The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems*

Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski,
Wolfgang Schröder-Preikschat, Olaf Spinczyk, Ute Spinczyk
University of Magdeburg
Department of Computer Science
Universitätsplatz 2
D-39106 Magdeburg, Germany
{danilo,guer,papajews,wosch,olaf,ute}@ivs.cs.uni-magdeburg.de

Abstract

Deeply embedded systems are forced to operate under extreme resource constraints in terms of memory, CPU, time, and power consumption. Typical examples are automotive systems. Today's limousines can be considered (large scale) distributed systems on wheels. There are cars in daily operation consisting of over 60 networked processors (i. e. μ-controllers). Reserved estimations say that in the near future every car will be equipped with about 20 networked μ-controllers, on average. The complexity of these "decentralized computer architectures" can be managed no longer by the application alone. Dedicated embedded operating systems are required to ensure manageability, adaptability, portability, and yet efficiency of the software. Resource sparing operation under (hard) real-time constraints must be the maxim. This paper discusses the design and implementation of a portable, universal runtime executive, PURE*, for these classes of deeply embedded systems.*

1. Introduction

Embedded systems are becoming more and more important — and they are becoming more and more complex. Getting through daily life without being faced with electronically controlled devices is almost unthinkable. This holds not only for the general consumer market regarding cameras, HIFI, kitchen aids, washing machines etc., but also for other markets such as aircraft or automotive industries. Today's limousines, for example, can be considered (large scale) distributed systems on wheels. There are cars in daily operation consisting of over 60 networked processors (i. e. μ-controllers). Although these systems are quite large with respect to the number of μ-controllers, they are still small with respect to memory size. In the above mentioned case, 1–2 MB of global memory (for the entire embedded distributed system) is not uncommon — and this would already be more than luxurious. Typically, a single μ-controller-based *electronic control unit* (ECU) will be equipped with less than 4 KB of memory.

The complexity of these "decentralized computer architectures" can be managed no longer by the application alone. Dedicated embedded operating systems are required to ensure manageability, adaptability, portability, and yet efficiency of the software. At the same time operating system and application together may not exceed the strongly limited available code size of a few KB.

One example for such a dedicated system is given by the OSEK/VDX standard, which includes the specification of an operating system API, especially trimmed to the automotive sector. With the commitment to this standard, the participating European car manufacturers hope to gain access to a broader range of exchangeable applications and tools. Unfortunately, a single standard API cannot possibly support all types of automotive applications without cutting back either on functionality or on efficiency. For this reason, OSEK/VDX specifies four so called conformance classes which offer more or less mighty (and expensive) subsets of the operating system functionality.

The PURE approach goes even further: Its goal is to offer an operating system not only trimmed to a dedicated application area, but to the application itself. In this sense it competes with the special purpose solutions which can hardly be afforded any longer. The idea is to build a highly configurable system and let the application designer choose the necessary functionality. While PURE is not limited to

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1.

any application area, its main focus lies on deeply embedded systems where a resource-sparing *portable, universal runtime executive* for (real-time) applications is of major concern.

The following sections first discuss related works and then describe the PURE system architecture. This is followed by an analysis of the actual PURE implementation, presenting figures regarding memory consumption and performance. A discussion of open problems and some concluding remarks complete the paper.

2. Related works

The observation that operating systems often seem to pose unnecessary burdens on the applications they should support, is not limited to the area of embedded systems, but can be noticed wherever high efficiency is mandatory.

Traditional operating systems often limit the performance of applications because the strategies they implement (e.g. to schedule threads or handle page faults) are not optimized concerning specific application demands. Extensible operating systems try to solve this problem by allowing applications to safely augment kernel functionality. In the approach followed by SPIN [2], application-level system functions may be added to the kernel. In contrast, the idea of Exokernel [7] is to move as much functionality as possible to user-level. In both systems the number of kernel (i. e. protection boundary) crossings is minimized to further improve performance by introducing an innovative operating system architecture. VINO [19] focuses on the reuse of code by structuring the kernel as a set of reusable, extensible abstractions.

While the above-mentioned extensible operating systems try to achieve a maximum of application orientation, they do not aim at supporting deeply embedded systems. In this area system software is forced to operate under extreme resource constraints regarding the available amount of RAM or ROM. The secure and simultaneous operation of several independent applications is of minor concern: typically, μ-controllers are run in single-user mode. Most embedded systems can neither afford to provide the security mechanisms used by Exokernel and VINO nor is it possible to use a safe language as SPIN assumes.

Other groups are working on operating system construction sets. In this sense, PURE has much in common with the Flux OSKit [9]. OSKit provides a large number of various components which may be assembled to complete operating systems or reused in other environments. It even allows the incorporation of unmodified code from existing operating systems. Thus, both OSKit and PURE share the idea that application orientation and reusability can best be achieved by providing a "box of building blocks". In the case of OSKit, these building blocks necessarily have to be separated, somewhat reducing interaction to a well-defined invocation mechanism. PURE on the other hand uses the concept of building blocks at design and configuration level only. To be able to fulfil the hard requirements concerning run time and code size, an implementation of PURE avoids artificial boundaries: "*It is the system design which is hierarchical, not its implementation*" [10].

The approach followed by PURE is to understand an operating system as a *program family* [15] and to use *object orientation* [20] as the fundamental implementation discipline. In this setting, PURE shares many of the ideas of Choices [4]. But although the name of its successor, μ-Choices [5], might be associated with μ-controllers, it does not aim at deeply embedded systems.

Tigger [3], a framework, or family of object-support systems, targets at a wide range of applications, including embedded (soft-real-time) systems. The main difference to PURE is that Tigger focuses on persistent and distributed objects and their integration into different programming languages, while PURE concentrates on very fine grain customization of kernel functions to fulfil the demands of deeply embedded systems.

An object-based operating system for the area of automotive applications, is ERCOS [1]. Here a clear design and reusability are achieved by structuring the application into many objects of predefined type which communicate with each other via state messages. To be able to meet the hard performance requirements, ERCOS uses off-line tools, that analyze and optimize the source code. For a mostly static application, the operating system may then benefit, for example, from predetermined task schedules or inline-expanded communication primitives.

Other commercial operating systems rely on a modular architecture that allows application developers to configure an operating system according to their needs. For example, VxWorks [8], pSOS [11], and QNX [16] all provide a real-time multitasking (micro) kernel on top of which a number of independent components may be placed. Although these systems try to reduce kernel functionality to a minimum, the kernel itself cannot be tailored with respect to application demands. As a consequence, most of these systems are too large to be used for today's automotive applications.

Last but not least, PURE owes much to PEACE [18], a family-based object-oriented parallel operating system. The main difference to PEACE comes from the radically different operational area, i. e. parallel systems on the one hand and deeply embedded systems on the other. PURE focuses on downward scalability to meet the needs of deeply embedded systems. PEACE laid it focus on upward scalability to efficiently support high-performance parallel computing. Compared to PURE, PEACE relied on a still complex minimal subset of system functions. Although PEACE realizes the concept of a *kernel family*, none of the kernel variants

are suitable for deeply embedded computing. However, the PEACE mechanisms for *remote object invocation* [13] will play an important role for PURE when targeting embedded parallel/distributed systems.

3. System architecture

Instead of inventing a new system architecture, PURE allows one to construct many architectures. A particular operating-system architecture is not prescribed by the design. Rather, an *operating-system construction set* 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 this sense PURE is an open operating system. All its abstractions are revealed to a system designer or even application programmer. To achieve the desired application orientation and to reduce memory and run-time consumption to the lowest, PURE is designed as a *program family*. The goal is to support a dedicated application with a specialized *family member* which provides all necessary functionality but omits any hardware abstractions or operating system features that are not required. This is realized by a highly configurable and very fine-granular structure. Based on a so called "minimal subset of system functions" the system is incrementally enriched by tiny "minimal system extensions". In each step a new minimal basis (i. e. abstract machine) for higher-level system extensions is created. Since extensions are only made on demand and design decisions are postponed as long as possible, a truly application-oriented system evolves. In its last consequence, applications become the final system extensions. The traditional boundary between application and operating system disappears. The operating system extends into the application, and vice versa. As a consequence, applications are not forced to pay for (operating system) resources that will never be used.

But the desired application-orientation or the extreme resource constraints of embedded systems aren't the only reasons that make the family-concept so favorable. The highly modular structure presumably also promotes the application of formal methods to gain clear statements on the correctness and safety of the resulting system complex.

Because the concept of a program family can easily be realized with object orientation, and C++ enables the efficient implementation of a highly modular and extensible system structure [6], the PURE system appears as a (C++) *class library*. Every class implements an abstract data type. Inheritance is employed extensively to build complex abstract data types. Conforming to the family concept, extensions will be made in rather small steps. For example, the thread control block is made of about 45 classes arranged in a 14-level class hierarchy.

The entire system is represented as an archive of small and "handy" object modules. These modules are small with respect to the number of exported references to functions or variables. This helps state-of-the-art binders creating slimline operating systems that contain only those components used (i. e. referenced) by a given application. Prerequisite however is a highly modular system architecture.

The coarse structure of PURE is depicted in Figure 1. PURE is made of a nucleus and nucleus extensions. The

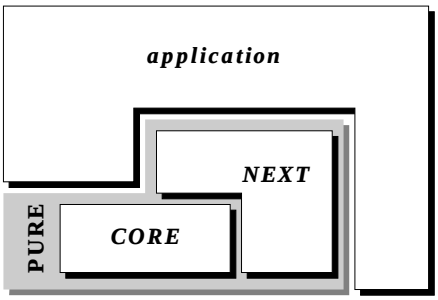

Figure 1. System architecture.

nucleus implements a *concurrent runtime executive*, CORE, for passive and active objects. By means of minimal *nucleus extensions*, NEXT, features such as application-oriented process and address space models, blocking (thread) synchronization, and problem-oriented (remote) message passing are added to the system. These extensions are present only if demanded by the application. They transform the nucleus into a distributed abstract thread processor.

3.1. The nucleus

CORE implements a *minimal subset of system functions* for the scheduling of events and/or actions. Events are implemented by (hardware) interrupts, while actions are implemented by (featherweight) threads. The functions are provided by four building blocks (Figure 2):

1. the *flange*, i. e. abstractions for the attachment of objects to trap/interrupt vectors and propagation of exceptional events to higher levels,

2. the *sluice*, i. e. abstractions for the synchronization of interrupt-driven activities in an interrupt-transparent manner (i. e. without disabling of hardware interrupts),

3. the *signal-box*, i. e. abstractions for the non-blocking synchronization of active objects (i. e. threads), and

4. the *threads reel*, i. e. abstractions for the construction, scheduling, and destruction of active objects.

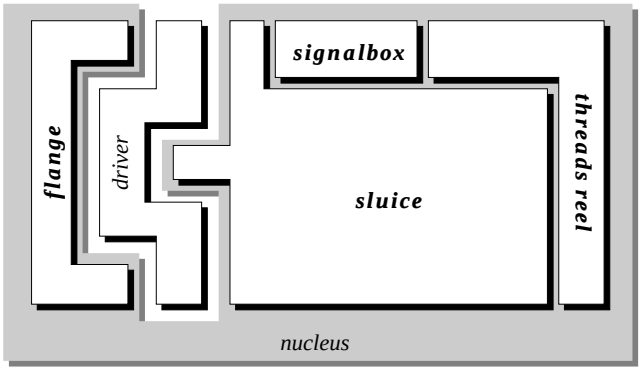

Figure 2. Nucleus building blocks.

Figure 2 shows a fifth building block, the (device) *driver*, which is to be taken into account in a complete system design. However, in PURE, device drivers are not really part of the nucleus, but of NEXT or even the application. Solely, CORE provides mechanisms for a smooth integration of driver modules. In addition to enabling the attachment/detachment of driver abstractions to/from the CPU exception vector entries (supported by flange), functions are provided (by sluice) for the synchronization of interrupt-driven code. In this sense, drivers are considered a glue between non-synchronized and synchronized parts of a PURE-based system.

3.2. The nucleus family

Depending on the actual application requirements, the nucleus building blocks appear in different configurations. Each of these configurations represents a member of the nucleus family. An excerpt from the nucleus family tree is shown in Figure 3. The picture indicates six family members, implementing specific operating modes. Each member is built by one or more function blocks and described by a functional hierarchy of these blocks. The function blocks can be reused in various configurations. In order not to restrict the family, design decisions have been postponed as far as possible and encapsulated by higher-level abstractions. 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 (*interruption*).

2. In order to *reconcile* the asynchronously initiated actions of an *interrupt service routine* (ISR) 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 (*driving*) in an interrupt-transparent manner (*serialization*).

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 (*threading*). There may be many active objects run by the system.

5. Adding support for the serialized execution of thread scheduling functions (*locking*) 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 (*serialization*).

6. Multiplexing the CPU between threads in an interrupt-driven manner establishes the autonomous, *preemptive* execution of active objects. In this case, the nucleus is extended by a device driver module (*multiplexing*) taking care of timed thread scheduling.

Comparing Figure 3 to Figures 1 and 2, the flange takes care of interruption, sluice is responsible for serialization, the threads reel implements locking, threading, and objectification, and NEXT covers (device) driving and (CPU) multiplexing.

A crucial aspect in the design of (embedded) operating systems concerns interrupt handling, in particular the synchronization of asynchronously initiated code fractions. The underlying model has to be efficient and should also reduce the periods of times during which interrupts are physically disabled to an absolute minimum. Ideally, at software level, interrupts should never be disabled at all. Thus PURE ensures *interrupt transparency*: at no time PURE disables (hardware) interrupts or exploits special CPU instructions for synchronization purposes.

In order to safely execute interrupt-driven concurrent code, a PURE-based interrupt handling subsystem, such as a device driver, is to be structured into two parts [17]. The

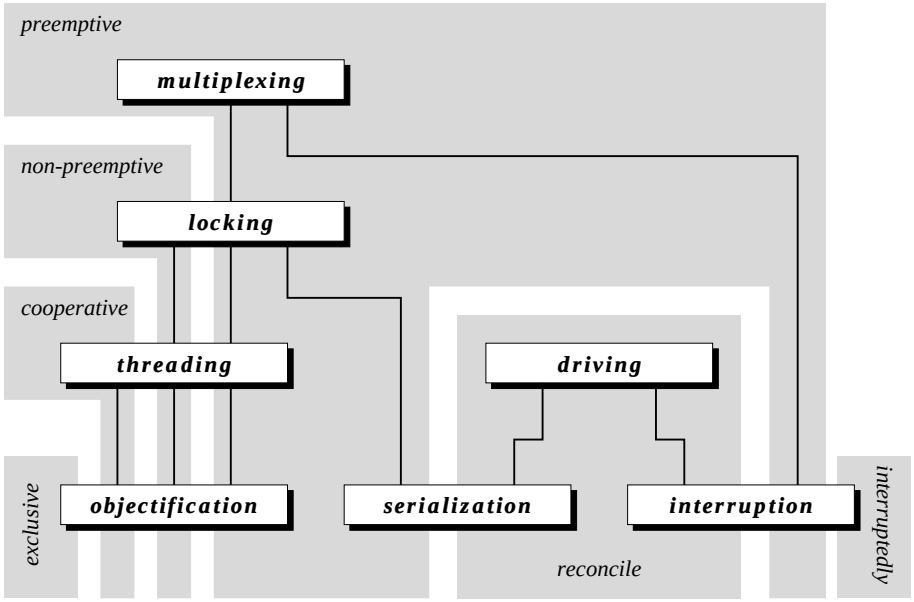

Figure 3. Nucleus family tree.

first part, the *prologue*, is attached to some CPU exception vector and will be immediately started upon signalling a hardware interrupt to the CPU. The second part, the *epilogue*, is considered a prologue continuation and will be started only if critical sections are inactive.

Interrupt synchronization then means *serialization* of event handlers, i. e. epilogues. This is accomplished using a synchronized queue that keeps track of delayed epilogues. As epilogues are considered to contain the less time-crucial part of the handler, a first-come-first-serve strategy is used. A special queue implementation enables the dequeue operation, in order to run delayed epilogues upon leaving a critical section, to be overlapped by enqueue operations issued by an interrupt handler (i. e. prologue). In addition, enqueue operations are allowed to overlap themselves. A high-priority interrupt handler may thus preempt a low-priority one.

Another important aspect is thread scheduling. As with all other operating-system functions, and so with scheduling, PURE does not implement a single solution but a family of solutions. The *scheduler family* currently supports cooperative, non-preemptive, and preemptive processing of threads or bundles of threads. A thread bundle is an ensemble of somewhat related threads that have to be scheduled cooperatively, while the entire bundle is subject to preemptive scheduling. Preemptive scheduling happens event-driven or is based on (static) priorities. Other strategies can be adopted with minimal effort.

3.3. OSEK extension

OSEK is an operating-system standard for automotive systems, specified by the leading European car manufacturers and supply industries. The specification describes four conformance classes. Each of these classes corresponds to specific application requirements and properties of the control system. A conformance class defines a process type, priority allocation, and a process activation pattern. Thus, in a certain sense, OSEK can be seen as an operating-system family consisting of four members.

To show the flexibility of PURE and ease of code reuse that may be achieved with the PURE design and implementation concept we integrated OSEK into PURE as a subfamily. The (minimal) extensions made to the existing PURE abstractions implement the *application programming interface* (API), thread management, and alarm handling as specified by the OSEK standard [14]. The $\text{PURE}_{\text{OSEK}}$ subfamily is made of 61 (C++) classes, 19 of which are dedicated to OSEK and cannot be reused in a non-OSEK environment. The majority of 42 classes, however, adds reusable abstractions to the already existing ones.

The OSEK API increases the applicability of the PURE concepts regarding industrial (or even commercial) environments. Moreover, the resulting system(s) can also be seen as an alternative to a “normal” OSEK implementation because $\text{PURE}_{\text{OSEK}}$ not only adds an object-oriented API, it can also be far better tailored according to the requirements of a dedicated application than conventional OSEK implementations.

4. Analysis

The PURE system is implemented in C++ and runs (as guest level and in native mode) on i80x86-, i860-, alpha-, sparc-, and ppc60x-based platforms. A port to C 167-based, "CANned" μ-controllers is in progress. At the time being, the nucleus consists of over 100 classes exporting over 600 methods.

Thus, PURE is of a really high modular structure. This rises the question of costs in terms of memory consumption and performance. The numbers presented in the following subsections were produced using the C++ compiler `egcs-1.0.2` release for i80x86 processors.

4.1. Memory consumption

As Table 1 shows, the highly modular nucleus structure still results in a small and compact implementation. Basis are the members of the nucleus family tree described along with Figure 3.

family member	**size** (in bytes)			
	text	data	bss	total
prototype				
interruptedly	3256	8	392	3656
reconcile	6406	8	416	6830
exclusive	1289	0	0	1289
cooperative	23988	28	428	24444
non-preemptive	24016	28	428	24472
preemptive	27096	36	428	27560
product				
interruptedly	812	64	392	1268
reconcile	1882	8	416	2306
exclusive	434	0	0	434
cooperative	1620	0	28	1648
non-preemptive	1671	0	28	1699
preemptive	3642	8	428	4062

Table 1. PURE memory consumption.

A PURE system appears in two different shapes, depending on the actual development phase. During prototyping, a conventional, procedure-based system representation founds the basis. In this case, every class serves as compilation/binding unit. As a consequence, every generated object module exports a number of references (i. e. symbols). In order to create a PURE product, a macro-based system representation is used. In this case, class methods are the compilation/binding units and every object module exports only a single reference. Furthermore, the C++ feature of inline-functions is exploited extensively. Even deep class hierarchies then result in flat data abstractions and the number of function calls is reduced to a minimum. Compared to a PURE prototype, memory consumption of a PURE product is about 3–7 times smaller, but system generation time can be the 23-fold.

4.2. Performance

Figure 4 shows the number of (Pentium II) CPU clock cycles spent during interrupt processing. The clock frequency in this experiment was 300 MHz and the measurement was made reading the (on-chip) counter register.

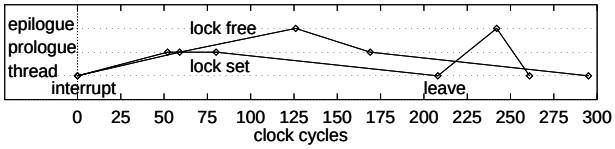

Figure 4. Interrupt latency.

At clock 0, an interrupt occurs. The numbers show that, depending on whether the interrupted thread was inside a critical nucleus section (*lock set*) or not (*lock free*), normal thread execution will be suspended for about 208 resp. 295 clock cycles. Prologue execution starts with clock tick 52 resp. 59. The minor difference is due to the different program instrumentations for the lock-free and lock-set cases. Epilogue execution starts with clock cycle 242 resp. 126. The difference of 116 cycles indicates the best-case delay of an epilogue when the lock is set. Further delays depend on two factors: (1) the time to *leave* the guarded section [1] and (2) the number of preceding epilogues in the queue. Delaying an epilogue (cycles 52–80) takes 28 clock cycles and propagating a delayed epilogue (cycles 208–242) takes 34 clock cycles. Both phases pass through the interrupt-transparent queue synchronization. Resuming an interrupted thread when a prologue terminates (cycles 80–208 and 169–295) costs about 128 clock cycles. This is nearly independent of whether the interrupted thread executed inside or outside a critical section. Leaving an epilogue (cycles 242–261 and 126–169) takes 19 resp. 43 clock cycles. The difference is due to the different actions to be taken when returning from a lock-set or a lock-free case.

Table 2 shows a comparison of the context switch times that result from the different PURE scheduling strategies and the employed nucleus configuration (refer to Figure 3). On basis of a 300 MHz Pentium II, they range from 61 clock

[1] The worst-case execution path of a guarded section has not yet been determined. However, it will be in the order of 368 cycles, which relates to the overhead of timer-driven thread rescheduling.

cycles or 203 ns (*cooperative*) to 368 clock cycles or 1.23 μs (*preemptive*).

scheduling strategy	nucleus configuration	context switch time
FCFS thread	cooperative	61
FCFS bundle	cooperative	78
FCFS thread	preemptive	368

Table 2. PURE context switch times.

4.3. Sample application

Figure 5 shows the pseudo-code of a sample application used to compare different PURE systems including the three major OSEK configurations. The application consists of two threads which synchronize each other by sending and receiving events.

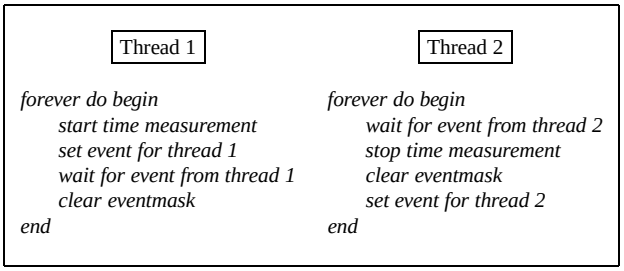

Figure 5. Sample application pseudo-code.

In Table 3 the memory consumption and the thread switch times of the sample application are presented. The numbers were once again measured with the i80x86 native implementation of PURE that was compiled with the `egcs-1.0.2` release, thread switch times are Pentium II clock cycles.

scheduling strategy	code	data	thread switch time
FCFS thread	2871	1052	94
FCFS same bundle	3391	1052	126
FCFS diff. bundle	3371	1052	154
OSEK cooperative	3242	2248	148
OSEK preemptive	3674	2248	202
OSEK mixed preemptive	3922	2248	218

Table 3. Sample application results.

Apart from the scheduling strategy the six scenarios are identical implementations of the sample application depicted in figure 5. All of them use a simple synchronization mechanism based on event masks, which was first introduced into PURE with the integration of the OSEK/VDX standard. So the differences in time and memory consumption are due only to the more or less sophisticated scheduler implementation.

The first three lines of the table contain the numbers for the sample application using two of the already mentioned different native PURE scheduling strategies, FCFS thread and FCFS bundle. Here we differentiate between two possible cases, i. e. whether the threads are in the same bundle or not. The FCFS thread scheduling strategy leads to about 500 bytes less of code compared with the two bundle variants. As expected the simplest scheduling strategy also results in the fastest thread switch.

The lower part of table describes the $\text{PURE}_{\text{OSEK}}$ versions of the test application. Note that the 2248 bytes of data memory contain the two stack memories of the application threads, each having 1024 bytes. So the real memory consumption of the thread control blocks and global scheduler variables etc. is only 200 bytes. Native PURE applications could use the initial boot stack for the first thread leading to only 1024 bytes of stack memory in the data area. The remaining 28 resp. 52 bytes do not contain the thread control blocks. They were allocated dynamically on the first thread's stack and require together 112 bytes. That means that we have to compare 200 bytes for $\text{PURE}_{\text{OSEK}}$ threads with 140 resp. 164 for native PURE threads. The difference is due to some OSEK API features that had to be implemented but are not necessary in every application. The ability to restart the application is an example.

The $\text{PURE}_{\text{OSEK}}$ scheduling strategies need more code than PURE native ones. Features like locking the scheduler, the priority ceiling protocol and multi activation are not for free. All these figures demonstrate the still "featherweight" structure of PURE, although quite a large amount of abstractions (classes, modules, functions) are involved.

5. Open problems

The program family concept helps to develop scalable (system) software. Concerning operating systems, an open design can be achieved that lets applications pay only for the resources they really need. The decision on which abstractions are necessary to bring the hardware close to the application, or vice versa, is made statically at the "office table" and not dynamically during runtime.

However, the main disadvantage of the family concept is that it gets increasingly difficult to select the appropriate components as the family grows. To find the best suited operating system for a given application both functional and non-functional requirements have to be considered. The former corresponds to the demanded capabilities (for exam-

ple whether or not preemption should be provided), while the latter describes issues concerning the implementation.

5.1. System composition

Our approach to cope with the problem of the manifold dependencies between the different family members is to attribute the various PURE abstractions. These attributes describe the capabilities a class/component provides as well as its interface and the requirements it poses on the rest of the system, for example the dependency on other classes. As only some of these attributes may be deduced automatically, others have to be specified manually by the operating system designer.

On basis of the attributed system an application developer may now select the PURE family member which is best suited to the given problem. In a first step, the desired functionality has to be described roughly in terms of module-level attributes. In a second step all modules and classes offering the demanded capabilities, as well as the components they depend on, are selected. While many class-level requirements can be deduced automatically from the demanded module-level capabilities, that description still often will not be sufficient for the unambiguous selection of a single family member. In this case the decision has to be handed over to the application developer. Alternatively a predefined rule may choose a default implementation, preferably obeying certain requirements such as code size or run time optimization.

The chosen approach does not (yet) use formal specifications to prove the correctness of a given selection. It rather defines rules which are applied to find valid class combinations in terms of these rules. There is no guarantee that the combination works as expected. But it is possible to define (via the rules) which combinations are known to work and which not. So the likeliness that later on a problem occurs is reduced as the rules are checked and refined.

5.2. Separation of concerns

Sometimes a configuration decision has global, or at least subsystem-wide, effects. Those decisions mostly deal with non-functional requirements like the selection of an error handling strategy or whether code size or speed optimization of the system is required. The program code that is influenced by that kind of configuration decision might be the complete system. The problem here is that the implementation of specific aspects of a program does not fit into the concept of functional decomposition. Code that is related to a specific (configuration) aspect is tangled with the "normal" code.

This is exactly the problem that aspect-oriented programming (AOP) [12] is about. Following the principle of separation of concerns, the idea of AOP is to separate the so called component code from the aspect code. The aspect code can consist of several aspect programs, each of which implements a specific aspect in a problem-oriented language. Afterwards, an *aspect weaver* takes the component and the aspect code, interprets both, finds join points and weaves all together to form a single entity.

Aspect-oriented programming delivers a direct relationship between the design and implementation entities, i. e. aspects and aspect programs. For a highly configurable system such as PURE, this feature is extremely useful because the process of configuration can be reduced to the simple selection of an aspect program. Without AOP a configuration of this kind of aspects would lead to a code polluted with directives for conditional compilation.

6. Conclusion

Deeply embedded operating systems are designed to specifically support the execution of (control) applications under extreme resource constraints. In consideration of the specific demands of these applications, it becomes extremely difficult, if not impossible, to successfully adopt existing operating systems.

In the realm of embedded systems consisting of several tens of interconnected (autonomous) μ-controllers, the operating-system design is crucial to meet the application demands. A lightweight system structure is required, which eases the adaption and configuration of the offered operating system capabilities to a dedicated application. The boundary where an embedded operating system begins and a runtime environment ends becomes indistinct. Furthermore, the boundary between application and operating system becomes indistinct. Distributed (parallel) embedded systems make the amalgamation of hardware, operating systems, runtime environments, compiler, programming language and application more important than ever.

In addition to providing a highly modular operating system for deeply embedded systems, the PURE project can be seen also as an experiment in the exploitation of object orientation for the implementation of resource-sparing system software. The "old-fashioned" program-family concept has been applied consequently to create featherweight system abstractions. In addition, "standard" development tools have been used for system generation. The results are quite promising—but not yet totally satisfactory.

Future work concentrates on the development of an operating-system workbench. The idea is to offer to system and even application programmers a set of tools that enable the construction, static/dynamic configuration, and (automatic) generation of application-oriented operating systems. An important issue thereby plays aspect-oriented programming. We are currently working on an aspect-weaver

environment that mainly consists of a C++ parser, manipulator, and unparser to enable source to source transformations of C++ code. The idea is to have different aspect weaver plug-ins that shall be able to modify the C++ syntax tree driven by their corresponding aspect program.

References

[1] *ERCOS White Paper*. Schwieberdingen, 1997. `http://www.etas.de/`.

[2] 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 Fifteenth ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain Resort, Colorado, 1995.

[3] V. Cahill, C. Hogan, A. Judge, D. O'Grady, B. Tangney, and P. Taylor. Extensible systems - the Tigger approach. In *Proceedings of the SIGOPS European Workshop*, pages 151–153. ACM SIGOPS, Sept. 1994. Also technical report TCD-CS-94-07, Dept. of Computer Science, Trinity College Dublin.

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

[5] R. H. Campbell and S.-M. Tan. μChoices: An Object-Oriented Multimedia Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HOTOS V)*, Orcas Island, Washington, May 1995. IEEE Computer Society.

[6] J. Cordsen and W. Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (I-WOOOS '91)*, pages 24–28, Palo Alto, CA, October 17–18, 1991.

[7] 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 Fifteenth ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain Resort, Colorado, 1995.

[8] J. Fogelin. The VxWorks real-time kernel. Technical report, WindRiver Systems.

[9] 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.

[10] 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.

[11] Integrated Systems, Sunnyvale, CA. *pSOSystem System Concepts*, 11 1996.

[12] 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.

[13] 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.

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

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

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

[17] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. 1998. Accepted for the *International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*.

[18] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[19] C. Small and M. Seltzer. Structuring the kernel as a toolkit of extensible, reusable components. In *Proceedings of the 1995 International Workshop on Object-Orientation in Operating Systems (IWOOOS '95)*, 1995.

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