On Architecture Transparency in Operating Systems*

Danilo Beuche, Antônio Augusto Fröhlich†, Reinhard Meyer, Holger Papajewski, Friedrich Schön†, Wolfgang Schröder-Preikschat, Olaf Spinczyk, Ute Spinczyk

University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany

†GMD FIRST
Kekuléstraße 7
D-12489 Berlin, Germany

wosch@{cs.uni-magdeburg,first.gmd}.de

1 Architectures Revisited

Operating-system development for deeply embedded parallel/distributed systems sometimes may become a fairly challenging undertaking. Here, the phrase "deeply embedded" refers to systems forced to operate under extreme resource constraints in terms of memory, CPU, and power consumption. The notion "parallel/distributed" relates to the fact that embedded systems are becoming more and more complex. Typical cases where appliances are to be controlled by systems like this come from the automotive field. From the perspective of a computer-science engineer, today's cars are distributed systems on wheels. Their operation is made feasible by a fairly large number of networked electronic control units (ECU), or μ-controllers, with each ECU being equipped with a thimble full of memory only. Depending on the requested furnishings, cars with 70 ECUs and 4 KB of memory per ECU are no rarity any longer. Aircrafts are much more complexer distributed systems: a Boing 747 is flown by about 1400 processors. The market of such systems is huge[1] and is subject to an enormous cost pressure. Although the cost of some networked ECUs are in many cases far below that of a single "standard PC", their operating-system demands are often much more challenging.

When looking into some more detail at the functions a deeply embed-able operating system has to provide, one will identify many commonalities with contemporary, e.g. UNIX-like, operating systems. In many cases some sort of process model needs to be supported, interrupt handling and device-driver services are required, synchronization becomes a demanding issue, memory and resource management needs to be provided, and network communication can not be sacrificed [13]. However, the given resource limits prevent the use even of compact microkernels such as QNX [15] and L4 [10].

One often hears arguments saying that systems like these have been developed for other (namely more general) purposes, thus bringing them into discussion here would mean comparing apples with oranges. There is a word of truth in it, but the salient point regards the (internal) design decisions of these systems that limit to some extent the applicability to a broader raange. Some decisions have been met too early during the system design phase. A typical example is an assumption that a context switch always means exchanging the contents of CPU registers and the address-space mapping, because a process always has to execute within its own address space. In this case, a specific architecture or outward manifestation

*This work has been partly supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-2 and the Bundesministerium für Bildung und Forschung (BMBF), grant no. 01 IS 903 D 2.

[1] [8] reports that in year 2000 about 8 billions microprocessors will be manufactured. About 2 % of which will go into the PC market, while 90 % are dedicated to embedded systems. About 5 billions of all will be 8-bit microprocessors.

of the operating system draws throughout many components. This limits component reusability and/or lets streamlining become somewhat difficult if not impossible.

Which operating-system architecture is the best, e.g. monolithic or based on a micro-kernel, promptly becomes a question of philosophy. To express that micro-kernels could be even more compact than they appear, nano-kernels have been introduced. The limitations of nano-kernels, in turn, motivated the invention of pico-kernels.[2] The dispute on this is not new—and questionable, since all the architectures are compatible to each other [9]. The choice of architecture should better be a question of the actually to be created system configuration and depends on the application field of the resulting operating system. That is to say, architecture should be considered a *non-functional property* of an operating system. There are many different *aspects* under which the system components may have to operate. To make them reusable for various application scenarios, design and implementation of each of the components should be architecture transparent.

2 Operating Systems and Aspect Orientation

In order to develop (operating-system) software for a broad application spectrum, design decision that restrict applicability must be postponed as far as possible. Perhaps certain decisions will never be made inside the system, but rather considered a case for the application programs to be supported. When carefully translated into action, this classical bottom-up design approach leads to a *program family* [14]. Strictly speaking, design decisions are met bottom-up, but the design process is controlled in a top-down manner. The idea is to design family members that are particularly tailored to support specific application scenarios by sharing as many as possible system abstractions, i.e. reusable components. A highly distinct *functional hierarchy* of "fine-grain sized" components is the outcome. The entire system structure is a logical one in the sense that the design is hierarchical, and not its implementation [6]. Combining this approach with an object-oriented implementation may result in highly flexible and yet efficient system structures [3].

Design decisions related to any sort of system architecture should be postponed in the above mentioned sense, and ideally never be made. As an example, Figure 1 illustrates the vision of an operating system that follows these design ideas. The goal of the PURE system [16] is to give applications exactly the resources they need to perform their tasks, no more and no less. In this sense PURE aims at being a "perfect universal runtime executive" for any application. This includes runtime as well as operating systems.

The figure shows the functional hierarchy of a PURE operating system on an intentional abstract level. The building blocks depicted in the vertical refer to various aspects, applications as well as system extensions may be subjected to. These aspects [3] describe different architecture manifestations. Each of these building blocks stands for a more or less complex sub-system, the designs of which are strictly family-based. Their association with a certain level in the overall hierarchy manifests a specific operating-system architecture. For example, pushing threads and address spaces in between scheduling and memory, and layering the others above persistency, results into a micro-kernel-like system organization. The basic idea behind PURE operating systems now is to let this association become a configuration matter.

Supporting architecture transparency is a key issue in the design of all PURE building blocks. The creation of a specific architecture then is achieved using *aspect-oriented programming* (AOP) [7]. Aspect programs take care of the manifestation of a particular architecture by describing code transformations that need to be applied to selected components. The transformation process then is performed by an aspect weaver. This way e.g. stubs are generated that hide the style of system-service invocation (e.g. local, remote, crossing address-space boundaries, perform-

[2] Since even pico-kernels can not be the end of the flagpole, should one therefore better try with femto-kernels? And what comes next?

[3] For an understanding of *dual objects*, refer to [12].

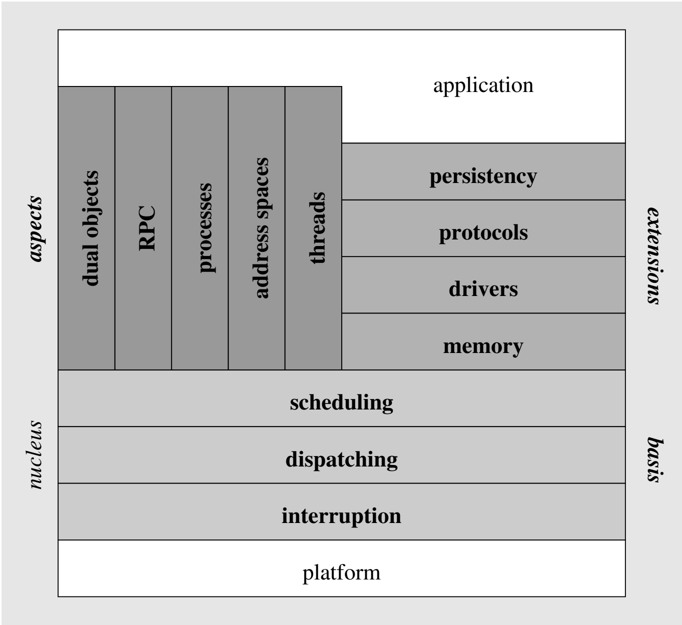

Figure 1: The PURE model of operating systems

ing mode changes, etc.) from the system components: the stubs encapsulate non-functional properties of the respective system components. Furthermore, synchronization primitives are inserted automatically to make e.g. thread-unaware components thread safe. Component instrumentation, e.g. for monitoring purposes, is made feasible as well [11]. Last but not least, to give pattern-based object-oriented designs a final polishing, AOP appears to be a promising technique for streamlining system code [1].

To catch on to the discussion made in the previous section and exemplify the PURE idea a little more, consider e.g. a device-driver component. A device driver in a micro-kernel architecture will typically be implemented as a user-level server process while in a monolithic system the same driver is provided by a set of kernel-level functions. It is not the functionality that differs, but the interaction between the driver and its clients. Using the AOP approach, the driver may be described as a component, or set of C++ classes, that concentrates on its functional properties. In contrast, the interaction is considered to be an aspect that defines a non-functional property of the driver component. It is then up to the aspect weaver to transform a simple method invocation either to a remote procedure call or to a system-call trap.

3 A PURE Case Study

Some of the ideas discussed in the previous section are still future aims. Currently, the PURE development is supported by an aspect-weaver suite to streamline object-oriented C++ programs depending on their actual use pattern [1] and to monitor the run-time behavior of system components [11]. Work on a generic component inter-action system is in progress. The PURE abstractions provided so far allow the composition of deeply embed-able operating systems. The following presentation gives an impression up to which extent the design method followed by PURE is suitable for the development of flexible and yet resource sparing system abstractions.

Family Member	**Size** (in bytes)							
	i686, egcs-1.0.2				c167, gcc-2.7.2.1			
	text	*data*	*bss*	total	*text*	*data*	*bss*	total
interruptedly	812	64	392	**1268**	860	122	2	**984**
reconcile	1882	8	416	**2306**	1792	232	2	**2026**
exclusive	434	0	0	**434**	376	96	10	**482**
cooperative	1588	0	28	**1616**	1762	218	6	**1986**
non-preemptive	1643	0	28	**1671**	1820	218	6	**2044**
preemptive	3786	8	412	**4206**	4856	552	6	**5414**

Table 1: Nucleus memory consumption

The PURE basis (see also Figure 1) is generic, it appears in different configurations. Each of these configurations represents a member of the *nucleus family* and implement a specific operating mode. As a consequence, PURE can be customized, for example, with respect to the following scenarios:

1. One way of operating the CPU is to let PURE run *interruptedly*. This family member merely supports low-level trap/interrupt handling. The nucleus is free of any thread abstraction. It only provides means for attaching/detaching exception handlers to/from CPU exception vectors.

2. In order to *reconcile* the asynchronously initiated actions of an interrupt service routine with the synchronous execution of the interrupted program, a minimal extension to 1. was made. The originating family member ensures a synchronous operation of event handlers in an interrupt-transparent manner [17].

3. The second basic mode of operating the CPU means *exclusive* execution of a single active object. In this situation, the nucleus provides only means for objectification of a single thread. The entire system is under application control, whereby the application is assumed to appear as a specialized active object. There is only a single active object run by the system.

4. A minimal extension to 3. leads to *cooperative* thread scheduling. No other design decisions are made except that threads are implemented as active objects and scheduled entirely on behalf of the application. There may be many active objects run by the system.

5. Adding support for the serialized execution of thread scheduling functions enables the *non-preemptive* processing of active objects in an interrupt-driven context. Thread scheduling still happens cooperatively, however the nucleus is prepared to schedule threads on behalf of application-level interrupt handlers. Actions of global significance, and enabled by interrupt handlers, are assumed to be synchronized properly.

6. Multiplexing the CPU between threads in an interrupt-driven manner establishes the autonomous, *preemptive* execution of active objects. This functional enrichment of the nucleus takes care of timed thread scheduling.

On this basis, a "mechanic" becomes able to compose a tailor-made system according to the functionality required by a given application. Different functionalities also implies different system parameters in terms of, for example, the memory consumption (Table 1) of the resulting configuration. In the realm of deeply embedded systems, memory consumption plays a decisive role.

Another system parameter which is of importance especially for applications that have to operate under real-time constraints concerns scheduling overhead. The family members that support scenarios 1.–3. exhibit no scheduling overhead at all: they simply do not include any scheduler. For the remaining variants, the overhead e.g. on a i686, including context switching, is 49 cycles (4.), 57 cycles (5.), and 300 cycles (6.). By distinguishing system parameters like this in relation to a specific system configuration, one gets a picture on which functionality can be offered to applications and yet fulfill a required real-time guarantee.

These numbers show that a fairly complex object-oriented system implementation [4] must not necessarily result into an overhead-prone system representation. Nevertheless, all is not gold that glitters: the big problem behind the PURE design is the exploding high configuration variety. This problem is going to be tackled using *feature modeling* and specifying Prolog rules by means of which the PURE components are automatically selected from the family or generated out of generic source modules [1]. The rule set ensures that only those components that are really intended to be used by a given application are considered in the configuration process. Conflicting and ambiguous configurations are recognized and handled appropriately.

A further idea is to let the configuration process happen in two main phases. The first phase identifies *scenario adapters* [5] by means of which knowledge about the actually used interfaces are collected. This phase is responsible for the coarse-grain assembly of needed system abstractions. The feature information delivered specifies which interface is being used. More dynamic aspects on how the selected interfaces are used mostly remain open. This problem is going to be solved by the second phase which finally composes an interface implementation that meets the requested requirements.

4 Conclusion and Future Work

A PURE operating system is meant to be an "open operating system". All its abstractions are revealed to a system designer or even application programmer. The entire system is represented as a library, or a set of libraries, of small and "handy" object modules. These modules are small with respect to the number of exported references to functions or variables. This helps, e.g., state-of-the-art binders creating slim-line operating systems that contain only those components used (i.e. referenced) by a given application. Prerequisite however is a highly modular system architecture—and this is achieved by a family-based design and an object-oriented implementation.

PURE has much in common with OSKit [4]. Instead of inventing a new system architecture, PURE provides abstractions that allows one to construct many of those architectures. An operating-system architecture is not prescribed by PURE. Rather, a construction set for the development of operating systems is established. Whether an operating system is monolithic or based, e.g., on micro-kernel technology, is up to the actual "mechanic" who uses PURE elements to create a product according to some blueprint. In order to create a tailor-made operating system, the blueprint comes from the application itself.

Besides undertaking a stepwise enrichment of PURE by UNIX-like system functions as part of student projects, future work also concentrates on providing a family-based Java execution environment. This plan encompasses two sorts of Java-related work: (1) PURE_{Java}, to add minimal Java-Extensions to PURE and (2) Java_{Pure}, that aims at developing a family of streamlined Java virtual machines. The works are done in close cooperation with a major German automobil manufacturer to provide a run-time executive that combines the joint processing of soft real-time (Java-based) telematic services and hard real-time (C/C++/assembler-based) engine control functions.

Portability, scalability, extensibility, and composability are the main keywords behind PURE to aim at the development of application-oriented operating systems. In this sense, the design and implementation of PURE adopts much of [2] to come up with a highly flexible software structure.

[4] To give an idea on the actual fine-grain design of PURE: the OSEK family member is made of about 45 C++ classes arranged in a 14-level (inheritance) hierarchy and offers about 600 methods.

References

[1] D. Beuche, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Streamlining Object-Oriented Software for Deeply Embedded Applications. In *Proceedings of the TOOLS Europe 2000*, Mont Saint-Michel, Saint Malo, France, June 5–8, 2000.

[2] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999. ISBN 0-201-82467-1.

[3] J. Cordsen, T. Garnatz, A. Gerischer, M. D. Gubitoso, U. Haack, M. Sander, and W. Schröder-Preikschat. VOTE for PEACE — Implementation and Performance of a Parallel Operating System. *IEEE Concurrency*, 5(2):16–27, 1997.

[4] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 38–51, Saint-Malo, France, 1997.

[5] A. A. M. Fröhlich and W. Schröder-Preikschat. Tailor-made Operating Systems for Embedded Parallel Applications. In *Proceedings of the 4th International Workshop on Embedded HPC Systems and Applications (EHPC'99)*, number 1586 in Lecture Notes in Computer Science, pages 1361–1373, San Juan, Puerto Rico, April 16 1999. Springer-Verlag. ISBN 3-540-65831-9.

[6] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox PARC, February 1997.

[8] H. Kopetz. Fundamental R&D Issues in Real-Time Distributed Computing. Panel at the 3rd ISORC, Newport Beach, USA, March 16, 2000.

[9] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *ACM Operating Systems Review*, 13(2):3–19, Apr. 1979.

[10] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 237–250, Copper Mountain Resort, Colorado, 1995.

[11] D. Mahrenholz. Aspektorientierte Realisierung eines generischen Systemmonitors. Master's thesis, Otto-von-Guericke-Universität Magdeburg, Germany, Apr. 2000.

[12] J. Nolte and W. Schröder-Preikschat. Dual Objects—An Object Model for Distributed System Programming. In *Proceedings of the Eighth ACM SIGOPS European Workshop, Support for Composing Distributed Applications*, 1998. http://www.acm.org/sigops/EW98/papers.html.

[13] OSEK/VDX Steering Committe. OSEK/VDX Operating System, Oct. 1997. Version 2.0 revision 1.

[14] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[15] QNX Software Systems Ltd. *QNX System Architecture*, 1997. `http://www.qnx.com/`.

[16] 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 9.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, Paderborn, 1998.

[17] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 270–277, Newport Beach, California, March 15–17, 2000. IEEE Computer Society. ISBN 0-7695-0607-0.