Automated Object Layout Optimization in a Portable Microkernel

Uwe Dannowski
System Architecture Group, Universität Karlsruhe (TH), Germany

***Abstract*— In a portable microkernel, the increasing diversity of target configurations can lead to software complexity problems. Insufficiencies of current kernel programming techniques manifest in excessive preprocessor use for code selection, in code duplication, and in suboptimal performance. Object-oriented programming can solve the portability problems. However, the language implementation of inheritance often enforces a memory layout of objects that is governed by inheritance relations, not by access patterns, resulting in suboptimal cache usage on the kernel's critical path.**

In this paper we present an automated approach to eliminating inheritance-induced overheads in selected performance-critical data structures. We combine class flattening and profile-guided data member reordering and heavily rely on microkernel characteristics. Evaluation in the L4 microkernel indicates that we can use fine-grained class hierarchies in the kernel at no cost and still optimize for the target system, allowing for portable yet efficient microkernels.

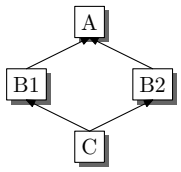
I. Introduction

Microkernels can and must be fast. A successful microkernel must have minimal cache footprint and execution time [11]. Any unnecessary overhead reduces the performance of the system on top of the microkernel. At the same time, microkernels, at core of the system, must be maintainable and portable — traditionally considered a contradiction to the first objective [10].

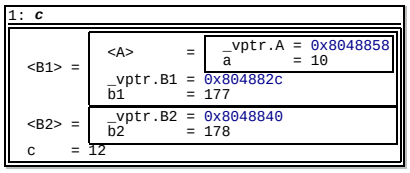
The diversity of target configurations is the root cause of portability problems. Modularity is the key to configurability and thus to portability: Code that is specific to one configuration or a group of configurations must be separated from generic code. Placed in different modules they are combined when building for the particular configuration. We followed this principle in our initial implementation of the L4Ka::Pistachio microkernel [17]. We identified configuration dimensions, such as the architecture, the processor family member, the platform, the amount of parallelism, or even the kernel API, and let the build system choose the appropriate fragments for the particular point in the configuration space. However, we found that insufficiencies of current kernel programming techniques lead to excessive preprocessor use for code selection, code duplication, or suboptimal performance.

Object-oriented programming strongly encourages modularity [4]. With inheritance, classes can be constructed from other classes, enabling fine-grained combination and stepwise refinement of functionality. In earlier work [6] we showed how inheritance can be used to construct classes for kernel objects from configuration-specific super-classes to manage the configuration diversity in the microkernel.

This approach solves problems regarding code duplication and code selection. However, the language implementation of inheritance imposes high run-time overheads that are unacceptable in a microkernel. Whereas in a simple class, the order of data members (or fields) in the class declaration determines the layout of the object [7], the object memory layout of classes using inheritance is governed by the inheritance relationship, and unnamed pointers (vtable pointers) may be added to the object — both in support of dynamic polymorphism. Figure 1 illustrates the object memory representation of a C++ class using inheritance.



(a) A class hierarchy with a virtual base class. Each class contains an `int` data member.



(b) The object layout of C. Addresses increase by four per line. Data members are located in inseparable subobjects. Three vtable pointers are added, inflating the object from 16 to 28 bytes.

Fig. 1. Object memory representation in C++ (gcc)

The superclasses A, $B1$, and $B2$ form subobjects in the object of the inheriting class C, allowing to treat an object of class C as an object of a superclass. To call the appropriate version of a virtual function for an object without statically knowing its exact type, some sort of run-time type identifier is required. The compiler therefore adds a pointer to a table of function pointers, the virtual function table, to each object and invokes the function via a double indirection [16].

Both, the inheritance-dictated placement of data members and the introduction of vtable pointers take away control over the memory layout from the programmer, resulting in poor cache usage on the kernel's critical path. The optimal layout of data structures accessed on the kernel's critical path depends on many factors, such as the architecture [10], [13], the particular choice of algorithms in the kernel, and the workload atop the kernel.

Dynamic polymorphism is, however, not necessary when inheritance is solely used as a tool to efficiently compose classes. This is exactly the case for the way we proposed to construct kernel objects from a set of configuration-specific superclasses. Code using such a class makes no assumptions about how the class was constructed, and inheritance can be

safely removed without changing the interface to the class.

In this paper we present an automated approach to eliminating inheritance-induced overheads in selected performance-critical kernel data structures. We combine class flattening and profile-guided data member reordering, and heavily rely on microkernel characteristics to customize the optimization process. Class flattening removes inheritance, turns virtual functions into normal functions, and prepares the class for data member reordering. Member reordering arranges data members in the class declaration for optimal cache usage. Profiling determines data member access patterns on the kernel's critical path under workload. We integrate flattening, profiling, and member reordering into the kernel build process as illustrated in Figure 2.

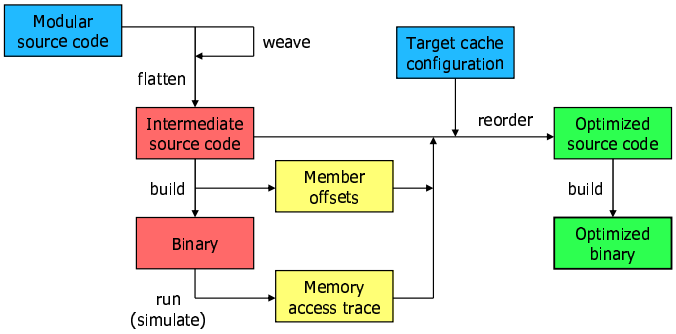


Fig. 2. The optimization embedded into the kernel build process. A kernel with flattened classes is built and profiled to collect access patterns on the critical path. In a second step, members in the flattened classes are reordered for optimal cache usage.

Class flattening and data member reordering are whole program transformations. We implement them as source-to-source transformations that replace the preprocessing stage in the usual per-file preprocess-compile-assemble build process. Not integrating both these transformations into the compiler has the benefit of compiler independence, and does not require a custom-built toolchain for using them.

In previous work [6], we have already demonstrated how class flattening can be successfully used to eliminate the overhead of virtual function calls in a microkernel. Due to space constraints, we will omit a detailed discussion of class flattening in this paper and focus on field reordering and profiling in the context of a portable microkernel.

The remainder of this paper is structured as follows: Section II briefly introduces class flattening. In Section III we discuss strategies for data member reordering in microkernel objects. Section IV presents a profiling approach tailored at extracting access patterns for the critical path from the microkernel. In Section V we evaluate our optimization approach. Section VI discusses related work and Section VII concludes.

II. Transparent Class Flattening

Class flattening produces a single flat class from a class hierarchy by copying all inherited members from base classes into the most derived class and removing the inheritance relationship. Following standard lookup, overriding functions and shadowing data members in a derived class hide their inherited versions, so that shadowed data members and overridden functions become inaccessible and can simply be removed.

Class flattening can be applied as a transparent optimization, such that the code using the class does not need to be modified. The conditions that enable transparent class flattening are discussed in [14].

III. Data Member Reordering

Data member reordering attempts to optimize the memory layout of compound objects (records, structs, classes) according to certain criteria by manipulating the location of data members inside the binary object representation. Type-safe languages abstract from the physical storage layout and leave placement of members to the compiler or run-time system. All layout optimizations are thus automatically valid with respect to program correctness. In contrast, type-unsafe languages expose the locations of members and allow (limited) control of member placement, for example by order of appearance in the compound type declaration. Compound types that are used to represent data structures with a predefined layout such as device registers, hardware-walked tables, API data types, or structured storage in files must not have their data members reordered. However, when code makes no assumption about the internal organization of a type, program correctness is not affected by reordering members. In such cases, data member reordering is transparent to the code using the type and can be applied automatically.

Reordering can even be applied automatically in the presence of programmer-written assembly code that references objects defined in the high-level language, as it is often found in kernels, for example in inline assembly fragments and entry and exit stubs of exception handlers and system calls. Such assembly code can automatically adapt to changing object layouts when it uses symbolic instead of literal offsets to address fields in objects. The respective symbols can be automatically derived from the high-level object definition at build time, appear as constant displacements in the assembly code, and thus will cause no run-time overhead.

Data member reordering maximizes spatial locality of compound data structures larger than a cache block in order to optimize cache behavior. Memory reference traces provide information about a program's memory access behavior. Data members accessed contemporaneously are placed close together to minimize the number of cache blocks used.

The mapping of data members to cache blocks depends on the location of the member in the object and the location of the object relative to the cache block boundaries. Allocating objects at cache block boundaries or at a fixed offset to them allows to minimize the number of cache blocks used. For arbitrarily allocated objects, the worst case cache footprint after reordering is one cache block more than the minimum. The performance gained by aligning objects at cache block boundaries may well make up for the potential waste of memory due to fragmentation. Furthermore, alignment restrictions of data members may already dictate minimum alignment of a

compound object. Also, when an object is known to be aligned at its size (or the next higher power of two), an object's base address can be derived by masking a pointer to an arbitrary location inside the object.

The remainder of this section describes strategies driving data member reordering that have not been considered by previous work. These strategies may lead to higher optimization potential or can simplify the reordering algorithm. Depending on hardware configuration and usage scenarios, not all strategies are necessarily applicable at the same time. Strategies may also, at least partially, contradict each other. It is left to the actual reordering algorithm to choose or prioritize them.

1) Object Roles: Based on the observation that objects of a class show similar access behavior [5], previous work does not distinguish objects of the same class when reordering fields. This certainly holds true for programs that operate on a large number of objects such as nodes in a tree. A microkernel, however, typically manipulates only very few objects during its short, performance-critical operations.

Objects of the same class that are referenced during an operation may actually expose very different access patterns for their fields. In the example shown in Figure 3, all fields of a class worth two cache lines are accessed in a first object, X, whereas only half of the fields (i.e., worth one cache line) are accessed in a second object, Y. Ignoring differences in access characteristics of different objects may result in four cache lines referenced for both objects (Figure 3(a)). The minimum of three cache lines can be achieved by clustering the fields accessed in the second object into one cache line within the cluster of fields accessed in the first object (Figure 3(b)).

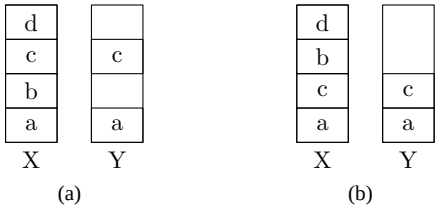


Fig. 3. Two objects, X and Y, of the same class are accessed with different characteristics (a). All fields a, b, c, and d are referenced in X, but only fields a and c are referenced in Y. The cache footprint for accessing both objects can be minimized by field reordering if the objects' access patterns are considered separately (b).

Although all objects of a class share one internal layout, considering access patterns to different objects of the same class separately may yield a higher optimization potential.

2) Field Access Mode: The set of fields of a data structure that are referenced during an operation can be divided into the two subsets of fields that are only read and fields that are written. Fields that are read as well as written belong to the written set.

Assuming a write-back cache, the number of cache lines marked dirty by an operation has no direct influence on this operation's execution time (assuming no self-interference occurs.) Instead, deferred write-back of dirty lines will penalize completely unrelated code. While not beneficial for the current operation, minimizing the number of dirty cache lines may improve overall performance.

A minimum number of dirty lines can be achieved by packing the written fields closely together within the referenced fields and aligning them on a cache line boundary.

3) Field Alignment: Proper alignment of fields can be a matter of performance (penalties due to cache-line splits) or, worse, a matter of correctness (for example, unaligned accesses with LDR and STR instructions on ARM).

However, strict natural alignment of fields is unnecessary as long as all accesses are aligned. For example, a 64-bit integer can safely be 4-byte aligned when it is only ever accessed in 32-bit words. The requirement for natural alignment can be relaxed when the generated code is *known* to be safe, i.e. by configuring the compiler to not use so-called multimedia instructions. Operating systems kernels rarely and microkernels never contain such complex instructions because of the extremely expensive management of the associated hardware state.

Relaxed alignment requirements for large fields increase the flexibility in placing these fields and may simplify the placement algorithm or allow a higher level of optimization.

IV. Determining Field Access Patterns

Reordering fields for optimal cache usage requires precise information about field accesses. The actual code that accesses fields is not very interesting; the memory accesses it generates carry the required information. To drive optimization as described in the previous section, field access information must include the order in which fields are accessed, the access mode (read or write), and the access width.

Profiling the actual kernel with workload on top has a major advantage over analyzing the kernel source code: the programming language (or languages), compiler, optimization level, etc. determine the resulting kernel and its data structures, but they are of no concern for the process of gathering field access information. Also, the set of fields that are accessed on the critical path can be a rather small subset of the fields that can possibly be accessed by the kernel source.

The various methods for analyzing a running program are more or less suitable for recording memory references of kernel code on the critical path: Statistical or event-based sampling easily identifies hot paths, but requires instruction and register analysis to infer the target of a memory operation. Instrumentation provides exact information, but — like sampling — requires substantial infrastructure: Code for logging and extracting the data from the kernel reside in (and pollute) the space the target code runs in. In contrast, an extensible full system simulator can execute the unmodified target kernel and its workload without any infrastructure to the system to be profiled. A simulator extension that collects and exports profiling data is likely to be portable across various simulation targets.

A slowdown of the target system due to run-time overhead of profiling may result in false identification of critical paths.

For example, a network server as workload may experience massive packet losses and behave differently, marking other paths as critical. However, these problems can be side-stepped by replacing the actual workload with a workload simulator causing a representative mix of kernel activities.

A. Microkernel Specifics

Complete system address traces are huge and require significant amounts of time for postprocessing. Often only a few seconds of program execution result in gigabytes of trace data. Field access information can be extracted from a full address trace. However, customized tracing targeted at the specific problem of collecting field access information for field reordering in a microkernel can significantly reduce the amount of trace data and the required processing.

The key to reducing trace data is to aggressively customize the tracing process by incorporating knowledge about the target. Part of this knowledge is inherent in the way microkernels are designed and used, part is available in the kernel source and/or configuration information.

1) Processor Mode: Kernel objects store state information pertaining to API objects or kernel-internal resource management. Kernel objects are accessed by kernel code. Code that accesses kernel objects is executing in the processor's privileged mode. Consequently, for collecting access information to kernel object fields, the tracing facility needs to consider memory accesses only while the processor is executing in privileged mode.

2) Path Length: Microkernel invocations can be thought of as separate, short runs of the program "microkernel", interspersed with executions of user code. Performance-critical system call handlers in a microkernel are rather short, typically in the order of tens or a few hundreds of instructions. With such a limited code path length, a complete trace of one kernel invocation is limited in size, too. For example, all paths taken through the L4Ka::Pistachio microkernel during a run of the pingpong IPC benchmark perform between 2 and 85 accesses to kernel objects.

3) Similarity: Kernel invocations that perform the same operation on different kernel objects produce similar traces. For example, a trace of an IPC system call transferring three words between threads A and B will not differ from a trace for that IPC call between threads C and D, except for the thread identifiers and hence the respective kernel objects being referenced. Short traces with an expected high similarity can be efficiently processed and compressed online instead of generating a complete trace for offline analysis.

4) Number of Objects: Often-called and thus performance-critical microkernel invocations typically reference only very few kernel objects. For example, a simple IPC message transfer between two threads in the L4Ka::Pistachio microkernel involves two, at most three thread control blocks (TCBs). More complex operations involving many kernel objects, such as address space deletion, tend to be invoked less frequently.

5) Address Ranges: The target classes for field reordering are known in advance and so is the size of objects of these classes. Addresses of statically allocated kernel objects are known at kernel build time. Addresses of dynamically allocated objects can only be determined at run-time, but may be easy to track in certain cases. For example, almost all L4 kernels store TCBs in a linear virtual array. At the time of writing, only one L4 kernel [13] allocates thread control blocks dynamically from the kernel heap. However, it then stores their addresses in a statically allocated table. Memory references can be filtered by address range immediately to further analyze only references to objects of target classes.

B. Precise Tracing for Field Reordering

Complete memory reference traces of programs are precise in the sense that they do not omit information. However, they often contain large amounts of useless information. In contrast, field affinity graphs [5] and member transition graphs [9] store only pairwise temporal information about field accesses. Prior research has shown that such pairwise information is theoretically insufficient for finding an optimal field placement [15], and has suggested to keep complete traces when the sequence of memory references is short.

The remainder of this section describes a tracing approach for collecting field access information to drive field reordering for selected target classes in a microkernel. The tracing facility performs aggressive online compression of memory reference trace data to customize tracing by exploiting the microkernel specifics described above. For static customization, the tracing facility uses information from various sources: definitions in the kernel source, addresses from the kernel binary's symbol table, and debug information from the kernel binary. This information is embedded when the tracing facility is built.

The tracing facility produces sequences of field references for different kernel invocations and their frequency of occurrence, whereby invocations that differ only in the addresses of referenced objects are considered identical. These sequences contain all the necessary information for field reordering.

1) Address Filtering and Type Inference: Memory references are filtered by processor privilege mode and address range as discussed in the previous section, so that the tracing facility receives only memory references to kernel objects that are objects of a target class for field reordering. From the address of the memory reference, the type of the object accessed can be inferred. Along with the information about the memory access, the address filter delivers the base address and the type of the referenced object.

Supporting large padding between objects in an array is necessary as this space is often abused. For example, most L4 kernels keep the kernel stack of a thread in the unused part of the memory block (usually 1KB or 2KB) that is allocated for each TCB in the linear virtual array of TCBs.

2) Address Abstraction: The actual addresses of referenced objects are not relevant for field reordering. However, accesses to fields of different objects still need to be tracked separately to allow optimizing for differing field usage patterns.

To distinguish between the kernel objects used during an invocation, the tracing facility assigns sequential object

numbers as different objects are encountered. Objects with different addresses that are used in the same place in similar invocations will be assigned the same object numbers: For example, the first TCB referenced during an IPC operation in the L4Ka::Pistachio microkernel belongs to the target thread of the send phase, while the second TCB referenced belongs to the source. Substituting object numbers for object addresses abstracts from the actual object in favor of an "object role." The number of objects is small so that object addresses can be tracked efficiently.

Memory references are converted to quadruples (n, o, s, m), with n being the number of the distinct object instance encountered since kernel entry (not the actual address of it), o the offset of the reference into that instance, s the access size, and m the access mode (read vs. write.)

3) Per-class Sequences: Using the type information from address filtering, quadruples are recorded in sequences of accesses since the kernel was entered. For every field reordering target class a sequence of references to objects of that class is built.

When exiting the kernel (or on the next entry), the sequence is compared with previously recorded sequences. On a match, a counter associated with the matching sequence is increased. Otherwise, the sequence is added to the list of known sequences with a counter value of one. Runaway sequences of long-running operations in the kernel (idle loop, kernel debugger, etc.) are cut off when reaching an unreasonable length.

4) Sequence Weights: The value of an access sequence's counter in relation to the sum of all counters represents the weight of that access sequence in the profile. Without information about the actual code paths taken, the sequences describe precisely the access patterns to fields in the class and the probability of the pattern during the tracing session. The sequence with the highest weight should be used to determine a new field ordering.

A sequence with a lower weight may be a subset of a sequence with a higher weight in terms of field footprint. That is, optimization goals do not contradict, and optimizing for the latter also optimizes for the former, although potentially not as much as possible. By comparing only the footprint, not the sequence of accesses, inclusion signals a possibility for merging both sequences, thereby increasing the weight of the more frequent sequence.

V. Evaluation

We evaluate data member reordering for kernel objects in the context of the L4Ka::Pistachio microkernel. Our workload is the standard L4 `pingpong` IPC benchmark which sends simple IPC messages back and forth between two threads. The measurement system is a 450MHz Intel Pentium III processor with a cache line size of 32 bytes. The kernel is configured to use the assembly implementation of the IPC path (the so-called "fastpath") whenever it sees fit. To simulate cache pressure from user code, we inserted a WBINVD instruction at the entry point of the IPC path. This instruction writes back dirty cache lines before invalidating all data caches.

We apply automatic field reordering to the `tcb_t` class that stores the kernel state of an L4 thread and is thus used heavily during an IPC operation. Member access patterns for the class are collected in the Simics extensible full system simulator [20] using a custom profiling extension as described in Section IV. The `reorder` tool, sharing its code base with the `collapse` class flattening tool [14], performs field reordering as a source-to-source transformation.

On the fastpath, a kernel with an optimized `tcb_t` class transfers IPC messages 21–25 cycles faster than the original kernel, about the cost of a cache miss on all levels without prior write-back. A cache analysis of the original kernel, shown in Figure 4, supports this: There is an outlier referenced data member in the fifth cache line of the destination TCB. Our field reordering algorithm moves all referenced members to the start of the class declaration and thereby reduces the number of cache lines for the destination TCB from three to two.

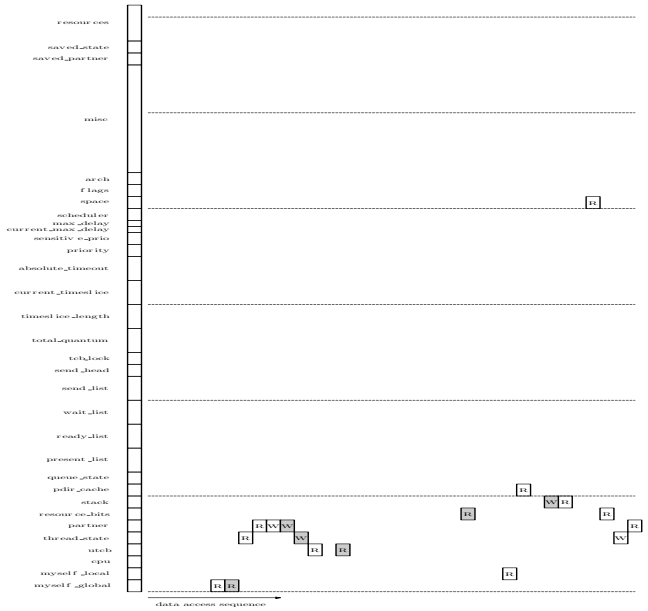


Fig. 4. Accesses to members of the tcb_t class on the IPC system call path. Time progresses in units of memory accesses from left to right. The object layout is shown vertically, with thin horizontal lines marking cache block boundaries. R=read, W=write, gray=source TCB, white=destination TCB.

The memory layout of the `tcb_t` class in the original kernel has been manually optimized by the kernel developers to use as few cache lines as possible. Yet, its layout was not optimal for the particular workload we happened to choose in our evaluation: IPC between threads in different address spaces — the most frequent kernel operation in well-structured microkernel-based systems.

Inheritance will result in a much less compact memory layout of the referenced members and thus in many more cache misses. However, we have not yet completed the transformation of the Pistachio source code to use fine-grained

inheritance to the full extend described in the introduction. For the purpose of evaluation we instead sort data members alphabetically by their name, assuming that we achieve a similarly suboptimal layout as inheritance would produce. For a kernel with alphabetically sorted `tcb_t` data members, we measured IPC times 240 cycles worse than for the original kernel.

Our evaluation shows that field reordering optimizes the memory layout of kernel objects for minimum cache usage. By automatically optimizing for the particular workload, it is superior to manual optimization, which is precluded by using inheritance anyway. By enabling the fastpath in the kernel we showed that our optimization applies not only to C++ code but to assembly code as well.

VI. Related Work

The *flat form* of a class was introduced by Meyer [12] in the context of Eiffel. Meyer sees two uses for the flat form: inspection of the full feature set of a class by a developer, and distribution of a class without its history. Bellur et al. [1] describe a class flattening tool for C++, targeted at eliminating virtual functions calls. An automated approach for variable flattening (replacing the type of a variable with the flattened version of the type) is suggested, but considered infeasible due to the high cost of a full data flow analysis. Bellur et al. also suggest use of class flattening in source code browsing to enhance program understanding, and as a debugging aid, because execution no longer jumps up and down in the class hierarchy. Beyer et al. [2] discuss the impact of inheritance on software metrics like size, coupling, and cohesion. Binder [3] applies class flattening to reduce complexity in the context of software testing. In previous work [6], we used class flattening to completely eliminate the run-time overhead of virtual functions calls in a portable microkernel. In this work, we apply class flattening to produce a flat version of a class whose memory layout can subsequently be optimized by member reordering.

Truong et al. [19] introduced *field reordering* as a technique to improve the cache behavior of dynamically allocated data structures in C. Truong leaves determining the optimal layout to the programmer, because "At present, the automatic detection of the most frequently used fields of a structure is beyond the possibility of current compiler technology." In combination with a second optimization technique, instance interleaving, Truong reports speedups of 1.08–2.53. Chilimbi et al. [5] and Zatloukal et al. [21] describe algorithms for field reordering in C. Based on profiling input and static analysis, a tool produces recommendations for new field orderings that need to be verified and implemented manually. Chilimbi et al. constructs a field affinity graph for every structure type. Nodes represent fields and edge weights are proportional to the frequency of contemporaneous field accesses. Fields with high temporal affinity are placed near each other; no assumption is made about structure alignment on cache-line boundaries, as this "can only be determined at run time". Zatloukal et al. use member transition graphs, where edges represent transition probabilities and cache line survival probabilities. From the graph, cache hit probabilities can be determined for any member ordering. Based on the initial address trace, the new ordering is subjected to a cache simulation to report the reduced cache miss rates. Chilimbi et al. report performance improvements of 2–3% after reordering five of the most frequently used data structures in Microsoft's SQL Server whereas Zatloukal et al. achieved only 1.3% with optimizing seven different structures. By focusing on type-safe programming languages such as Oberon, Kistler and Franz [9] can automate field reordering. They identify memory interleaving and cache line-fill buffer forwarding as source of different latencies for the words in a cache line after a cache miss. Kistler and Franz also discuss optimizing the layout of derived objects. They reorder only the fields introduced in a derived class, because encapsulation often restricts access to inherited fields and thus reduces temporal relations between fields from different levels. Kistler and Franz report combined speedups of 3%–96% for their layout optimizations.

Data packing for a given block size using pairwise information is NP-hard [8], [18]. However, for complete access traces, for example from extremely short sequences that are reused many times, an algorithm can find the optimal layout in exponential time [15]. Algorithms targeted at specific access patterns can be more efficient [22].

Except for type-safe languages, all field reordering approaches merely produce suggestions for a new ordering and require manual checking and implementation. We build on programming conventions, class use restrictions, and programmer's knowledge to automate field reordering for C++, including rewriting the class declaration.

VII. Conclusion

In this paper we describe an automated approach to optimizing the object memory layout of performance-critical classes in a portable microkernel. Inheritance, used to improve kernel portability through modularity, dictates suboptimal object layouts unsuitable for a microkernel's critical path. We combine transparent class flattening and profile-based field reordering to optimize classes composed from many small, configuration-specific classes. We rely on microkernel characteristics to automate and aggressively customize the optimization process. Evaluation indicates that we can eliminate inheritance-related overheads. We enable the use of fine-grained class hierarchies in the kernel at no cost and can automatically optimize for the target system, allowing for portable yet efficient microkernels.

References

[1] 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.

[2] Dirk Beyer, Claus Lewerentz, and Frank Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object oriented systems. In R. Dumke and A. Abran, editors, *Proceedings of the 10th International Workshop on Software Measurement (IWSM 2000): New Approaches in Software Measurement*, LNCS 2006, pages 1–17. Springer-Verlag, Berlin, 2001.

[3] Robert V. Binder. Testing object-oriented systems: a status report. *American Programmer*, 7(4):22–28, April 1994.

[4] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.

[5] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[6] Uwe Dannowski. Managing code complexity in a portable microkernel. In *Proceedings of the ECOOP Workshop on Programming Languages and Operating Systems at ECOOP 2004 (ECOOP-PLOS'04)*, Oslo, Norway, June 2004.

[7] International Organization for Standardization (ISO). *ISO/IEC 14882:1998(E) Programming Languages — C++*, September 1998.

[8] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[9] Thomas Kistler and Michael Franz. The case for dynamic optimization: Improving memory-hierarchy performance by continuously adapting the internal storage layout of heap objects at run-time. Technical Report 99–21, University of California, Irvine, May 1999.

[10] 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.

[11] 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.

[12] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[13] Abi Nourai. A physically-addressed L4 kernel. BE thesis, University of NSW, Sydney 2052, Australia, March 2005.

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

[15] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages (POPL'02)*, Portland, OR, January 2002. Extended abstract.

[16] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceeding of the Spring '87 European Unix Systems User's Group Conference*, pages 189–208, Helsinki, Finland, May 1987.

[17] System Architecture Group. The L4Ka::Pistachio microkernel. White paper, Karlsruhe University (TH), May 1 2003.

[18] Khalid Omar Thabit. *Cache management by the compiler*. PhD thesis, Dept. of Computer Science, Rice University, Houston, TX, 1981.

[19] D. N. Truong, François Bodin, and André Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 322+, October 1998.

[20] Virtutech Inc. Simics — a full system simulator, 1998–2006.

[21] K. Zatloukal, A. Corduneanu, R. E. Ladner, V. Grover, and S. Meacham. Improving cache performance by structure reordering. Extended Abstract, November 1998.

[22] Chengliang Zhang, Yutao Zhong, Mitsunori Ogihara, and Chen Ding. Harness of modeling data locality and a sampling approximate approach. Technical Report TR 877, Computer Science Department, University of Rochester, September 2005.