On Minimal Overhead Operating Systems and Aspect-Oriented Programming *

Andreas Gal, Wolfgang Schröder-Preikschat, Olaf Spinczyk
{gal,wosch,olaf}@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. The program family concept can be applied in this domain to implement reusable and highly configurable operating systems. In this paper we present our approach to use aspect-oriented programming to modify and optimize the inter-object relations and communication pathways in an object-oriented family of operating system. The aim of this approach is to produce streamlined application-specific operating systems, minimizing the code size and runtime overhead.

1 Introduction

In the year 2000 8-bit μ-controllers had a market share of 58% of all produced units. Such a μ-controller has typically only a few thousand bytes of memory and its processing power is very limited. These controllers are still heavily present in the area of deeply embedded systems because manufacturing costs dominate the hardware design. To optimally exploit all available hardware resources highly application-specific operating system software is necessary.

Until a couple of years ago *machine-level programming* was believed to be the only practical approach to implement operating systems for this domain. With the rapidly evolving and changing development projects this approach became unfeasible as code-reuse is barely possible on the assembly language level.

This need for code reuse calls for the use of object-oriented methods and languages in embedded system programming. Instead of implementing dedicated

*This work has been partly supported by the German Research Council (DFG), grant no. SCHR 603/1-1 and SCHR 603/2.

operating systems for every development project a common source code base, an operating system family, can be used [11].

To fulfill the resource constraints of deeply embedded systems the operating system family is configured accordingly to the requirements of the specific application. The result of this configuration is a system free of unneeded and unused functionality. Such a family of operating systems is PURE [12]. PURE is an object-oriented operating system targeted on deeply embedded systems and implemented in C++.

When PURE was originally designed the functional hierarchy of the family based design was mapped directly on a class hierarchy using class inheritance. Abstract classes have been avoided as their implementation in C++ through virtual functions has a negative influence on the code efficiency. The configuration was performed at compile-time by removing branches with not needed functionality from the class hierarchy.

While this is a way to achieve minimal code overhead, it causes serious design problems, for example the class number explosion. In figure 1 there are three independent functionalities implemented by class A, B, and C. To represent all possible combinations four derived class are needed. This number increases dramatically if more functionalities and combinations are available. From the design perspective this can be solved by using abstract interfaces. While these abstract interfaces can be efficient for family members, which use many implementations of that interface, they would cause a significant runtime overhead for family members having only a single implementation for the interface. The overhead results from the unnecessary abstract interface, which is implemented using expensive virtual method invocation.

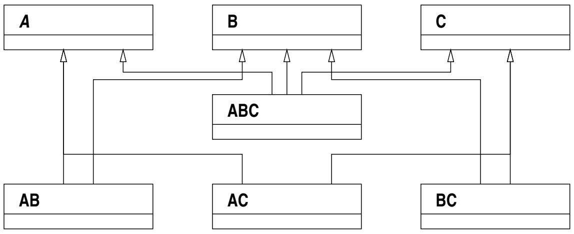

Figure 1: Class explosion

To resolve this conflict between the runtime efficiency and the reusability of the implementation special design and implementation techniques are required. While object-oriented programming is a good mean to deliver highly reusable code, pure object-oriented systems often fail to provide the required minimal overhead. For this reason we exploit the concept of aspect-oriented programming (AOP) [10] to implement an application-specific tailoring of our reusable operating system components. Two examples of this will be presented in the next sections.

Following the principle of separation of concerns the idea of AOP is to separate the component code from so-called aspect code, which is used in PURE to implement the application-specific adaptation. 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 everything together to form a single entity. The advantages of this meta-programming-like approach over conventional techniques for static configuration are that the problem-adequate language allows a higher level of abstraction and that the number of (visible) configuration points can be reduced significantly.

2 Restructuring the Class Hierarchy

The first example of an aspect-oriented implementation technique is the aspect program driven restructuring of PURE components. With this technique the flexibility of pattern-based designs [8] and implementations in the sense of scalability, extensibility, and composability is reconciled with the efficiency needed in the area of deeply embedded systems. The following paragraphs illustrate using a small example component the benefits of our approach. A more detailed discussion can be found in [6].

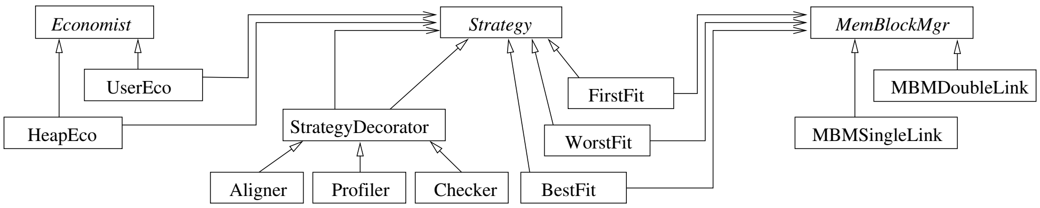

Figure 2: A PURE component for memory management

Figure 2 shows a class diagram of a reusable PURE component, which directly reflects the C++ implementation. The component provides a set of memory management strategies like *first fit* and *best fit*. These strategies can be combined with any sequence of strategy decorator classes to extend their functionality. For example the `Checker` may be plugged in to check a freed block of memory, whether it was really allocated and not already freed. Classes derived from `Strategy` do not know the address and size of the memory they have to manage. This knowledge comes from classes derived from `Economist`. For example the `HeapEco` knows the (platform dependent) address of the global heap used by `new` and `delete`.

The design of this component is based on the well-known *strategy* and *decorator patterns* [8]. Following these patterns the ability to combine classes in a flexible way is achieved by using abstract interface classes (`Economist`, `Strategy`, and `MemBlockMgr`). Object connections could be changed even at runtime and extensions are possible without changing existing classes. No class number explosion takes place.

The problem with this reusable component design is that C++ language features like dynamic binding, which is used excessively in this example, do not come for free. The memory consumption is increased by virtual function tables and the additional code needed to check the object type at runtime and find the relevant function address in the function table. At the same time potential optimizations like function inlining are not possible with virtual functions.

A better memory consumption and runtime behavior could be achieved with special purpose implementations, which are customized for a specific application. As an example consider the relation of the `FirstFit` strategy to the classes derived from `MemBlockMgr`. A `MemBlockMgr` manages a list of either free or allocated blocks of memory. This list can be implemented single or double linked (`MBMSingleLink` vs. `MBMDoubleLink`). Which one an application needs is a trade-off between runtime overhead and code size. If an application designer decides to use the single linked list it does not make sense to compile the other version into the system. Further the abstract base class could be omitted. A special purpose implementation might also move the functionality of `MBMSingleLink` directly into `FirstFit`. In this case a single object would provide the whole functionality, no virtual function table would be required, and unneeded classes would not consume memory.

Compilers are not able to do such optimizations because they do not have access to knowledge about the requirements of the application, when libraries, frameworks, or - in this case - operating systems are translated.

```
require MemoryManagement Application
 {
   aggregate UserEco;                  // UserEco is used as it is
   reference Economist;                // Economist base class is needed
   aggregate FirstFit[MBMSingleLink];  // FirstFit should contain an
                                       // MBMSingleLink instance
   StrategyDecorator(FirstFit);        // the decorator class references
                                       // a first fit object, not any
                                       // strategy
 }
```

Figure 3: Aspect code used to implement the structural aspect of a component

An aspect-oriented implementation solves this problem of efficiency vs. reusability. The class structure of a component is seen as an aspect, which is implemented by a separate aspect program. As an example figure 3 shows an aspect code fragment used to describe a specific component structure by listing names of needed classes and their required relations. The source code of the component and its class diagram become just templates[1], which are used as a starting point for the code transformation that leads to the final application-specific component instance. This technique combined with a static configuration of the aspect programs allows to generate a whole family of component instances from single source code, each optimized for a specific application scenario. This is illustrated in figure 4. While scenario 1 is a subset of the original component with all abstract classes, scenario 4 is completely free of virtual functions.

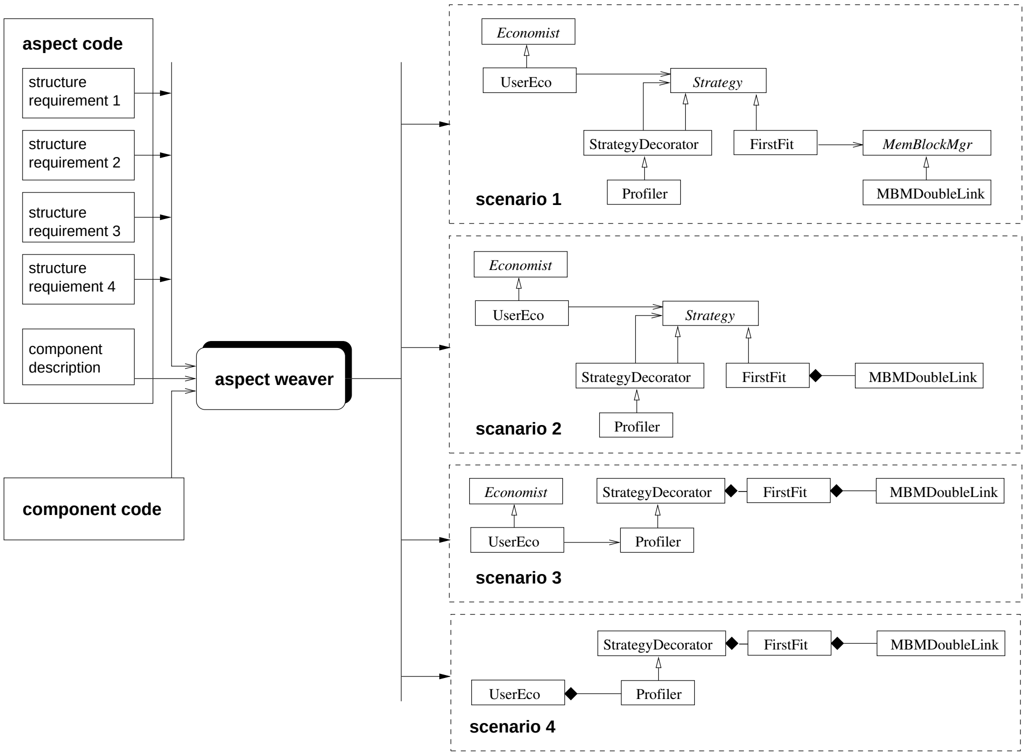

Figure 4: Restructuring of reusable components

As a result of this approach the resources needed by the component now scale with the application requirements. Table 1 presents the code sizes[2] and execution times[3] needed by the four different instances of the memory management compo-

[1] this has nothing to do with C++ template classes and functions

[2] measured in bytes with gcc 2.96

[3] 1000 subsequent allocations and releases of 10 bytes measured in μs on an AMD Athlon 700MHz system running a Linux with no load and in clock cycles for a single allocation and release of 10 bytes on an AVR software simulator.

Sc.	Intel x86						Atmel AVR					
	text	data	total	%	time	%	text	data	total	%	time	%
1	1304	224	1528	100	260	100	2208	372	2580	100	862	100
2	984	116	1100	72	191	73	1784	193	1977	76	663	77
3	705	48	753	49	146	56	1362	51	1427	55	484	56
4	508	0	508	33	78	30	966	0	966	37	318	37

Table 1: Code size and execution times of component instances for different application scenarios

nent shown in figure 4. In the best case this is a reduction down to 33% for the code size and 30% for the execution time.

3 Inline Optimization

As a second example this section presents and aspect oriented approach to implement a global inline optimization strategy.

To limit the impact of function calls and function epiloges/prologes on the overall code size and execution time, C++ supports the inlining of bodies of called functions into the callers code flow. Using the object-oriented design approach results often in large, fine-grained class hierarchies. This also means that even small applications consist of a large number of methods and inline optimization can have a significant effect on the overall code size.

The decision whether to inline a method or not is left to the programmer as the compiler does not have the necessary project-wide knowledge to decide whether a method inlining pays off. Instead, the programmer has to use the '*inline*' keyword. Inlining all method invocations does not necessarily produce the smallest code, as no code can be shared among different callers of the same function. Not choosing any inlining at all produces even larger code sizes, as for example, no register allocation optimization can be performed and the code is bloated with many function prologes and epiloges. Due to the large number of side effects it is very difficult for the programmer to decide which method should be inlined. For a program family these decisions are even harder due to the manifold configuration options and the resulting differences in the code-reuse profile.

Thus it makes sense to separate this concern from the implementation. We understand the inlining problem as an aspect which can be dealt with separately. Instead of scattering the inlining control information throughout the whole project we use code manipulation controlled by an aspect program to explicitly implement a global inlining strategy for each family member.

To find a set of inlining decisions producing a small code size for a certain configuration we use a heuristic inline optimizer based on a genetic algorithm. This would be impossible without the described separation of concerns. The algorithm starts with a random set of different inlining settings and evaluates this population by recompiling the whole project according to the settings of each individual in the population. The best individuals found are intermixed to increase the probability of finding superior successors. It is neither guaranteed to find a globally optimal solution, nor will successive computations of the same problem yield to exactly the same results (figure 5), because the initial population is selected randomly. Always starting with the same population would annihilate the chance to ever reach better results than dictated by an arbitrary selected initial population. Our experiments showed that while the results slightly differ from run to run, in most cases all results stay within an approximately 5 percent window. The set of inlining decisions determined with this algorithm usually results in much smaller code sizes than inlining all methods (figure 5).

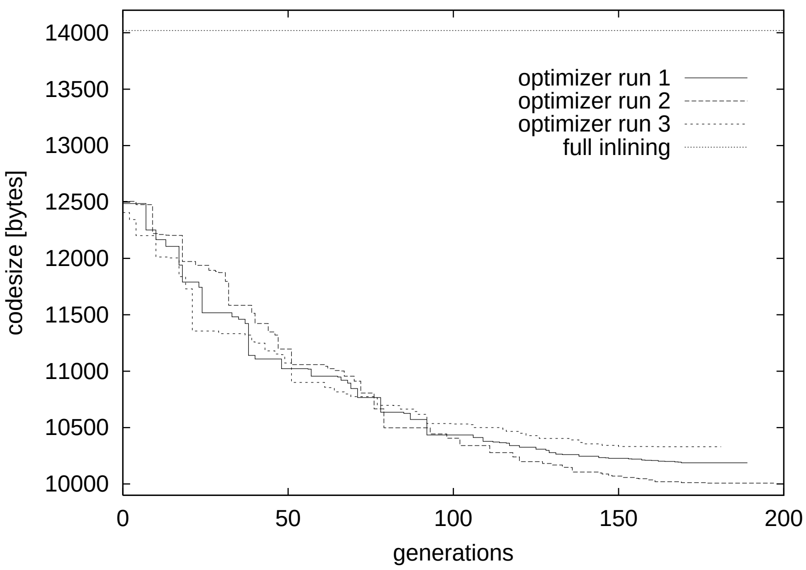

Figure 5: Three different optimization runs of the same project

4 Related Works

Using AOP to optimize software for resource constraint execution domains seems to be quite esoteric in the software engineering community. Most researchers use AOP in combination with languages with inherent efficiency problems like Java [1]. Only two papers are known to the authors, which describe applications of AOP with C++ component code [13] [5]. One reason for this is that tool support for developing aspect weavers for C++ is missing. For this reason our group has developed the PUMA aspect weaver environment [2].

Using global project knowledge for program optimization has also been investigated in Dean [4]. In his thesis he discusses various approaches for improving the generated code from object-oriented languages. The presented approach for inline optimization tries to evaluate the cost of each inline path by comparing inlined and not inlined machine code for that path. The final decision what to inline is done by a global decision-maker within the compiler. Although this approach is much faster at compile-time, the gained benefits in code size are much lower. It also differs from our approach as it relies on integration into the compiler.

The virtual function call elimination process for C++ by Aigner and Hölzle [3] also deploys whole program optimization techniques but uses instrumented source code and a test run of the executable to gather input data for the optimization process. While this paper concentrates on runtime efficiency, it does not include the degree of program structure simplification as our presented approach.

5 Conclusions

The previous sections presented two examples how aspect-oriented implementation techniques can be used to reduce the overhead of object-oriented operating systems like PURE significantly if knowledge about the application(s) is given. This makes AOP an ideal approach for the development of operating system families for the area of deeply embedded systems but also for application development frameworks. Comparisons of the PURE port to the Atmel AVR 8-bit RISC μ-controller with other operating systems used on this platform show that the highly configurable object-oriented PURE requires in many cases less resources than even systems hand-coded in assembly language [7].

Parallel research activities of our group deal with adequate methods to handle the enourmous number of systems configuration available with the presented implementation techniques like feature modelling [9] and application code analysis.

References

[1] The AspectJ Homepage, 2001. `http://www.aspectj.org/`.

[2] The PUMA Project, 2001. `http://ivs.cs.uni-magdeburg.de/~puma/`.

[3] G. Aigner and U. Hölzle. Eliminating Virtual Function Calls in C++ Programs. Technical Report TRCS 95-22, University of California, 1995.

[4] J. Dean. Whole-Program Optimization of Object-Oriented Languages. Technical Report TR-96-11-05, University of Washington, 1996.

[5] L. Dominick. Aspect of Life-Cycle Control in a C++ Framework. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, Lisbon, Portugal, June 1999.

[6] M. Friedrich, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Efficient Object-Oriented Software with Design Patterns. In *Krzysztof Czarnecki, Ulrich W. Eisenecker (Eds.): Generative and Component-Based Software-Engineering. First International Symposium, GCSE'99*, Erfurt, Germany, Sept. 1999. Springer-Verlag. Revised Papers. LNCS 1799.

[7] A. Gal. Reconciliation of an Object-Oriented Runtime Environment and Resource Restricted Systems. Master's thesis, University of Magdeburg, 2001.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[9] K. Kang, S. Cohen, J. Hess, W. Peterson, and S. Peterson. Feature-Oriented Domain Analysis (foda) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

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

[12] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In F. J. Rammig, editor, *Proceedings of the International IFIP WG 9.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, Paderborn, 1998. Kluwer Academic Press. ISBN 0-7923-8614-0.

[13] E. D. Willink and V. B. Muchnick. Weaving a Way Past the C++ One Definition Rule. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, Lisbon, Portugal, June 1999.