Managing Code Complexity in a Portable Microkernel

Uwe Dannowski

System Architecture Group
Universität Karlsruhe

Uwe.Dannowski@ira.uka.de

Abstract

Increasing code complexity can become a serious issue even in a software project as small as a microkernel. This paper reports on how we address this problem in the L4Ka::Pistachio microkernel.

We define multiple configuration dimensions and assign code fragments to the appropriate dimensions. The kernel build system combines code fragments for the specific configuration. While this approach avoids the run-time costs of a full-blown object-oriented design, it does not avoid code duplication.

To address the code duplication problem, we model the code selection with class hierarchies using multiple inheritance and polymorphism. However, the run-time overhead of virtual functions results in a serious (2x) performance hit for the time-critical kernel functionality.

To address this latter problem, we apply class flattening to completely eliminate the overhead of virtual function calls. Our evaluation shows that a kernel with flattened class hierarchies performs as fast as one without class hierarchies. Thus, advanced object-oriented programming techniques need no longer be avoided in performance-focused microkernels.

1 Introduction

Even in a software project as small as a microkernel, code complexity and maintainability often become serious issues. These problems amplify when the kernel needs to support various hardware configurations, i.e., needs to be portable. A configurable source base and an object-oriented (OO) design are promising approaches to tackle the problem. C++ code is portable yet allows sufficient control over data and code layout as it is required in a microkernel.

A successful microkernel must have minimal cache footprint and execution time [6, 7]. Any unnecessary overhead reduces the performance of the system on top of the microkernel. Using OO techniques such as class hierarchies with multiple inheritance and polymorphic calls in a microkernel is prohibitively expensive due to the target address of a virtual function call being resolved at run-time. A common implementation is to store a pointer to a table of function pointers (vtable) in the object, requiring up to two data references and an indirect call for each virtual function call. These costs are usually in the order of tens of processor cycles. For user applications the effective overhead of virtual function calls has been measured to be as high as 40% [2, 5]. This overhead is often aggravated within a microkernel, since the kernel's critical path, the IPC path, is only in the order of 100 instructions long.

In our initial implementation of the L4Ka::Pistachio microkernel [15] we strictly avoided all expensive features of C++ that cause additional run-time overhead: virtual functions, exception handling, and run-time type information. Nevertheless we aimed at well-structured code. We use classes with member functions to implement our kernel objects and the operations on them. Data members are accessed via inline access methods, which does not result in any overhead compared to direct member access. Heavy use of inlining generally reduces run-time overhead despite a fine-granular method structure. Due to aggressive optimization the performance of our implementation rivals that of assembly-only implementations.

With the increasing number of supported target configurations[1] we have identified the need for using more advanced OO features to address problems of reduced maintainability due to excessive code duplication. As we will demonstrate in this paper, we can eliminate the run-time overhead of virtual function calls using class flattening. This enables us to use class hierarchies with multiple inheritance and polymorphic calls without adversely affecting the microkernel's performance.

The remainder of this paper is structured as follows: Section 2 sketches our initial kernel design with multiple configuration dimensions and how it can be represented with a class hierarchy. In Section 3 we describe flattening of C++ classes and present the conditions that enable its use in our case. The evaluation in Section 4 determines the overhead of using virtual functions in a microkernel and shows the effect of class flattening. Section 5 deals with related work and Section 6 concludes.

[1]L4Ka::Pistachio supports nine architectures and 18 platforms.

2 Multidimensional Configuration

To address portability issues, we defined multiple configuration dimensions, most prominent being the kernel API (v4, x0, etc.), the target architecture (ia32, ia64, arm, powerpc, etc.), and the target platform (pc99, ipaq, miata, etc.). We determined whether a certain kernel functionality (implemented in a class member function) needs to be specific for one or more dimensions (e.g., architecture-specific or API-and-architecture specific). We assigned code fragments, namely class member function implementations, to specific values in one (e.g., powerpc) or more dimensions (e.g., v4-ia32) and grouped them into separate files. The build system combines fragments for the target configuration from various files and feeds them to the compiler. This way, we are able to implement the kernel's performance-critical data structure, the thread control block class `tcb_t`, as a simple class.

There are, however, limitations with the above approach. A member function that must be implemented differently for one configuration (e.g., for one architecture while all other architectures can share the same implementation) must be re-implemented in all other configurations as well. Likewise, a data member that is required by only one configuration will be present in all configurations. We have partially addressed the latter problem by introducing dimension-specific sub-structures, i.e., an architecture-specific data member that contains data members only used by that specific architecture. We have also used preprocessor logic to address these problems, but found that it reduces readability of the code. Another minor issue is that source browsing tools (which proved useful for new project members) often get confused when they encounter several implementations of a single member function.

All problems mentioned above can be solved with a class hierarchy. The different configuration dimensions and their code fragments can be modeled with a class hierarchy with mix-ins as illustrated in Figure 1.

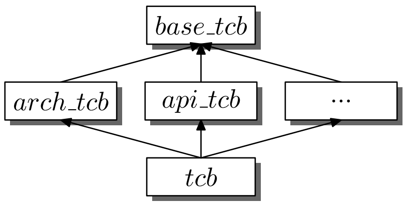

Figure 1. The `tcb` class is constructed from a set of mix-in classes that is determined by the configuration.

The common virtual base class defines the interface between the mix-in classes. Since member functions in mix-ins may need to call member functions implemented in other mix-ins, they need to be virtual functions. Member functions and data members required for their implementation can be logically grouped together. Default implementations can be specialized where necessary. Code duplication and unused data members can be avoided.

It is important to notice that independent of how the class is implemented internally (i.e., as a single flat class or as a class hierarchy), the interface to the class does not change. This minimizes the costs of migrating an existing source base to a class hierarchy, since no code that uses the class needs to be modified.

3 Flattening C++ Classes

We apply class flattening [3] to eliminate the run-time overhead of virtual function calls. The idea is to create a flat class from a whole class hierarchy by moving members of base classes into the most derived class while maintaining semantic equivalence as far as possible. From the class hierarchy we can derive which particular implementation of a virtual function should be moved to the most derived class. In the flattened class, all member functions can be made non-virtual since only one implementation exists that objects of the most derived class would use. The technique eliminates the indirection via the vtable and therefore leads to faster method invocation.

It is important to notice that class flattening can be applied transparently. While the implementation of the most derived class may change significantly, the interface to the class remains unchanged.

A class hierarchy can be flattened with very little effort given the following preconditions:

- The class hierarchy is a valid C++ class hierarchy. We want to maintain the semantics of C++ as programmers know them.
- Outside the class hierarchy, only the most derived class is ever instantiated or referenced by a pointer. No pointers to base classes exist. The internal structure of the class is invisible to the user of the class.
- Data members in the class hierarchy have unique names. Thus, no renaming of data members is required.
- No ambiguities with respect to name resolution exist inside the class hierarchy (that would require a base class specifier to resolve.)

Given that the above preconditions apply, flattening a class hierarchy then involves the following steps:

1. Clone base class members (functions, data) into the target class. Following the class hierarchy in a breadth-first manner beginning at the target class will find the correct implementation of a member function.
2. Clone non-inline base class member definitions and adjust their scope.

3. In cloned members, rename all base class types to the target class.

4. Clone base class constructors into target class functions. Insert calls to these functions into target class constructors.

5. Remove the keyword virtual from all member functions, the base class specifiers from the target class definition, and the base class definitions themselves.

With the aforementioned rules (that are easy to follow when constructing class hierarchies for the microkernel), class flattening does not require semantic analysis. In the microkernel we can leave the decision of which class to flatten to the kernel designer. Thus we can avoid the infrastructure to identify classes that would be candidates for flattening.

Applied to a class hierarchy like `tcb_t` from Section 2, we would expect performance similar to that of a single flat class.

4 Evaluation

In this section we first determine the performance impact of using a class hierarchy with virtual functions in the L4Ka::Pistachio microkernel. We then apply class flattening and compare the resulting performance with the performance when not using a class hierarchy.

We evaluate the kernel's IPC performance with the standard L4 IPC benchmark `pingpong` on a 2.8 GHz Pentium 4 processor. The `pingpong` benchmark sends short messages between two threads and determines the number of processor cycles for a single IPC operation. The numbers presented here are for intra-address space IPC and include hardware costs for entering and leaving the kernel.[2] Benchmarking additional scenarios like cross-address space IPC would report the same absolute overhead since the code paths are identical with respect to the number of virtual function calls.

The *baseline* performance (i.e., the performance when not using a class hierarchy) is shown in Figure 2.

4.1 Overhead of Virtual Function Calls

To determine the performance impact of using virtual functions, we slightly modified the flat class `tcb_t` to turn it into a class hierarchy with a virtual base class and mix-in classes. We moved some performance-critical member functions into the mix-ins. The class hierarchy used for the measurements acts as an early performance indicator only; the complete conversion remains to be done.

The functions we chose to turn into virtual functions are used several times on the critical path. Figure 2 shows an

[2]On a 2.8 GHz Pentium 4 processor, we measured 207 cycles for entering and exiting the kernel using the SYSENTER/SYSEXIT instructions.

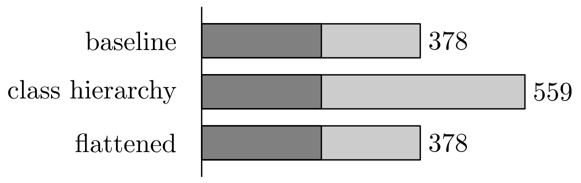

Figure 2. IPC performance in terms of processor cycles (less is better). The dark parts of the bars indicate kernel entry and exit costs. Introducing a class hierarchy adds an overhead of 181 cycles (48%). Applying class flattening to the hierarchy completely eliminates this overhead.

absolute run-time overhead of 181 processor cycles per IPC operation for a kernel that uses a *class hierarchy* to implement `tcb_t`. This amounts to 48% overhead for the IPC scenario we measured. The actual costs per virtual function invocation are irrelevant since we are not interested in determining the exact overhead for a given code path.

4.2 Class Flattening

We flattened the `tcb_t` class using a tool called `collapse` [9]. `collapse` is invoked between the preprocessing and the compilation stage of the build process. The tool implements class flattening as a source-to-source transformation of the preprocessed source and does not require manual interaction.[3] The class to be flattened (`tcb_t`) is specified as a command line option.

The identical performance numbers for baseline and the *flattened* hierarchy in Figure 2 demonstrate that we completely eliminated the run-time costs of virtual functions.

While in the baseline kernel the memory layout of the class `tcb_t` was optimized for minimal data cache usage, it is very likely that the flattened class has a suboptimal layout only. However, we do not observe a performance degradation due to increased cache usage. The low cache footprint of the `pingpong` benchmark lets the involved `tcb_t` objects and the kernel code remain hot in the cache.

5 Related Work

The performance overhead introduced by the powerful mechanisms that C++ and other object-oriented languages have to offer is well-known. The run-time penalty incurred by dynamic dispatch for virtual functions has been discussed in various research papers [2, 5].

Several approaches to reduce the number of virtual function calls in object-oriented programs exist. Class hierarchy analysis [4] inspects call sites to potentially reduce dynamic dispatch to static dispatch by inferring from context

[3]Implementing class flattening as a source-to-source transformation by a stand-alone tool avoids dependencies on a specific compiler and makes it usable in various build environments.

(pointer p will always point to objects of class C.) Profile-based type feedback [1, 12] allows well-predicted run-time type checking followed by static dispatch or inlining. However, the virtual function call is only turned into a likely to be taken, yet conditional call. Sometimes virtual inheritance is unnecessary for a given application and can be turned into normal inheritance after whole-program inspection [2, 13]. Class hierarchy specialization [14, 16] creates a new class hierarchy with reduced object size and potentially devirtualized functions.

Flattening C++ classes is not a new idea [3]. It is superior to aforementioned approaches in that it requires neither class hierarchy analysis nor profiling information. Class flattening has been implemented once in a prototype, yet we miss further work that is based on it. As such, we don't know of any previous use of class flattening to allow object-oriented programming in an OS kernel or in a microkernel in particular.

Flattening can also be seen as an application of aspect-oriented programming in that it applies code transformations to remove the aspect of inheritance. An alternative to introducing class hierarchies with mix-ins would have been to rewrite L4Ka::Pistachio for AspectC++ [11] with configuration dimension specifics formulated as aspects. However, given the size of the L4Ka::Pistachio code base and the frequent use of GCC extensions that the AspectC++ parser cannot handle, this would have been a substantially larger effort than introducing a class hierarchy. Furthermore, it is not clear yet whether AspectC++ is equally suitable for all the cases we can address with our technique.

The Fiasco microkernel [10] is implemented in C++ using class hierarchies and virtual functions, too. However, due to the project's focus on real-time (with performance as a secondary concern) the Fiasco developers have accepted the overhead of virtual functions [8]. Fiasco's use of pointers to base class objects does not allow direct application of class flattening.

6 Conclusion

In this paper we have shown how we manage code complexity in the L4Ka::Pistachio microkernel. We introduced class hierarchies with multiple inheritance and polymorphic calls to replace preprocessor based code generation. We analyzed the costs of using such class hierarchies in a microkernel focusing on performance — something that has previously been considered infeasible. We demonstrated the effectiveness of applying class flattening to the class hierarchy, completely eliminating the run-time overhead of virtual functions. Our result encourages the use of OO design techniques, even in such performance-critical areas as microkernels.

The single flat class remaining from flattening the hierarchy allows for further optimization steps. In future work we plan to apply member reordering in order to optimize cache usage on the kernel's critical path.

References

[1] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *ECOOP '96—Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer, 1996.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96 Conference Proceedings*, pages 324–341, San Jose, CA, October 1996.

[3] Umesh Bellur, Al Villarica, Kevin Shank, Imram Bashir, and Doug Lea. Flattening C++ classes. Technical Report TR-92-23, New York CASE Center, Syracuse NY 13244, August 21 1992.

[4] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[5] Y.-F. Lee and M. J. Serrano. Dynamic measurements of C++ program characteristics. Technical Report ADTI-1995-001, IBM Santa Teresa Laboratory, January 1995.

[6] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[7] J. Liedtke. μ-kernels must and can be small. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 152–155, Seattle, WA, October 1996.

[8] Frank Mehnert. private communication. April 2004.

[9] Jan Oberländer. Applying source code transformation to collapse class hierarchies in C++. Study Thesis, System Architecture Group, University of Karlsruhe, Germany, December 2003.

[10] TU Dresden Operating Systems Group. The Fiasco microkernel. Available from `http://os.inf.tu-dresden.de/fiasco/`.

[11] University of Magdeburg OS Research Group. AspectC++. `http://www.aspectc.org/`.

[12] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav. Compiler optimization of C++ virtual function calls. In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 3–14, Berkeley, CA, June 17–21 1996. USENIX.

[13] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997.

[14] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–332, 1998.

[15] System Architecture Group. The L4Ka::Pistachio microkernel. White paper, Universität Karlsruhe, May 1 2003. Available from `http://l4ka.org/projects/pistachio/`.

[16] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36(12):927–982, 2000.