A Customizable Library to support Software Synthesis for Embedded Applications and Micro-Kernel Systems*

Carsten Ditze
cadi@uni-paderborn.de
University of Paderborn, Paderborn, Germany

Abstract

Experiences gained from the design of micro-kernel related to either high-performance or hard real-time computing have shown that customization plays a major role to enhance the performance of applications while maintaining a reusable and flexible software architecture. Our goal is to cover both fields by developing a customizable library operating system (DREAMS[1]) intended to be used as a basis for the synthesis of application-specific run-time platforms or operating system kernel.

This paper focuses on a new concept for extremely fine-grained customization of objects and their interrelationships at a source code level and shows how this is applied to the C++ implementation of the DREAMS system to minimize run-time and memory demands. Furthermore, we sketch the architecture of DREAMS and illustrate by example how it is customized to support either a time-driven scheduler for embedded real-time applications or a preemptive round robin scheduler for a multi-threaded micro-kernel.

1 Introduction

For a long time it seemed to be common sense that the objectives of embedded real-time computing (RTC) and those of high-performance computing (HPC) are too contradictory to be handled by an uniform operating system (OS). Although research was independently conducted in both fields, their efforts indicate that a common approach is feasible.

Examples for this observation include the introduction of fine-grained micro- and nano-kernel and approaches related to application-specific kernel customizations and adaptations. In order to maintain a predictable timing behavior, real-time kernel prefer off-line system configuration whereas HPC kernel prefer on dynamic techniques.

Embedded control systems, like motor management in cars, are often used in mass-production. Therefore, minimizing hardware costs is an important factor in the system development stage. A low-cost micro-processor which provides `sufficient` performance to execute the control algorithm at a given sampling rate is preferred opposite to a more expensive high-end processor. The same arguments hold for memory and I/O resources. In general, control systems demand for system software which

- can be run on simple processor architectures,
- does not waste any memory,
- provides application specific services,
- is sufficiently (high) efficient and
- has predictable execution times.

These goals can only be achieved by constructing an execution platform that is dedicated to a single (a class of) application(s). Furthermore, a RTC system design hampers HPC because some overhead must be tolerated in order to ensure predictability (e.g. software priorities, priority inversion protocols, time-divided media access protocols,etc.).

This paper focuses on a new object-oriented approach to enable fine-grained customization at the source-code level without incorporating any overhead and shows how this technique is applied to construct a minimal run-time platform tailored to the needs of an arbitrary application (e.g. an embedded control application, a real-time or high-performance kernel, or even a full-featured OS).

2 Related work

State of the art real-time micro-kernel like QNX^{TM}[12] and RTEMS[14] have got a small memory footprint of about 10K and allow to extend the system at compile time by system modules. Permanent services within the QNX kernel cope with network-wide interprocess communication, process scheduling and interrupt dispatching. RTEMS has got similar kernel concepts and provides a form of dynamic extensions by allowing the programmer

*This work was supported in part by the Deutsche Forschungsgemeinschaft, DFG Sonderforschungsbereich 376 "Massive Parallelität".

[1] Distributed Extensible Application Management System

to bind application-specific functions called on certain kernel events like a context-switch. However, most systems have got a fixed set of kernel services and cannot deal with explicit time constraints. To support extensible and portable real-time applications there are also efforts to develop an embedded real-time Java machine [10]. The Java machine is actually a kernel that provides a fixed set of services (e.g. multiple threads). As mentioned above, our approach is to develop a minimal run-time platform which does not necessarily include multiple threads. However, we can imagine to build a Java machine using the concepts presented in this paper.

Techniques for extensions, adaptations and customizations have become a major subject to recent research in HPC OSs. Choises [2] extends the semantic of polymorphism in C++ by manipulating the virtual function tables to load method and class implementations at run-time. A new re-design of Choices proposes to combine Java and C++ to allow for dynamic customizations[9]. The Exokernel [6] exploits address fault exceptions generated by the processor MMU to trigger insertion of user-level code. Experiences gained from its system design have shown that a fixed interface to a kernel causes immense overhead in the implementation of higher-level services. As a consequence, the Exokernel is only responsible for multiplexing available hardware resources in a secure way among multiple library OSs running on top of it. With respect to embedded real-time system, protection is useful in the implementation and test phase applications, however, it becomes superfluous afterwards because the system is usually not subject to future changes.

On a first view our approach is similar to the design of the Flux OS Toolkit[7] which supports also construction of application-specific run-time platforms based on a library of customizable system services that minimize dependencies to other services. However, we enable extremely fine-grained system customization at the source code level without relying on a kernel architecture which seems to be a new approach. Customizations are used to turn a universal skeleton into concrete so that it represents a minimal execution platform tailored to the needs of the application(s). The objectives of the CHamelEon[8] project are also close to our idea. The project relies on static and dynamic analysis of the application to obtain a *customization profile* . However, we could not yet figure out if the profile is just used to tune an existing kernel or if it can be used to synthesize application-specific execution platforms.

Since we enable customizations at the source code level it is also worth to compare to current research in software synthesis. The Chimera project proposes to build real-time software by interconnecting components that follow the paradigm of *Port-Based Objects*[13]. In contrast to DREAMS the Chimera methodology depends on a non-customizable kernel that supports this type of objects. Ptolemy[11] is able to generate code from multiple programming paradigms like dynamic data-flow. Code generation targets exist for hardware synthesis systems, native assembler or for the $VxWorks^{TM}$ RTOS.

3 Customizations

While studying the state of the art in the fields of HPC OSs and RTOSs, we have observed that there are multiple terms that classify flexible system designs. However, it seems that many of these terms are not yet clearly defined. Our definitions are closely related to those of A. Gopal et al. [8] which appear to be the only ones that differentiate between customizations, adaptations and extensions. A *customizable* system allows applications to specify their (functional and quality) requirements so that a synthesis system can choose the most appropriate components from a library of alternatives to generate a dedicated execution platform. This is synonymous to the term off-line configuration in RTC. Likewise, an *adaptable* system is able to reconfigure itself because either requirements of dynamically created applications must be taken into account or due to requirement changes of already running applications. A system provides *static extensions* if it allows to incorporate new services without editing the source code and *dynamic extensions* if services are added to the running system. As a consequence, extensions are restricted to code integration and do not cope with parameter adaptation/customization.

The DREAMS system is intended to support customization and adaptation. However, this paper deals primarily with customization facilities and the way they are implemented in DREAMS. We argue that customization should take place prior to adaptation in order to decide off-line for which services the overhead due to dynamic extensions is acceptable.

3.1 Classification of techniques

In general, customization is a method to optimize the run-time platform for a given application. It is used to specialize general-purpose system components by exploiting application-specific knowledge. Like code optimization, customization can take place at different levels within the construction process of an application as outlined in Tab. 1. Today, customizations at the application level[2] are excessively supported for many years by most (not to say all) commercial OSs. The majority of recent research OSs discuss various customization and extension techniques to integrate and exchange parts of the OS including its micro-kernel based on compiled object files. Since object files may consist of a single function, this approach enables fine-grained customizations. However, at

[2] Here, an application is regarded as a collection of interconnected programs including their configuration files and environment variables.

Construct. Level	Involved tool	Example subjects of optimization / customization
Source code	Pre-processor	Define constants, addresses and data structures depending on hardware and software package constraints.
Object code	Compiler	Eliminate unreached code fragments, decide about function inlining.
Program	Linker	Decide which object files become part of the executable program.
Application	Make tools	Customizable by environment variables or configuration files.

Table 1: Levels of customization

this level the data structures used by functions cannot be customized. Furthermore, the granularity is still limited. E.g. it is extremely overhead-prone to encapsulate one to five statements in a function just to allow that these become exchangeable.

3.2 The DREAMS approach

In contrast to most other systems, customizations in DREAMS starts at the source code level. Each system component of DREAMS represents a general-purpose template that is specialized according to knowledge about the application and the hardware environment. In principle, each source of run-time and memory overhead within the application and the underlying system software can be eliminated. This is accomplished by heavily exploiting the features provided by standard software development tools. Customizations are specified in a description file using a special configuration language [4]. The file is transformed by an interpreter tool of DREAMS into a set of *#define* statements. Thus, the preprocessor is used to embed knowledge directly into the source code.

E.g. assume the *hasTimer()* function in Fig. 1 (b) evaluates to false because the application programmer has selected a processor architecture that does not support a timer device. An optimizing compiler will detect that the execution path will never reach *getTimer()* or *setHandler()* and eliminate all shaded components in Fig. 1(b). A sophisticated compiler can also deduce that the shaded components in part (a) become superfluous. Hence, it is controlled at the user-level whether dead code eliminations can take place in the compilation phase to remove run-time and memory overhead. In fact, each function called in the figure is customizable so that the installation procedure of the timer interrupt handler is an example for an universal OS service template in DREAMS. Customizations at the source code level affects also the program linkage phase because the linker excludes those functions from the executable that are not actually used, e.g. because all calls have been subject to dead code elimination.

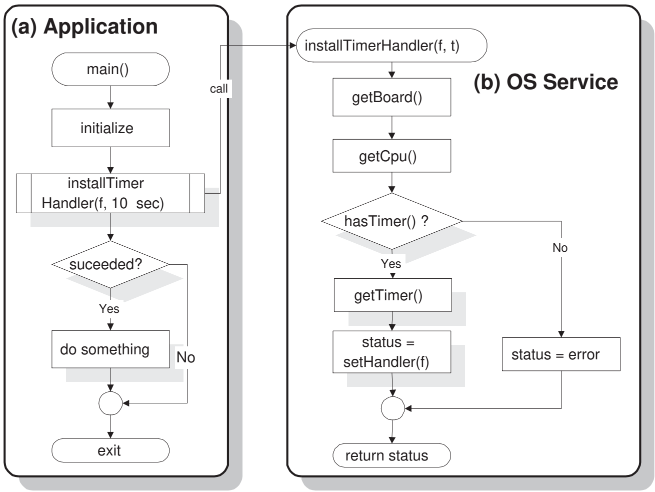

Figure 1: Example: (a) An application uses a timer to schedule the execution of a function f in 10 seconds. (b) Template of a system service to install a timer interrupt handler.

As a conclusion, we propose to base the implementation of any application on a customizable source code library of general services. As outlined above, standard software development tools can also serve as a generator for a highly optimized application-specific run-time platform. For the rest of this paper we always refer to an *application* as a piece of software that may be as well part of some library or OS.

4 Class-level customization

The overall challenge of the DREAMS library OS is to support continuous customization including the application, the system service library and the interfaces to the underlying hardware. Thus, customization must be either supported by the programming language or the language must be pre-processed appropriately by a tool. Today, a new system is only accepted by the community if it supports a broad variety of applications without implying any disadvantages. At the lowest level, the quality of OS software depends pretty much on the programming language that is used for the implementation. The most stringent requirements arise from embedded hard real-time applications:

- Predictable execution times.
- No implied run-time and memory overhead.
- Support simple micro-processor architectures.
- Sophisticated code optimizations.
- Ability to address hardware components directly.

Many object-oriented languages do not fulfill these requirements. E.g. Smalltalk and Java are not predictable and require sophisticated functions provided by a virtual machine (which can be viewed as an OS written in another language). Furthermore, Java includes dynamic memory management and cannot access hardware components directly so that it is not suited for low-level system programming.

According to our knowledge there's no ideal programming language for building OSs. The most closest ones are still C and its superset C++. Both of them fulfill all the requirements mentioned above. We have chosen C++ because object-orientation has proven to be a useful paradigm to describe and implement real-world applications in a natural way.

However, C++ has also some (minor) disadvantages: It suggests to exploit inheritance and polymorphism to specialize classes and methods. Unfortunately, resolving polymorphism takes place at run time and produces overhead. This becomes worse when polymorphism is consequently applied to a strict object-oriented approach to build a fine-grained class library (like for an OS) where the majority of methods consist just of one to five statements.

Another disadvantage is due to hard wired relationships between C++ classes. While developing a C++ program, the programmer has to compose the class structure, the components that form a class and must decide whether they are contained fully or just by a reference. Once a class structure has been designed, it is a tedious job to provide more flexibility by adding a template argument.

The next sections present the fundamentals of our tool-based approach for a fine-grained customization of classes to scale run-time and memory overhead of objects and method invocations. Two goals are pursued by the customizations:

(1) Embed knowledge to the system source code which results in a application-specific definition of the concrete class structure. Note, this implies the definition of the concrete method execution path relying on the techniques outlined in 3.2.

(2) Reduce the disadvantages of C++ and fine-tune the application.

Within the remainder of this section we investigate in more detail the facilities and implementation of customizations guided by a simple application example.

4.1 Scope of customizations

Assume an operating system programmer has to develop a class to represent semaphores. Fig. 2 shows a fraction of an approach that is taken from the DREAMS library

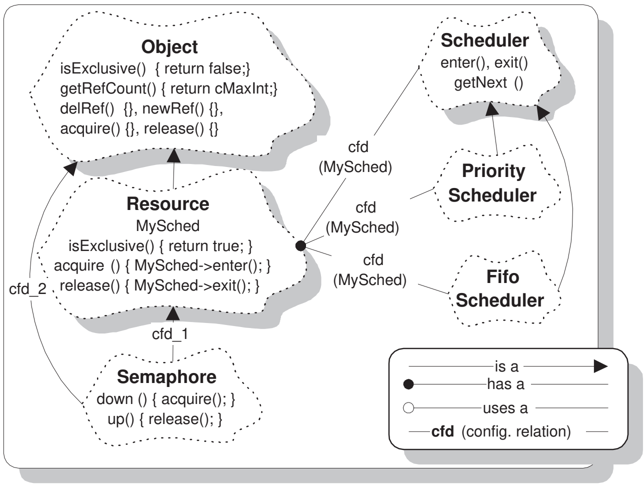

Figure 2: Design of resources in DREAMS. *cfd* denotes a configurable inter-object relation that belongs to *MySched*.

OS[3]. He decided that a *Resource* is a simple *Object* that is exclusive and must have a scheduler that determines which object is allowed to acquire the resource next. However, he should concentrate on the realization of the resource class and defer technical decisions (e.g. which scheduler should be used) to the customization phase.

In general, the following decisions depend on the application and are left to the customization phase:

- Aggregation decisions: Which class fits best the needs for each component? E.g. which scheduler is taken as a default for all resources?
- Inheritance decision: E.g. is a *Semaphore* really a resource? For a single-threaded system it is sufficient if it is regarded as an ordinary object!
- Initialization parameters for each component.

While the former types of customization support application-specific structuring, customizations can also be used for fine-tuning:

- Polymorphism: Is the overhead for virtual methods (in C++) of any value for the application?
- Memory allocation for each component:
  - ❏ Is the allocation policy dynamic? If true, which memory manager is responsible? Otherwise, should it be placed at a special memory location?
  - ❏ Has each object a local instance of the component or do they share a reference to a an external object? E.g.: Are all resources serialized by the same scheduler or by individual ones?

[3] All diagrams are in Booch notation and show only those classes that are essential to understand this paper.

- ❑ When is the memory reserved for this component? Within the constructor of the *Resource* class? On demand?

The design process for a customizable object is sketched in Fig. 3. The programmer declares relationships to be customized within the source code implementation file. For inheritance this is accomplished by subclassing from symbolic class names. Likewise, symbolic member objects (termed *features* in DREAMS) are used to declare a customizable aggregation. These are objects of unknown type and memory requirements. A class is a *skeleton* if it is derived from a symbolic class or if it contains one or more features.

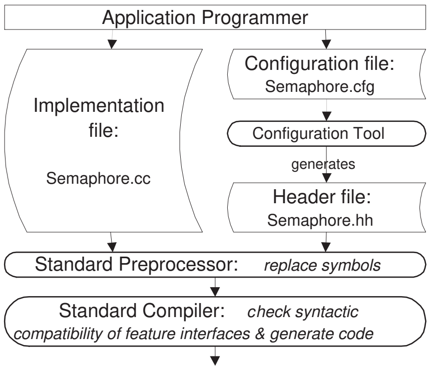

Figure 3: Design flow of customizable objects

Specification of customization decisions as mentioned above are subject to configuration phase of the design process which takes place later on when a skeleton is actually (re-)used by some application programmer. Reusable skeletons are turned into concrete by integrating knowledge about the application, its functional demands and its hardware demands in a separate configuration file that resolves all symbols in the implementation. E.g. the *Semaphore* skeleton is refined by specifying that it is a *Resource* which has a reference to a *FifoScheduler*.

Finally, the standard compiler checks if all required symbols have been defined and if the interface of the selected classes is compatible to the methods that have been applied to the symbols within the implementation file.

4.2 Classes to support customizations

The *Object* class in Fig. 2 is a member of a hardware and OS independent customization support library (CSL). It provides a set of useful attributes and default method implementations that are common to OS objects. The attribute values of objects are implemented as simple inlined *attribute methods* that are either empty or just return a constant value used to control the behavior of services of the DREAMS library. By this way, the state is stored within the object code rather than in member objects which consume memory resources. We preferred this approach because according to our experiences high-quality compilers do provide sophisticated code optimization techniques, but cannot eliminate unreferenced components of data structures. Inlined methods do not hamper the quality of code generation. Each method would consume at least twice the amount of memory and run-time in case it is explicitly represented as a non-inlined function. Therefore, it becomes clear why customizations must be incorporated at the source-code level.

The default values for objects are set so that system overhead is minimized. E.g. for a single-threaded non-preemptive application it is sufficient to regard resources like a *Semaphore* as a simple *Object*. Since the empty *acquire()/release()* methods of *Objects* are used, overhead due to unnecessary resource serialization is avoided. On the other hand, semaphore objects may be customized as *Resources* in case of multi-threading. Now the application is extended by a scheduler feature *MySched* that is accessed upon calls to *acquire()* and *release()*. Like for inheritance, there are also several facets to customize the scheduler feature. The simplest way is to assign a scheduler directly to the *MySched* feature (e.g. *MySched* := *FifoScheduler*). This is convenient if each resource should be serialized independently of all others and if it is acceptable that they all follow the same scheduling strategy.

More flexibility is added at the price of increasing overhead by applying customizations based on more sophisticated members of the CSL (see Fig. 4). The trade-off between flexibility and overhead is organized in different levels:

(1) Direct objects: no overhead

(2) Indirect objects (*ObjPtr*): overhead due to pointer dereferencing

(3) Dynamic objects

(4) Extensible objects

Indirect objects enable sharing of objects so that in this example[4] a group of resources may be managed by a single scheduler. Likewise, a *Resource* may be configured to use an adaptive scheduler which changes its policy according to external events. In that case the *Scheduler* class is configured to provide polymorph (virtual) methods and *MySched* reflects a pointer to the (abstract) *Scheduler*. However, changing an object implementation at run-time involves additional actions. State information must be passed from the old to the new implementation (e.g. the set of ready processes). Due to lack of space we refer to [4] for a closer insight into this technique.

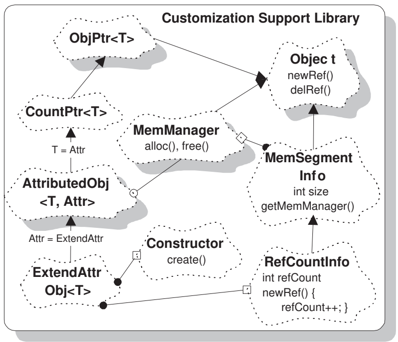

Figure 4: Components of the CSL

[4] Accomplished by *MySched* := *ObjPtr* <*FifoScheduler*>.

4.2.1 Dynamic objects

Dynamic objects are rarely used in the implementation of hard real-time applications with tight deadlines, because dynamic memory management incorporates overhead and the policy used is often hard-wired and not predictable. The CSL provides classes for a new type of dynamic objects which are constructed by individual memory managers. For objects with tight real-time constraints, a manager based on a list of free memory pages that exhibits constant allocation time may be selected whereas an overhead-prone best-fit manager may be preferred for non real-time objects.

The implementation of dynamic objects is based on a more general object type. An attributed object is a template that glues two other objects into a single memory segment allocated by a private memory manager as shown in Fig. 5.

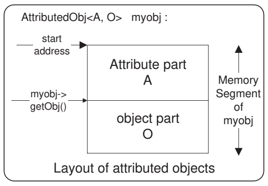

Figure 5: Organization of attributed objects

The attribute part stores additional information about the object part. The minimal information that is available in the attribute part of a dynamic object is its size and a reference to the memory manager that has been used to allocate the memory segment. The latter is required to determine the correct memory manager when the object is destructed. If the attribute part provides also reference counting destruction takes place automatically when the object is no longer used.

The scheduler feature in Fig. 2, may be customized as a dynamic object by *MySched* := *AttributedObj* *<FifoScheduler, RefCountInfo>* [5].

4.2.2 Extensible objects

Many current OSs support techniques to extend the set of functions provided by integrating new system services on demand when they are first called. However, most techniques are based on *software traps* and require complex system services like searching of symbol tables or even a file-system. Although dynamic functional enrichment is a promising approach to adapt an OS to the needs of applications, it seems to be inconvenient for hard real-time systems because the integration is overhead-prone and not predictable. Since DREAMS is intended for HPC and real-time systems, we argue that it must be decided by customization if a particular object/method may be integrated on demand.

The CSL introduces *extensible objects* as a first step towards dynamic code integration. In contrast to dynamic CSL objects which are created where they are defined in the source code, extensible object are created later on when they are first used. Within each dereference operation it is checked whether the object is valid. If not, a *Constructor* object (see Fig. 4) is responsible for allocating memory and initializing the object. The constructor, the memory manager as well as the initialization parameters are subject to customization. We propose to use existing techniques for on demand code integration by exploiting extensible CSL objects.

[5] Attributed objects are especially useful for objects that are customized for communication: Since the memory manager is always known, they may be created by one thread and moved to another that takes over the ownership and becomes responsible for the destruction. For interprocessor communication, the attribute part can be used to add a protocol data unit (PDU) to an arbitrary object which will be send to a remote thread.

5 The DREAMS library OS

The customization facilities for arbitrary objects supported by the CSL classes play the key role to enable synthesis of application-specific run-time platforms that are directly executable on a dedicated processor board. We used the CSL to develop a skeleton for a general OS. Zero-DREAMS is the minimal concrete instantiation of the skeleton and contains only those functions and objects that are mandatory to let a simple application run on some processor architecture. Depending on functional, architectural or quality requirements of the application, further skeletons taken from the DREAMS library can be inserted into the OS skeleton. These in turn may insert either skeletons or concrete classes.

DREAMS is logically structured into multiple layers as shown in Fig. 6 . At the lowest level, the hardware layer is directly accessible from all software layers. By this fact embedded control applications which require computation performance close to the processor limit are not necessarily punished by any system overhead.

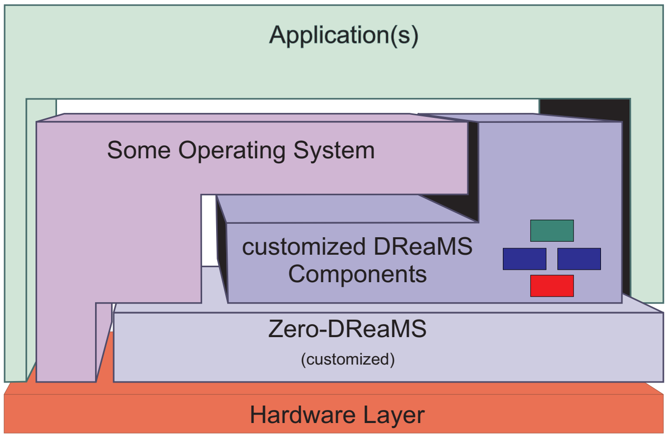

Figure 6: Structure of DREAMS applications

Next, the Zero-Dreams layer represents a tiny and extremely fine-grained skeleton, which performs only indispensable tasks required to initialize the processor and to start the initial application. Depending on the quality of the compiler and on the word size of the processor it consumes about 1-4 Kbytes of memory. The classes of this layer are subdivided into three categories:

- Hardware and OS independent classes of the CSL that build a framework to support software customization as introduced in Section 4.
- A set of customizable classes that define a minimal framework of low-level operating system interfaces which are always required to implement an execution platform (e.g. resource, stack, memory segment, thread, task[6]).
- A framework to encapsulate the minimal *(Von Neumann)* architecture with classes to describe the *Board*, the *Processor* and the *RAM* segment.

As mentioned above, according to the concepts of Dreams[3], a concrete implementation of a micro-kernel is considered as a memory resident application which multiplexes the execution of higher-order applications. We suggest to construct an OS by customizing and assembling classes taken from a library of system services. The Dreams library on the next layer adds optional skeletons for services like *locks, a process table, multi-threading, kernel entry, resource scheduling,* etc. and mechanisms to support dynamic system extension and adaptation to Zero-Dreams if these are required by the application.

This concept is likewise applied to add facilities to the hardware layer: For each hardware component (clock, interrupt controller, etc.) which was developed for some concrete processor board an interface class is a member of the library OS. In case a component is required by the application, it is added to a *Board* skeleton class extending the minimal *Von Neumann* board. However, if the component is not supported by the underlying physical board, one of its emulation members can be selected within the customization phase from the library OS.

Within the remainder of the paper we investigate how skeletons are used to enrich the board and the services of the system by adding skeletons to Zero-Dreams so that the run-time platform becomes able to cope with timer events.

A case study: Stepwise extensions towards a simple time-driven scheduler

The design of a system service that supports time-scheduled functions depends pretty much on the hardware environment and on quality requirements regarding timing precision. As a first step towards a time-driven scheduler, we consider an application that demands for the current time of day. Zero-Dreams does not provide services to manage functions with time constraints so that its skeleton must be customized accordingly (see Fig. 7)

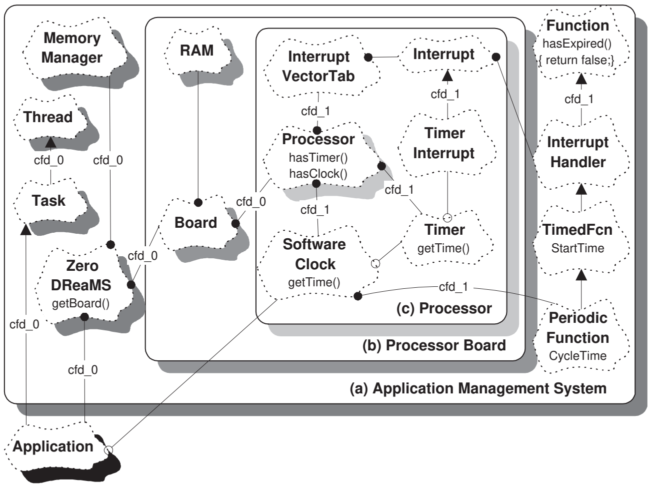

Figure 7: Dreams extensions for clock support: Shaded classes represent classes of Zero-Dreams with customizations denoted by *cfd_0*, unshaded ones have been added by customizations (*cfd_1*) due to clock support.

using the techniques mentioned above: The *Processor* is an embedded skeleton including customizable features for all components that characterize an up to date processor architecture. Assume a concrete processor has a timer device but no clock. Since the application demands for a clock device, the corresponding feature is customized with a software emulation. The *SoftwareClock* installs a private *PeriodicFunction* as a special *InterruptHandler* for the *Timer* which just increments a counter for time units. An interrupt is treated as an asynchronous function call because it borrows the stack of the interrupted thread to save its context. Dreams remains in a single-threaded mode and it is sufficient to save just a minimal context (if at all). In general, overhead is minimized which is for the benefit of HPC applications. Besides, the timing precision is optimized which is of particular interest for RTC applications.

Real-time control applications behave similar to a clock device. Instead of periodically accounting clock ticks, a control algorithm is executed at a predefined rate. However, missing a cycle may result in catastrophic consequences so that an interrupt handler must have a chance to call an emergency handler in that case. In Fig. 7 a handler is a simple *Function* object, thus it has no means of a deadline. Although reactions to missed deadlines are anchored within the source code of interrupt handlers in Dreams, they become subject to dead code elimination because the *hasExpired()* condition evaluates always to false for sim-

[6] The latter are required because even a mono-programmed system has a single task (the application) which is a thread.

ple *Functions*. In order to enable emergency handlers for missed deadlines, each *InterruptHandler* must be viewed (customized) as a *RealTimeFunction* (not shown) which is a specialized subclass of *Function* that has an emergency handler and a deadline attribute and redefines *hasExpired()*.

The goal to provide a time-driven scheduler is scored by replacing the *Timer* in Fig. 7 by a *TimerManager*. While the timer device is restricted to a single handler, the manager multiplexes the timer interrupt and introduces additional virtual timer devices. Real-time issue become much more complicated in this case. If interrupts are enabled while a handler is active the handler can be interfered by others. Otherwise, a deadline is even more likely to be missed due to disabled interrupts. Fortunately, interrupts are expected events in RTC since the behavior of the environment is usually known. Planing events for both cases at the expense of some system overhead (software priorities) is subject to current research (e.g. see [1]).

On the other hand, the DREAMS run-time platform is further extensible to multi-threaded system [4]. That scenario requires to customize the *Processor* as a *Resource* which is serialized by a *Scheduler*. A time-sliced round-robin scheduler is implemented like a clock device: It is comparable to a *"controller"*[7] that is based on a *PeriodicFunction* and performs a context switch when a time-slice *hasExpired()*.

6 Current state and conclusions

We have implemented Zero-DREAMS and the CSL on a PowerPC 604 architecture and on top of Solaris^{TM} which serves as a platform for all developments of the DREAMS system. The minimal customization of Zero-DREAMS for the PowerPC version consumes 2.4 Kbytes of memory (instructions & data). At the moment, the DREAMS library consists of about 250 classes including management of threads, memory and communication. Our long-term objective is to have a synthesize system that generates appropriate platforms using customizations based on static analysis of the application.

The integration of fine-grained customizations to all services is still in progress. We measured the benefit of customizations based on a prototype of DREAMS that supports customizations by specification of system parameters. We observed that up to 60% of run-time overhead for inter-processor communication can be saved by exploiting application-specify knowledge [5] and expect comparable savings regarding memory usage by using the approach presented in this paper.

[7]For the Transputer processor architecture a controller for a scheduler is in fact implemented in firmware.

References

[1] P. Altenbernd. CHaRy: The C-LAB hard real-time system to support mechatronical design. In *Int. Conf. and Workshop on Engineering of Computer Based Systems (ECBS)*, 1997.

[2] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Experiences designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM*, Sept. 1993.

[3] C. Ditze. DReaMS – Concepts of a distributed real-time management system. *Proc. of the 1995 IFIP/IFAC Workshop on Real-Time Programming (WRTP)*, 1995.

[4] C. Ditze. A step towards operating system synthesis. In *Proc. of the 5th Annual Australasian Conf. on Parallel And Real-Time Systems (PART)*. IFIP, IEEE, 1998. (to appear).

[5] C. Ditze and C. Böke. Supporting software synthesis for communication infrastructures for embedded real-time applications. In *Proc. of the 15th IFAC Workshop on Distributed Computer Control Systems, (DCCS), Como, Italy*, Sept. 1998. (to appear).

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating System Principles*, Dec. 1995.

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. of the 16th ACM Symp. on Operating System Principles*. ACM, Oct. 1997.

[8] A. Gopal, N. Islam, B.-H. Lim, and B. Mukherjee. Structuring operating systems using adaptive objects for improving performance. In L.-F. Carbrera and M. Theimer, editors, *Proc. of the 4th Int. Workshop on Object Orientation in Operating Systems, Los Alamitos, CA, USA*. IEEE, 1995.

[9] W. S. Liao, S.-M. Tan, and R. H. Campbell. Fine-grained, dynamic user customization of operating systems. In *Proc. of the 4th Int. Workshop on Object Orientation in Operating Systems, Los Alamitos, CA, USA*, pages 62–66, 1996.

[10] K. Nilsen. Embedded real-time development in the Java language. In *Proc. of Embedded Systems Conf. East*, Boston, 1996.

[11] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for dsp using ptolemy. In *Journal of VLSI Signal Processing*, volume 9, pages 7–21, Jan. 1995.

[12] D. C. Sastry and M. Demirci. The QNX operating system. *IEEE Computer*, 28:75–78, Nov. 1995.

[13] D. B. Stewart and P. Khosla. Chimera methodology: designing dynamically reconfigurable real-time software using port-based objects. In *Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), Piscataway, NJ, USA.*, pages 46–53. IEEE, 1995.

[14] U.S. Army Missile Command, Redstone Arsenal, Alabama 35898-5254. *RTEMS/C Applications User's Guide*, May 1995. [8].

[8]http://lancelot.gcs.redstone.army.mil/ rg4/c/cuser.html