its operations between the `enter` and `leave` scenario primitives (see figure 5). Besides enforcing scenario specific semantics, a scenario adapter can extend the state and behavior of an abstraction, for it inherits from both scenario and abstraction. For example, all abstractions in a scenario could be tagged with a capability, without internal modifications, by associating the capability with the corresponding scenario.

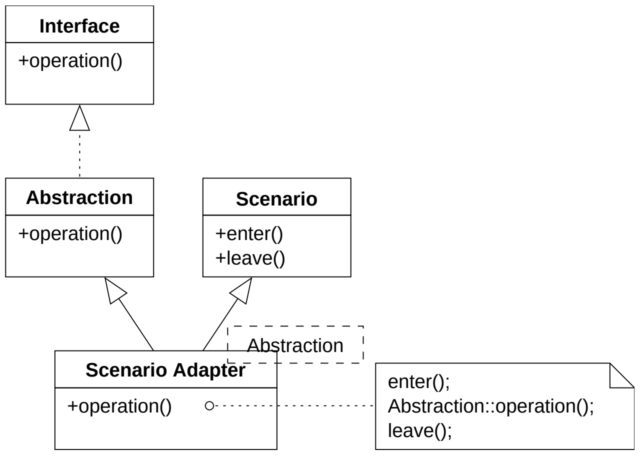

Figure 5. The structure of a scenario adapter.

EPOS statically metaprogrammed framework is defined around a collection of interrelated scenario adapters. As shown in figure 5, scenario adapters are designed as parametrized classes that take a component (abstraction) as parameter. Hence they can act as placeholder for components in the framework. In order to generate a system, information about the mapping of inflated interfaces to realizations, and also about system-wide properties such as target architecture and protection, are passed as input to the metaprogram. The resulting system would include only the components needed to support the corresponding application in the respective execution scenario.

[2] In case an application programmer with enough expertise about the system wishes to extend a component, or bypass automatic configuration, the individual interfaces of each family member are also made available.

[3] Both `Port` and `Mailbox` support multiple senders, but the first supports a single receiver, while the second support multiple receivers too.

4. The Implementation of the Communication System

Following the design described earlier, EPOS is being implemented as a collection of components, a framework, and a set of tools that support automatic generation of application-oriented run-time systems. Currently the system can run in two modes: native on IX86 computers, and guest on LINUX systems. The IX86-native version can be configured either to be embedded in the application (single-task), or as μ-kernel (multi-task), while the LINUX-guest version comprises a library and kernel loadable modules.

Components are being implemented in C++ and described in XML. The XML description is used by the tools that support automatic system generation. The framework is also implemented in C++, but mainly with its built-in static metalanguage. The tools to proceed syntactical analysis of applications, to configure the target system, and to check configuration dependencies are made available to users through a compiler wrapper similar to `mpicc`. This enables users to implicitly generate the run-time system during the compilation of applications. Nevertheless, if these tools fail to configure the system, user intervention is requested via an interactive graphical tool that supports configuration adjustments by feature selection[4].

EPOS family of communication abstractions is currently being implemented for the Myrinet high-speed network [2]. So far, we concluded the implementation of the `Datagram Network`, the `Port` and the `Mailbox Communicators`, and a mechanism to support *remote object invocation* (ROI). These components can be adapted to the following scenarios: `Protected`, `Multitask`, `Multithread`, and `Global`. The `Protected` scenario ensures that only authorized agents gain access to abstractions. The `Multitask` and `Multithread` scenarios adapt abstractions to execute in the presence of multiple tasks and threads. The `Global` scenario adapts abstractions to interact in a cluster-wide environment of active objects, hiding communication behind ordinary method invocations.

With these components we generated a couple of application-oriented run-time systems. One of them supports two simple applications that communicate intensively in a producer/consumer fashion. For this purpose, they use the `Datagram Network` and the `Port Communicator`. Figures 6 and 7 show respectively the latency and the bandwidth available to these applications in both IX86-native and LINUX-guest modes. The hardware test-bed for this measurements consisted of two PCs connected to the same Myrinet switch. Each PC has a 266 MHz Pentium II processor, 128 MBytes of memory (10 ns DRAM) on a 66 MHz bus, and a 32-bits Myrinet NIC on a 33 MHz PCI bus.

[4]The same tool can be used to tailor the system manually.

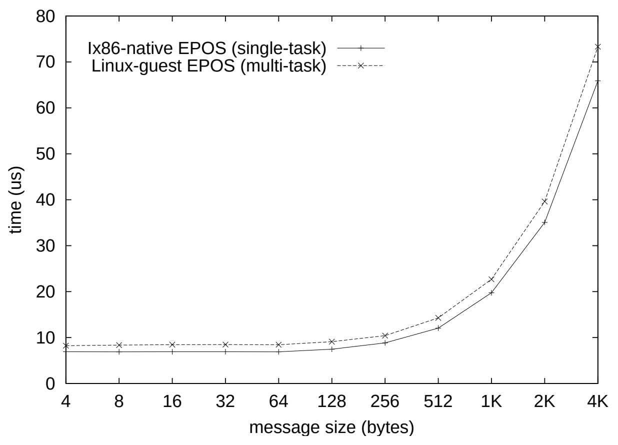

Figure 6. Datagram/Port one-way latency.

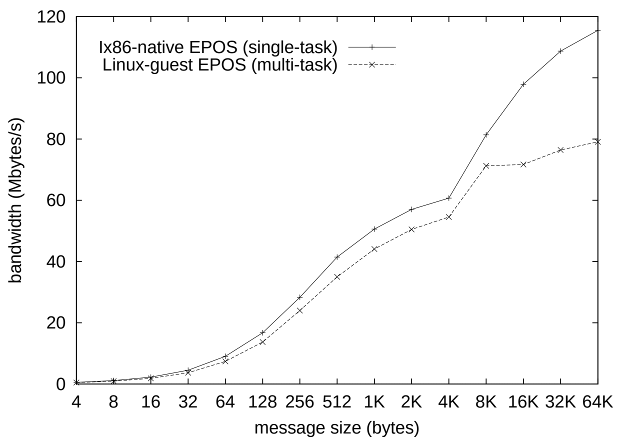

Figure 7. Datagram/Port one-way bandwidth.

The difference in favor of the IX86-native version arises from the contiguous memory allocation method adopted, which allows the DMA engines on the Myrinet card to be programmed with logical addresses and eliminates an additional message copy into a system DMA buffer. This difference could have been even more expressive if the applications were multithreaded, since the extra copy would have concurred with application threads for processor time and especially for memory bandwidth. Nevertheless, most parallel applications execute on a single-task-per-node basis and will benefit from the single-task versions of EPOS. Other communication systems, such as Active Messages [8], Fast Messages [9], PM [15], and BIP [12], run exclusively on top of ordinary operating systems, such as UNIX or WINDOWS NT, and have no alternative to escape the extra copy than making a system call to translate logical addresses into physical ones, what is usually even more time consuming[5].

Furthermore, EPOS quality evaluation should not be restricted to performance. Because only the components effectively required by the application are included, the resulting system is usually extremely compact. The system in the example above, which in addition to communication also includes process and memory management, has a size of 11 KBytes. This means less resource consumption and less space for bugs. Usability is also improved, since EPOS visible interfaces are defined in the context of applications.

5. Conclusion

In this paper we introduced *Application-Oriented System Design*, a novel design method that prevents the monolithic conception of generic solutions that fail to scale along with application demands. We also described how this method has been deployed to construct a communication system for the Myrinet high-speed network. This communication system, implemented in the realm of project EPOS, consists of a collection of *application-ready, scenario-independent abstractions* (components) that can be adapted to specific execution scenarios by means of *scenario-adapters* and can be arranged in a *statically metaprogrammed framework* to produce an application-oriented communication system. The system is presented to application programmers through *inflated interfaces* that gather all variations of an abstraction (family members) under a single comprehensive interface. By programming based on these interfaces, programmers enable EPOS tools to automatically generate an adequate system for their applications.

The results obtained so far are highly positive and help to corroborate the guidelines of *Application-Oriented System Design*, as well as EPOS design decisions. The evaluation of EPOS communication system revealed performance figures that, as far as we are concerned, have no precedents in the history of PC clusters interconnected with 32-bits Myrinet. Nevertheless, EPOS is a long term project that aims at delivering application-oriented run-time systems to a large universe of applications. Therefore, several system abstractions, scenario adapters, and tools are still to be implemented or improved.

References

[1] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[3] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[5] A. A. Fröhlich and W. Schröder-Preikschat. Tailor-made Operating Systems for Embedded Parallel Applications. In *Proceedings of the 4th IPPS/SPDP Workshop on Embedded HPC Systems and Apllications*, pages 1361–1373, San Juan, Puerto Rico, Apr. 1999.

[6] A. A. Fröhlich and W. Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[8] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of Supercomputing'97*, Sao Jose, USA, Nov. 1997.

[9] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2), June 1997.

[10] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Mar. 1976.

[11] R. Pike. Systems Software Research is Irrelevant. Online, Feb. 2000. [http://cm.bell-labs.com/who/rob/utah2000.ps].

[12] 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, Apr. 1998.

[13] T. Reenskaug, E. P. Andersen, A. J. Berre, A. Hurlen, A. Landmark, O. A. Lehne, E. Nordhagen, E. Nêss-Ulseth, G. Oftedal, A. L. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-oriented Systems. *Journal of Object-oriented Programming*, 5(6):27–41, Oct. 1992.

[14] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice-Hall, Englewood Cliffs, U.S.A., 1994.

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

[5] Sharing system DMA buffers with applications is not really an alternative, because they are usually restricted in size and will not be able accommodate the large data structures typical of parallel applications. It would most likely result in the application performing the additional copy.