EPOS: an Object-Oriented Operating System *

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

[1] GMD FIRST *guto@first.gmd.de*

[2] University of Magdeburg *wosch@ivs.cs.uni-magdeburg.de*

Abstract

This position paper reports the current development stage of project EPOS, which aims to deliver, whenever possible automatically, a customized runtime support system for each (high performance) parallel application. In order to achieve this, EPOS introduces the concepts of *adaptable, 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 an operating system blueprint. This blueprint is then submitted to another tool that will tailor an operating system for the given application.

1 Introduction

Our experiences developing runtime support systems for parallel applications [5] 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. Therefore, each application must have its own runtime support system, which has been specifically designed and implemented to satisfy its requirements (and nothing but its requirements).

Automatic tailoring an operating system for a given application is a challenging task that begins 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 will be implemented by classes and will be stored in a repository (usually 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 strategy of developing operating systems based on 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 more the system evolves, the more its repository grows.

The process of selecting and adapting building-blocks to generate an application-oriented operating system is the task of EPOS. EPOS is actually an extension of the PURE [3] family based operating system, since PURE supplies the building-blocks that EPOS utilizes. PURE

*Work partially supported by UFSC, by CAPES grant no. BEX 1083/96-1 and by DFG grant no. SCHR 603/1-1.

building-blocks are implemented as C++ classes and are designed to be portable and not to incur in unnecessary overhead, therefore they are suitable 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 multi-threading on a C 167 μ-controller: the nucleus would be comprised by more then 100 classes exporting over 600 methods [2]. Although the resulting nucleus would not be bigger than 4 Kbytes, generating it is a quite complex task. EPOS aims to automate this task in such a way to make it tractable by application programmers.

This position paper reports the current development stage of project EPOS. The following sections describe the design of EPOS as well as key points of its implementation. Afterwards, some preliminary results are presented together with an outline for the project continuation.

2 The Design of EPOS

EPOS has been designed to reduce the gap between PURE building-blocks and high performance parallel applications. To achieve this, EPOS follows the fundamentals of object-oriented design proposed by Booch [1] and introduces three main design elements: *adaptable, scenario independent system abstractions*, *scenario adapters*, and *inflated interfaces*. The two first elements tackle the gap to application by hiding PURE building-blocks and by enabling the automatic generation of new system abstractions; the third element effectively exports EPOS repository.

2.1 Adaptable, Scenario Independent System Abstractions

When observing the 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 can easy 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, renders a system abstractions repository (the EPOS repository) with far less components than the corresponding PURE repository.

In turn, when we analyzed the early EPOS repository, we observed that those abstractions designed to present the same functionality in different execution scenarios are indeed quite similar. In the same manner, 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 multi-processor scenario reveals synchronization mechanisms that can also be found in the *mailbox* abstraction for that same scenario, 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, 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.

2.2 Scenario Adapters

Being able to design and implement adaptable, scenario independent system abstractions give us a chance to considerably save development time, since many system abstractions 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, PURE 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 support 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.

2.3 Inflated Interfaces

The combination of adaptable, scenario independent system abstractions and scenario adapters makes EPOS repository much simpler than the original PURE repository. It also enables the automatic generation of new system 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 some set of automatic tools, in such a way that application programmers will no longer be requested to browse the repository 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* for a system abstraction embraces most of the consensual definitions for that abstraction. It is inflated because, instead of a single representation, it brings together the most usual representations for the abstraction it exports. 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 will only be present 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 to these interfaces at any time.

The use of inflated interfaces for system abstractions shall enable 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) while referring to its inflated interface. It is important to notice that inflated interfaces are mere tools to export system abstractions. They will never be implemented as a single class, but as a (possibly huge) set of scenario specific ones. A tool shall bind (reduce) inflated interfaces to one of its specific implementations.

3 The Realization of EPOS

With the design mechanisms described earlier, we can now consider the automatic generation of an application-oriented operating system. Our strategy begins top-down at the application, with the programmer specifying application requirements regarding operating system by designing/coding the application while referring to the set of inflated interfaces that exports the system abstractions repository. An application designed and implemented in this fashion can now be submitted to an analyzer that will conduct syntactical and data flow investigations to determine which system abstractions are really necessary to support the application and how they are invoked. The output of this analysis is a blueprint for the operating system to be constructed (figure 1).

Our primary operating system blueprint is, unfortunately, not complete, since there are aspects that can not be deduced by analyzing the application. Factors such as target architecture, number of processors available, network architecture and topology are fundamental to tailor a good operating system but are usually not expressed inside the application. Therefore, we still need user intervention to identify the application's execution scenario. The description of the

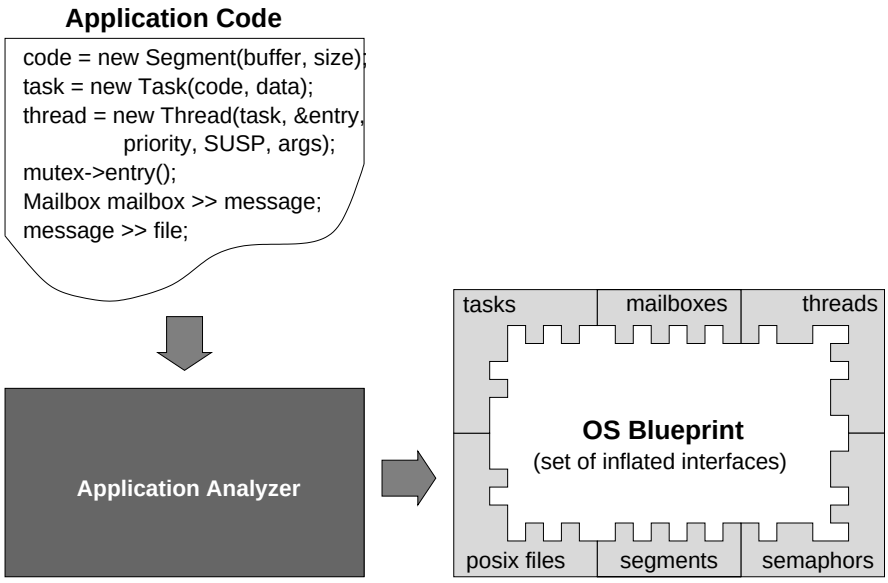

Figure 1: Extracting an operating system blueprint.

available resources however is due to the operating system developers and the interaction with the user will be done via visual tools.

The refinement of the operating system blueprint, through dependency analysis against the context information acquired from the user, will render a more precise description of how the operating system for a given application should look like. This refined blueprint can now be used to bind the inflated interfaces referred by the application programmer 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 single-task, priority-scheduled thread for a certain processor. The application-oriented EPOS resulting from this process is organized as shown in figure 2.

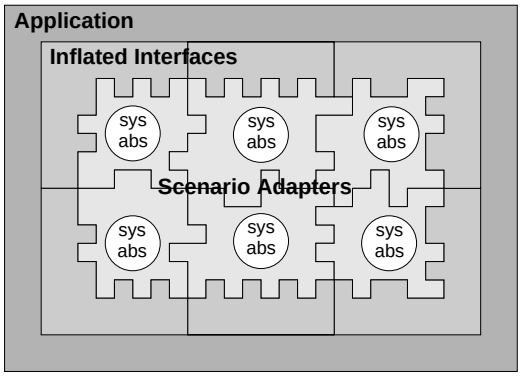

Figure 2: Organization of an application tailored EPOS.

4 Preliminary Results

So far we have implemented several system abstractions and scenario adapters that have been put together to assembly 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 [4]. Very often we face the affirmation that moving communication to user level can alone 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 a 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 on 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 MPI applications, which usually run on a single-task-per-node basis, the multi-task "feature" of the operating system turns into pure overhead. 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 it is wrapped by a scenario adapter that performs a copy to a temporary buffer for the second case. The results show a difference, in favor of the single-task configuration, of about 22% for messages of 16 bytes and 46% for 64 Kbytes messages. This makes evident that it pays off to give each application its own operating system.

5 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. 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 system abstractions repository is now quite small and the tools described in this paper are still under construction.

References

[1] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[2] D. Beuche et al. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the Second IEEE International Symposium on Object-Or iented Real-Time Distributed Computing*, St Malo, France, May 1999.

[3] F. Schön et al. 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.

[4] A. Fröhlich and W. Preikschat. SMP PCs: A Case Study on Cluster Computing. In *Proceedings of the First Euromicro Workshop on Network Computing*, Västeras, Sweden, August 1998.

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