Towards Fast and Portable Microkernels

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Uwe Dannowski

aus Dresden

Tag der mündlichen Prüfung:	12. Dezember 2007
Erster Gutachter:	Prof. em. Dr. Dr. h.c. Gerhard Goos Universität Fridericiana zu Karlsruhe (TH)
Zweiter Gutachter:	Prof. Dr. Hermann Härtig Technische Universität Dresden

Zusammenfassung

Mikrokerne müssen maximal effizient sein. Als Basis feingliedrig komponentisierter Betriebssysteme stellen sie den Kommunikationsmechanismus zwischen den Komponenten zur Verfügung und spielen damit eine besonders leistungskritische Rolle im Gesamtsystem. Minimale Ausführungszeit und minimale Cachebenutzung des Mikrokerns sind dabei Schlüsselfaktoren. Gleichzeitig sollen Mikrokerne als zentrale Systemkomponente jedoch auch portabel und leicht wartbar sein. Traditionell werden diese Ziele als unvereinbar angesehen, da Mikrokerne für die jeweilige Systemkonfiguration optimiert werden müssen, um ausreichend effizient zu sein.

Aus der hohen Zahl möglicher Systemkonfigurationen eines portablen Mikrokerns ergibt sich ein Komplexitätsproblem. Durch Modularisierung und die damit erreichbare Konfigurierbarkeit kann der Umfang der notwendigen Optimierungen jedoch reduziert werden. Allgemein verwendbare und konfigurationsspezifische Teile des Kerns werden voneinander getrennt, in verschiedenen Modulen platziert und bei der Erzeugung eines Kerns entsprechend der Konfiguration zusammengefügt. Dieses Vorgehen wird bereits erfolgreich im portablen Mikrokern L4Ka::Pistachio angewandt. Durch Unzulänglichkeiten heutiger Programmiertechniken für Mikrokerne — hauptsächlich durch unzureichend feingranulare Konfigurierbarkeit — lassen sich jedoch Probleme wie übermässiger Präprozessoreinsatz und Quelltextduplikation oder, als Alternative, suboptimale Leistung nicht gänzlich vermeiden.

Der Einsatz objektorientierter Programmierung und speziell der Vererbung zum Zwecke der Konfiguration und Komposition von Kerndatenstrukturen ist ein Erfolg versprechender Ansatz, diese Strukturprobleme zu lösen. Konfigurationsspezifische Aspekte der Kernfunktionalität und die dafür benötigten Daten werden in relativ kleinen Klassen gekapselt, die je nach Zielsystem durch Vererbung zu vollständigen Klassen zusammengefügt werden. Jedoch verursacht die flexible Implementierung von Objektorientierung oft einen zusätzlichen Laufzeitaufwand, der in einem Mikrokern nicht tolerierbar ist. Zum einen werden zur Unterstützung dynamischer Polymorphie manche Funktionen durch indirekte und somit nicht vorhersagbare Sprünge realisiert und behindern dadurch eine zügige Ausführung der Instruktionsfolge durch den Prozessor. Zum anderen wird durch die Vererbungshierarchie die interne Struktur von Objekten in einer Weise festgelegt, die eine optimale Cache-Ausnutzung auf dem kritischen Pfad des Mikrokerns verhindert.

Diese Arbeit stellt ein automatisiertes Optimierungsverfahren vor, das es erlaubt, Objektorientierung zur Komposition von Datenstrukturen im Mikrokern einzusetzen, ohne die traditionell damit verbundenen Laufzeitkosten tragen zu müssen. Wissen, das dem Kernprogrammierer bekannt ist, jedoch nicht in geeigneter Weise an den Compiler weitergegeben werden kann, wird dazu verwandt, den Quelltext automatisch so umzuformulieren, dass der Compiler optimalen Code und optimale Datenstrukturen erzeugen kann. Die Transformationsschritte im Einzelnen sind:

1. Das Umwandeln der Vererbungshierarchien ausgesuchter Klassen in einzelne Klassen ohne Vererbung unter Beibehaltung der Schnittstelle der Klassen. Dadurch wird

verhindert, dass der Compiler unnötigerweise Code zur Laufzeitunterstützung für Polymorphie generiert. Da keine Vererbung stattfindet, kann die interne Struktur von Objekten der resultierenden Klasse nun gezielt beeinflusst werden.

2. Das Umordnen der Datenelemente innerhalb der Definition ausgesuchter Klassen, so dass die resultierende Anordnung der Daten innerhalb der Objekte dieser Klassen zu optimaler Cache-Benutzung auf dem kritischen Pfad führt.

Diese Schritte werden — für den Kernprogrammierer transparent — zur Übersetzungszeit vor Aufruf des Compilers durchgeführt. Damit ergibt sich trotz separater Übersetzung der einzelnen Quelltextdateien effektiv eine Optimierung des Gesamtprogramms Mikrokern. Durch die Realisierung der Transformationen auf Quelltextbasis wird kein speziell angepasster Kern-Compiler benötigt, und es wird weitestgehende Unabhängigkeit vom eingesetzten Compiler erreicht.

Bislang war automatisches Umordnen der Felder einer Klasse nur in typsicheren Sprachen gefahrlos möglich. Für Objekte, deren Struktur nicht durch kern-externe Spezifikationen vorgegeben ist, lässt sich die Manipulation der Objektstruktur jedoch auch in typunsicheren Sprachen voll automatisieren, ohne dabei die Korrektheit des Kerns zu gefährden.

Die Entscheidung über die Auswahl der leistungskritischen Klassen im Mikrokern ist unabhängig vom Zielsystem und kann daher statisch erfolgen. Der kritische Pfad und die Zugriffsfolge sind jedoch einsatzabhängig und müssen deshalb während eines Profiling-Laufs bestimmt werden. Dabei kann durch gezielte Ausnutzung von mikrokernspezifischen Eigenschaften eine sehr kompakte und leicht auswertbare Darstellung der Zugriffsinformationen erreicht werden. Beispiele für solche Eigenschaften sind der extrem kurze kritische Pfad sowie die geringe Anzahl der referenzierten Kernobjekte und eine sehr hohe Ähnlichkeit der Zugriffsmuster auf dem kritischen Pfad.

Über die Vermeidung des Laufzeitaufwands der Vererbung hinaus erlaubt das Verfahren, leistungskritische Klassen automatisch für den spezifischen Einsatzfall des Mikrokerns zu optimieren, so dass die notwendigen Anpassungen nicht mehr manuell vom Programmierer vorgenommen werden müssen. Die automatische Transformation des Quelltextes beschränkt sich dabei auf die Definitionen der laufzeitkritischen Klassen. Teile des Kerns, die diese Klassen lediglich benutzen, bleiben somit unangetastet.

Das vorgestellte Verfahren wird exemplarisch auf den L4Ka::Pistachio Mikrokern angewandt und evaluiert. Die Leistung eines Kerns mit Vererbungshierarchie und optimierten Klassen wird der des für eine Architektur handoptimierten Originalkerns ohne Vererbung gegenübergestellt. Dabei zeigt sich, dass das Verfahren die Laufzeitkosten der Vererbung vollständig beseitigt und darüber hinaus bisher ungenutztes Optimierungspotential ausschöpfen kann.

Acknowledgements

Some say I am a man of few words, so I could do away with this section by thanking all who helped making this work happen. Yet, I would like to express my thanks to some people in particular.

I am indebted to Jochen Liedtke, the father of the L4 microkernel, in many ways. He offered me a Ph.D. position in Karlsruhe and made the System Architecture Research Group a fun place to work, learn, and do research. Jochen introduced me to the Sherlock Holmes debugging style and to Dr. Wagner Riesling, and his fantastic cooking let me eat fish again after years of abstinence. His wit made even the toughest disputes enjoyable. Jochen was a visionary, a great researcher, an excellent teacher, and a friend. He inspired me and many other people for years, and he still does.

I would like to thank my advisor Prof. Gerhard Goos for "adopting" me after Jochen passed away in 2001, for his guidance and especially for his patience over the years. Also, I would like to thank Hermann Härtig, my second reviewer, for the confidence he had in me and for freeing a few hours for my defense in his busy sabbatical calendar.

I am thankful to Volkmar Uhlig, Espen Skoglund, Joshua LeVasseur, and Jan Stöß, with whom I enjoyed working on the Pistachio kernel, as well as to Stefan Götz and Andreas Häberlen, for the creative years with many interesting technical discussions. I also thank the students at UNSW/NICTA who ported the kernel to so many other architectures.

Thanks to Sebastian Biemüller for asking many many questions, for proofreading this thesis, and for the irregular quiz nights at Scruffy's Irish Pub. I would like to thank James McCuller for the most professionally managed IT infrastructure ever and for his critical view on many things. I have to thank Frank Bellosa for supporting my research for a year and for keeping an office for me to work in even after I had left university. I would also like to thank AMD, my employer, for providing plenty of distraction during the last year of this work.

Special thanks go to Adelheid, Jochen's wife, for her friendship and support over the years, and for the lovely evenings with long conversations over experimental cooking and excellent wine. I thank Ahmad for his friendship and support, for advice in so many areas but this thesis, and for the supply of excellent coffee and food during my off-university thesis writing hours at Café L'île. I am grateful to my parents for their constant support from my very beginning, for the occasional nudge, and truckloads of chocolate. Last but not least, a very special Thank You to Sinéad for her love and understanding. Being in thesis write-up mode herself she knew very well what I was going through.

Contents

1 Introduction

Microkernels can and must be fast. A successful microkernel must have minimal cache footprint and execution time. Any overhead in the microkernel reduces the performance of the system on top. Early microkernels failed to deliver on the performance promise, so that despite their conceptual superiority microkernel-based systems suffered from poor acceptance outside a small research community. Lessons learned, today's microkernels are designed and often hand-optimized to add as little software overhead as possible to the bare hardware costs of microkernel operations.

Microkernels must also be portable and maintainable. The complexity of the systems built on today's rapidly evolving hardware forbids write-once software. Even a component as small as a microkernel is too complex (i.e., too expensive) to be completely re-written, re-tested and re-verified from scratch for every new piece of hardware.

These two requirements are traditionally considered to be contradictory. Liedtke [35] even argued that microkernels are inherently nonportable and need to be designed and implemented from ground up for every new processor to achieve the necessary performance. Later he accepted that even a microkernel designed for multiple architectures and written mostly in a high-level language can perform sufficiently well. Key to the excellent performance of such kernels was, however, to avoid the powerful but expensive (in terms of run-time overhead) features of the high-level language.

I argue that object-oriented programming techniques such as inheritance for composition can be introduced to a microkernel to improve its portability whereby its initial performance is at least maintained if not even improved.

A microkernel is a rather contained software environment. The kernel typically is configured statically at build time and no additional code is loaded or generated at run time, so that all the code that can possibly be executed is already known at build time. Such a closed environment allows to make simplifying assumptions that enable various optimizations. The internal implementation of the kernel can be changed rather freely provided the kernel's interface and behavior remain constant.

The root cause of the portability problem is the diversity of target configurations. A target configuration traditionally includes one or more external aspects such as the processor architecture, the hardware platform, and the operating system (OS) or run-time environment. A target configuration may also include internal aspects such as different selections of program features or alternative algorithms. Each configuration requires specific handling of its particular features, which is implemented in configuration-specific code. Naively, every new configuration could be implemented in a completely separate code base. However, target configurations naturally have more in common than what they differ in so that configuration-specific code often amounts to only a small fraction of the active code. Large amounts of code can be reused across configurations.

Modularity is the key to reuse and configurability and thus to portability. Code is either specific to one configuration, can be used for a set of configurations, or is generic and can be used for all configurations. Each of these groups contributes a set of modules of which selected modules can be combined accordingly to produce the code base for a particular target configuration. There are three hardnesses of modularity: identifying dependencies of code on configurations so as to minimize code duplication and maximize reuse; finding the right interface for modules such that they can be combined in any required way; and combining modules in an efficient way to achieve acceptable run-time performance.

Object-oriented programming strongly encourages modularity [10] by encapsulating data and functionality in classes with well-defined interfaces. Inheritance allows to compose classes from one or more other classes. New methods and data members can be added and methods and data members inherited from a base class can be overridden, enabling fine-granular combination and stepwise refinement of functionality. In earlier work I showed how inheritance can be used to compose classes for kernel objects from configuration-specific classes to manage the configuration diversity in a portable microkernel written in C++ [15].

Efficient cache usage is of paramount importance to microkernel performance. Besides generating compact and well-scheduled code, frequently accessed data structures must be optimized for minimum cache footprint on the critical path. However, the optimal layout of those data structures heavily depends on many factors such as the processor architecture [35, 40], the particular choice of algorithms in the kernel, and the workload running on top of the kernel. There is no one-fits-all layout. Consequently, performance-critical data structures are often optimized manually for a particular, expected to be common configuration by rearranging the order in which their data members appear in the structure definition. Other configurations either suffer performance losses from suboptimal layout or are considered important enough to justify their own manual layout optimization. The latter then results in multiple configuration-specific definitions of the same data structure in the code base, which reduces the maintainability of the code.

Inheritance addresses many problems with regard to code duplication and code selection. However, the language implementation of inheritance often imposes considerable run-time overhead — mostly in support of dynamic polymorphism — which stems from two sources: inefficient method invocation and suboptimal object layout. Firstly, the implementation of virtual function invocations often uses indirect calls. Such calls are problematic because their target depends on the actual object that the function is invoked on, creating a hard to predict data dependency in the control flow. This overhead can amount to as much as 40 percent in applications [33] and more than 120 percent in a microkernel [15]. Secondly, the object memory layout of classes using inheritance is primarily governed by the inheritance relationship [25], i.e., members of base classes form subobjects in a derived object. Furthermore, meta-data may be added to the object to help finding subobjects and virtual functions. Thus the programmer has only limited control over the resulting objects' memory layout and cannot achieve optimal cache usage by designing the kernel objects properly. Suboptimal data structure layout of kernel objects can result in overheads of 11 percent or more in a microkernel. In this thesis I devise a methodology to eliminate the overheads of inheritance inside a microkernel.

Support for dynamic polymorphism is not necessary when inheritance is solely used as a tool to efficiently compose classes, an approach that perfectly lends itself to managing modularity: A class for a kernel object inherits from a number of small, configuration-specific classes the selection of which is determined for each particular configuration by the kernel configuration framework. By convention, only the most-derived class is instantiated and its objects are never treated as base-class objects. Then the exact type of all that class' objects is known at compile time and various optimizations can be applied.

Class flattening [37] creates a representation of a derived class that directly contains all inherited members. Transparent class flattening is a variant of class flattening that translates the class hierarchy of a derived class into a simple, flat class without visibly changing the class' interface. The definition of the original class is thereby replaced with the definition of the flattened class. Code using a transparently flattened class does not need to be adapted to accommodate for flattening. A compiler will generate code for invoking methods using direct instead of indirect calls and will also not generate any inheritance-related meta-data. I identify the properties of a class hierarchy that make it suitable for transparent class flattening and describe an approach to safely automate the transparent flattening process as a source-to-source transformation in a microkernel environment.

Field reordering [60] changes the placement of fields inside an object to optimize the memory layout of the object according to certain criteria such as cache usage. This reordering can take place at various locations: at runtime, through an indirection mechanism; in the compiler or runtime system of a type-safe language; or before compilation. Applied after class flattening, field reordering can eliminate the overhead of inheritance that relates to suboptimal object memory layout caused by encapsulation of subclass objects. For performance reasons and based on the assumption that kernel usage in a microkernel-based system will not drastically change during execution, only a static reordering at build time is feasible. In this thesis I devise a profiling-based approach to automatically optimize the internal layout of performance-critical data structures in a microkernel.

The optimal layout of kernel objects is derived from access patterns observed while the kernel is executing the envisaged workload. Profiling in an extensible full-system simulator is the least intrusive and most precise method compared to instrumentation and in-target statistical profiling. A profiling extension in the simulator collects information about memory references to data members of performance-critical kernel data structures and provides input to a layout optimizer that produces an optimal ordering of the data members in the data structure definition. To minimize the overhead of profiling, the extension exploits characteristics of microkernels such as execution in privileged mode, extremely short and similar paths through the kernel, and a small number of performance-critical objects being involved in every kernel operation. The optimal ordering is applied automatically and transparently in a source-to-source transformation.

First flattening performance-critical classes and then reordering their members eliminates the run-time overheads of inheritance, so that inheritance can be used to improve the structure of the code base with no negative effect on performance. Both flattening and reordering can be automated and integrated into the kernel build process as transparent optimization steps. Whereas manual layout optimizations of kernel data structures target exactly one configuration and require both time and strong specific kernel expertise, the proposed optimizations apply to any configuration and thus are immediately usable for all kernel programmers.

Prototypically applied to the L4Ka::Pistachio microkernel, the combination of class flattening and field reordering completely eliminates the run-time overhead of inheritance. Performance of a kernel with a performance-critical class implemented as a class hierarchy, flattened, and reordered, matches that of the original kernel with a simple, hand-optimized class. In several configurations the automatic optimization even improves performance, because the hand-optimized class is not optimal for those configurations.

Names and Notations

This thesis discusses optimizations of kernel data structures that are instances of compound data types. Throughout this document these compound data types will be referred to as *classes* and instances of them will be referred to as *objects*, following the naming as used in the C++ specification [25]. The terms *data member* and *field* are synonyms.

Throughout this thesis, the term *x86 architecture* refers to the common subset of the Intel 64 and AMD64 instruction set architectures implemented by processors of Intel and AMD, respectively.

In this thesis, the initialism TCB always refers to the Thread Control Block kernel data structure holding the state of a thread. In the literature this initialism also stands for the trusted computing base of a system. Should this thesis ever refer to the trusted computing base it will do so by using the unabbreviated name.

This work was applied to and evaluated in the L4Ka::Pistachio microkernel, a project of the System Architecture Research group at the University of Karlsruhe, Germany. Throughout this document, the names Pistachio, Pistachio kernel, and Pistachio microkernel shall all refer to said microkernel.

Organization

This thesis is structured as follows: In Chapter 2, I provide background information that helps understanding the remainder of the thesis. Section 2.1 discusses caches and strategies for reducing cache misses and cache miss latencies. Section 2.2 illustrates how an in-memory object representation relates to its type specification, especially in the presence of inheritance. In Section 2.3, I introduce program transformations, in particular class flattening and data member reordering. I also discuss related work in these two areas. Section 2.4 presents portability considerations for kernel code.

Chapter 3 introduces the Pistachio microkernel and its portable application programming interface (Section 3.1) and describes how it addresses portability at the source code level (Section 3.2.) I propose inheritance as a solution to structural problems in Section 3.3 and provide experimentally determined estimates of how severely the introduction of inheritance will reduce the performance of the kernel in Section 3.4.

In Chapter 4, I detail my approach to eliminating the overheads of portability. Section 4.2 describes transparent class flattening to enable data member reordering for complex class hierarchies. Section 4.3 discusses novel strategies for data member reordering that are motivated by observations on object usage in a microkernel. In Section 4.4, I present a profiling approach to determine data member access patterns that is aggressively optimized to leverage microkernel specifics. I give an example of a data member reordering algorithm in Section 4.5. Section 4.6 illustrates how transparent class flattening and field reordering can be seamlessly integrated into the kernel build process.

I evaluate my approach in Chapter 5. Section 5.1 describes the evaluation environment, including hardware and software tools. In Section 5.2, I show by means of a cache analysis and several microbenchmarks that the proposed optimization technique can completely eliminate the overheads of inheritance and beyond that can optimize object layouts for a particular configuration and workload. Section 5.3 provides an insight into the costs of the optimization, and Section 5.4 compares automated optimization to a manual approach.

Finally, Chapter 6 concludes my thesis with a summary of its contributions and gives directions for future work.

2 Background and Related Work

This chapter provides background information for the remainder of the thesis and discusses related work in the following areas:

- I optimize the cache footprint and thus execution performance of operations on data structures. In Section 2.1, I describe cache architectures, cache misses, and strategies for reducing or hiding cache latency.

- I optimize the memory layout of data structures by changing their definition. In Section 2.2, I describe how a structure's memory layout relates to its definition, and especially in the presence of inheritance.

- I employ and specialize the program transformations class flattening and field reordering. I describe program transformations in general and both techniques in particular in Section 2.3 and discuss related work.

- I present a technique that can be used to improve portability of kernel code. In Section 2.4, I describe aspects that need to be considered when writing portable kernels.

2.1 Caches

A cache is a place for storing items for easier or faster access. To use an item, a *cache lookup* is performed first. If the item is found in the cache, the lookup resulted in a *cache hit*, and the item from the cache can be used. Otherwise, a *cache miss* is said to have occurred and the item is to be found somewhere else.

The uses for caches are manifold. Buffer caches in an operating system store recently accessed disk blocks; web caches store documents recently retrieved from the network; DNS caches store network addresses of recently resolved host names. Processors are often equipped with memory caches to store frequently used information such as data or instructions for fast access.

Caches exploit the *principle of locality.* Two types of locality can be observed in programs: *Temporal locality* means that an item used recently is likely to be used again soon. *Spatial locality* means that items placed near each other are likely to be used close together in time.

Over the decades, the performance of processors has improved much faster than the performance of memory. In recent years, this disparity worsened by approximately 42 percent every year [20]. Ideally, one would desire huge memories accessible at the full speed of the processor. However, hardware needs to be small to be fast, and fast hardware also tends to be expensive.

Caches are small but fast memories. Building on the principle of locality, they allow to maintain the illusion of huge, fast memories. They hold only a tiny subset of main memory's contents but make this subset accessible with a fraction of main memory's latency. Until the 1980s, processors often did not have any caches, whereas today two levels of caches are a commodity, and a third level can be found in larger servers systems. The size of the caches increases with their distance from the processor, but so does also the time to access them. Die photos of today's general purpose processors show that caches consume approximately 50 percent of die real estate.

2.1.1 Cache Architectures

A cache holds copies of portions of main memory's contents for fast access. The unit of data that is transferred between the cache and memory (or the next cache level) is called a *cache block*. The optimum size of a cache block has been found to be 32 or 64 bytes, depending on cache size. Memory can be thought of as an array of cache blocks or memory blocks.

The locations in the cache where a cache block can be stored are called *cache lines*. The process of loading a cache block into a cache line is called a *line fetch*; the process of removing a block is called *eviction* or *flushing*.

Associated with each cache line is the block's address in memory, called the tag, and various status bits. A typical status bit is the *valid bit* indicating whether the cache line is currently holding a cache block. A cache is organized as a number of sets of a fixed number of cache lines. The cache lines in a set are also called *ways*, and their number defines the cache's *associativity*. A cache with n cache lines in n/m sets of m cache lines per set is called an *m-way set-associative* cache. The two special cases $m = n$ and $m = 1$ are referred to as a *fully-associative* and *direct-mapped* caches, respectively.

Figure 2.1 illustrates the process of a cache lookup. The address of the data requested is used as input for the cache lookup process. The index bits of the address select a set. The tag bits of the address are compared in parallel with the tag bits of each line in the selected set if that line's valid bit is set. If a match is found, the offset bits of the address locate the requested data within the matching cache line. Due to the use of parts of the address for selecting a set, the mapping of memory locations to cache sets is known. Neighboring memory blocks map to different cache sets unless the cache is fully-associative.

Depending on where the cache is positioned relative to the address translation hardware, the addresses used for indexing and tag comparison can be virtual or physical addresses. Virtually addressed caches remove address translation from the critical path of a cache lookup. However, they need to be flushed when address translations change, for example during an address space switch. Physically addressed caches avoid this at the expense of a generally higher access time. Virtually-indexed physically-tagged caches determine the

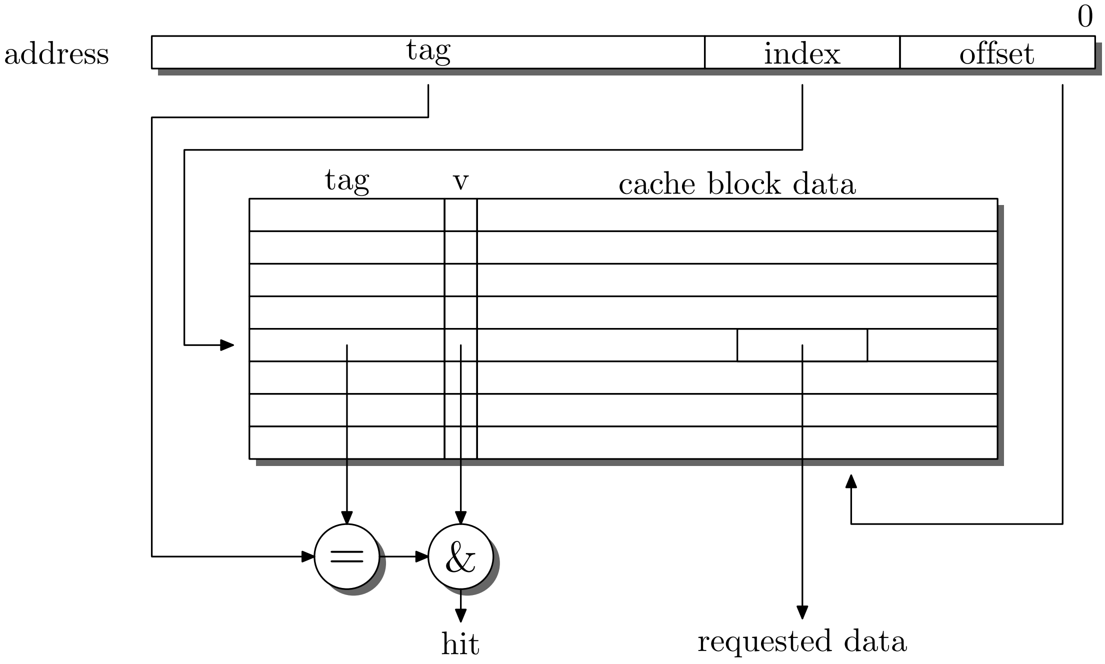


Figure 2.1: Cache lookup in a direct-mapped cache.

cache set using bits of the virtual address but then match the physical address against the tags, allowing indexing and address translation to proceed in parallel.

Except for direct-mapped caches where the decision is trivial, the replacement policy of a cache determines the line in a set a new cache block should be fetched into and thus replace that line's old contents. Ideally, the line that is least likely to be used in the near future, i.e., whose next use is the farthest into the future, should be replaced. However, such a prediction is usually impossible to make. Common policies that are nevertheless reasonably effective are *Round Robin* or *First-In First-Out*, *(Pseudo-)Least-Recently-Used*, or *(Pseudo-)Random*.

In a *write-through* cache, a cache hit on a write updates both the cache and memory. In a *write-back* cache, the same access updates only the cache; cache and memory become inconsistent, and the cache line is marked *dirty* by setting the respective status bit. The contents of a dirty cache line that is about to be replaced must be written back to memory before a new cache block can be loaded into the cache line. Replacement of dirty lines is therefore more expensive than replacement of *clean* lines which can simply be discarded.

A miss on a write in a *write-allocate* cache causes the respective cache block to be fetched into the cache. In a *no-write-allocate* cache, a write miss goes to memory and does not affect the cache. Write-back caches usually perform write-allocation in the hope for future accesses to hit in the cache. Write-through caches usually follow the no-write-allocate policy because future writes still need to update memory.

Multi-port caches can handle more than one request at a time, such as accepting a second request while the first request's data is being fetched from memory through a second port.

A cache may also serve requests in parallel through multiple request ports. To allow for such simultaneous accesses, caches may be organized in separately addressable banks with bits of the offset inside the cache line selecting the bank. Bank conflicts may arise from requesting data in the same bank but in different indices and may prevent simultaneous accesses.

In systems with more than one entity accessing memory, such as multiple processors with private caches or active devices, coherence between caches must be maintained for system correctness. Clean cache lines held in one cache whose contents are requested by another entity for modification must be invalidated; dirty cache lines must be written back to memory (or transferred to the remote cache) before their content can be accessed remotely. If all entities share a common memory bus, coherence can be achieved by monitoring or *snooping* the other bus agents' accesses. Each agent checks its local cache for blocks being requested remotely. In contrast, *directory-based* coherence protocols maintain information about the caching location and status of a block in one place, the directory.

2.1.2 Reducing Cache Misses

Unfortunately, not all memory accesses always hit in the cache. Various approaches in hardware and software have been devised to reduce the rate of certain types of cache misses. Hill and Smith [21] categorized cache misses as follows:

Compulsory misses occur when data is not found in the cache because it is being referenced for the first time. This situation arises after starting a program or after a cache flush, for example due to an address space switch.

Capacity misses occur when data is not found in the cache and all cache lines already hold valid data, i.e. when the cache is full. This situation arises when the working set of the program is too large to fit completely in the cache. Capacity misses would not occur in a cache of infinite size.

Conflict misses occur when data is not found in the cache and all ways of the set in which the data should be found already hold data but other sets still have unused lines. Conflict misses would not occur in a fully-associative cache of the same size.

Compulsory misses can be reduced by increasing the cache block size. However, with larger cache blocks the time to fetch a block into the cache increases as well. Capacity misses can be reduced by increasing the cache size, thereby also increasing hit time and power consumption. Conflict misses can be reduced by increasing associativity, which again may result in higher hit time and power consumption.

Alternatively, cache miss rates may be reduced by changing the software to best utilize the cache. Calculations on arrays can often be rearranged without affecting program correctness. *Loop Interchange* exchanges inner and outer loops such that the inner loop operates on array elements within a cache block rather than striding through the array. Nested loops that work on both rows and columns of arrays can in some cases be broken down into several smaller loops that operate on subsets of the elements; this technique is called *blocking* or *tiling* and can, like loop interchange, be performed by the compiler.

Cache-conscious data placement arranges non-array objects such as global variables, constants, and stacks globally to minimize cache misses. Cache-conscious allocators exploit domain knowledge to collocate and reallocate objects that are accessed together, for example the nodes in a tree. Structure splitting gathers the hot parts of objects and allocates the cold parts separately which are then referred to from their hot counterparts. Reordering functions within the program image can reduce conflict misses. Aligning functions and basic blocks with the beginning of cache blocks can reduce cache misses for sequential code by increasing utilization of a cache block.

2.1.3 Reducing Miss Latency

Cache misses are expensive. First, the next lower, slower level of the memory hierarchy must be informed of the access. Then it takes several bus transactions to fetch the block into the cache, because the data bus width is most often much smaller than a cache block. The parts of the cache block are transferred either in sequential order or according to some interleaving scheme. Several techniques reduce or hide the latency of a cache miss: On one hand, enhanced line fetch strategies reduce the time between the request of a datum and its availability to the processor. Prefetching, on the other hand, tries to hide the latency of the memory hierarchy by requesting data before they are accessed.

A straightforward implementation of a cache might stall the processor until the complete block has been fetched into the cache. The *Early Restart* strategy releases the processor as soon as the requested data has been fetched, even though parts of the cache block are still in transfer. A subsequent request to data in the same block will be stalled until the block has been fetched completely. *Streaming* applies Early Restart also to subsequent requests to the line being fetched. *Requested Word First* transfers the part of the cache block first that contains the requested word, thereby minimizing the latency for the requested word. *Nonblocking caches* support out-of-order execution cores by continuing to serve requests to other cache lines while handling a miss.

Hardware prefetching looks for regularities in the addresses of memory accesses in order to detect streams and requests cache blocks before the program accesses the data. Often, multiple independent streams can be tracked concurrently. With *software prefetching*, the compiler or developer inserts special prefetch instructions to request the data. In contrast to hardware prefetching, software prefetching can avoid prefetching beyond the end of a stream and the resulting unnecessary bus traffic. Software prefetching can also benefit irregular access patterns if the distance between the prefetch instruction and the actual access is large enough. Prefetching too early, however, can be counterproductive by replacing other data that are needed between fetch and use of the prefetched data.

2.2 Data Structure Layout

Computers store the information they process in data structures. A *data type* is the description of a data structure whereas an *object* is an instance of a data type. The description of a data type is called *type definition*.

Data types can be classified into elementary types and compound types. *Elementary types* describe the basic data types of the language such as characters, integer and floating point numbers. Objects of elementary type are stored in consecutive bits of storage such as in a register or in one or more bytes in memory.

Compound types are constructed from a finite number of components, each of which can be of elementary type or of compound type again. Examples for compound types are arrays and classes. A known or bound number of objects of the same type can be stored in an array. With the help of object references, objects can also be arranged in lists or trees, which are suitable for managing an unknown number of objects. A *class type* groups a set of objects of elementary or compound type and a set of associated functions.

Type-unsafe languages, such as C and C++, allow the programmer to reason about the internal structure of an object. This is one of the reasons why C is popular for system programming where externally defined data structures need to be manipulated. In contrast, type-safe languages generally hide the details of object representation from the programmer, leaving the object layout to the compiler or the run-time system. In some cases, however, extensions to type-safe languages allow the precise definition of compound types' object representations, for example in the Common Intermediate Language [17].

This section describes the in-memory representation of compound C++ objects of class type and illustrates how this representation is influenced by the various methods of type construction.

2.2.1 Classes

A class groups related variables and functions. The variables are called data members or fields; the functions are referred to as member functions or methods. Data members represent the attributes (or the state) of an object, whereas member functions define the behavior of an object. A class defines a *has-a* relation between the class and its data members. Each object of a class contains all nonstatic data members of the class. In contrast, static data members exist only once per class; they are shared between all instances.

The following extract from the C++ specification [25, clause 9.2, Class members, paragraph 12] defines how the internal memory layout of an object is derived from the structure of the class definition:

> Nonstatic data members of a (non-union) class declared without an intervening access-specifier are allocated so that *later members have higher addresses* within a class object. The order of allocation of nonstatic data members separated by an access-specifier is unspecified (11.1). Implementation alignment requirements might cause two adjacent members not to be allocated immedi-

> ately after each other; so might requirements for space for managing virtual functions (10.3) and virtual base classes (10.1).[1]

Access specifiers are the keywords `private`, `protected`, and `public` that define the accessibility of members from non-member functions of the class. Figure 2.2 shows an example of a simple class definition and the corresponding object memory layout.

```
class A
{
  char a;
  char b;
  char c;
};
```

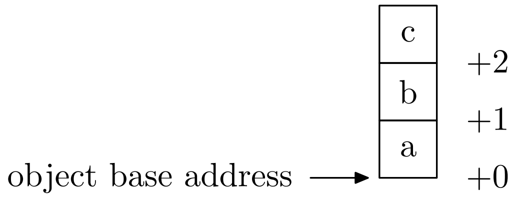


Figure 2.2: A simple C++ class and its memory layout.

Alignment requirements of members may cause insertion of unnamed space, padding, in the object. Furthermore, the alignment requirement for objects of the class is derived from the maximum alignment requirement of all data members [25, clause 3.9, Types, paragraph 5].

Often, however, alignment requirements for members of a class can be overruled. With the GNU C++ compiler, for example, the `attribute(packed)` type attribute added to a class specification forces the compiler to assume the minimum alignment for all members: one byte for members, one bit for bitfield members. As a result, no padding larger than or equal to a byte will occur in objects of that class. The same can be achieved by placing `#pragma pack(1)` before the class definition. Packing can also be specified globally as a command line option`-fpack-struct[=<n>]`. The pragma specification is the most portable as it is consistently supported by most C and C++ compilers [19, 22, 24, 39].

Using the packing mechanism, a programmer has complete control over the object representation of a simple class type.

2.2.2 Inheritance

Inheritance allows to form new classes based on classes that have already been defined. The new, *derived class* (or subclass) *inherits* all data members and member functions from the *base class* (or superclass.) An object of the derived class is often referred to as a *derived object*.

A derived class can add new data members or methods, thereby enabling extension. It can also *override* inherited methods or *shadow* inherited data members to modify the behavior, thereby enabling specialization. Code reuse is enabled by inheriting methods from the base class that have identical implementations for derived and base classes instead of duplicating the implementation in the derived class.

[1] Italic typeface not present in original.

Inheritance defines an *is-a* relation between a derived class and its base class. Since a derived class inherits all members from its base class, an object of a derived class (*actual type*) can serve as an object of a base class (*current type*), for example through an explicit type cast or by assignment to a base class pointer. This concept is called *polymorphism*.

Inheritance can be applied recursively, forming a *class hierarchy*. The class that a derived class inherits from, i.e. that is listed as a base class in the derived class' specification, is referred to as a *direct base class*. Base classes of direct base classes are *indirect base classes*. A class can serve as a direct base class for zero or more derived classes. Likewise, a derived class can inherit from one or more direct base class. A *mixin* is a class that cannot stand by itself but provides functionality to be inherited by a derived class. The effects of different types of inheritance on object layout are discussed below.

Simple Inheritance

Inheriting from a base class places an object of the base class as a compact subobject into the derived object. The layout of the base class subobject is most often identical to that of an object of the base class, although the C++ specification does not mandate this. By placing the subobject at the start of the derived object and any additional data members declared in the derived class behind it, a derived object can be treated as an object of its base class at the same address. For example, a pointer to the derived object can be assigned to a pointer to a base class object without any address calculation required.

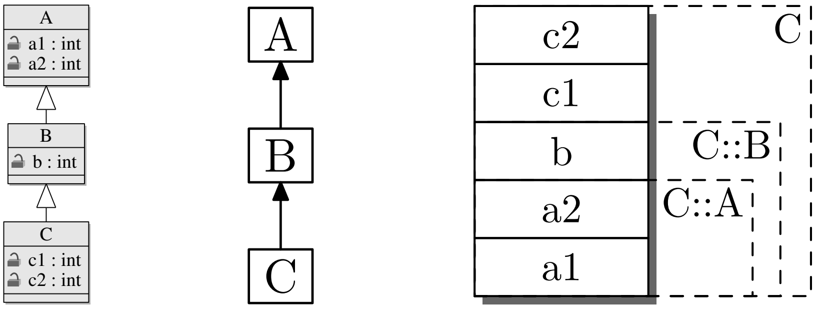


Figure 2.3: A simple C++ class hierarchy, the object lattice, and the memory layout of a most derived object. Inherited members form compact base class subobjects within the derived object. Members added in a derived class follow.

Figure 2.3 shows an example of a class hierarchy and the memory layout of the most derived object. Placement of data members in a derived object is first governed by the inheritance chain, then by the order in which added members are declared.

Polymorphic Functions

A derived object can be treated as an object of a base class; a derived class can override functions defined by a base class. If a method is invoked on such a derived object while it is being treated as a base class object, the version of the method defined by the base class will be called. In contrast, polymorphic functions, also known as *virtual functions*, allow to invoke the implementation defined in the derived class. The keyword `virtual` in a method declaration is used to explicitly request this behavior. For a virtual function, the version of the method that should be called depends on the actual type of the object. Therefore, the object needs to carry some sort of type identifier with it.

The address of a virtual function must be found dynamically at execution time. This process is referred to as *late binding*. A common implementation of late binding uses a virtual function table, the *vtable*. Such a vtable contains the entry points of all virtual functions of a class. Since the entry points are the same for all objects of a derived class, the vtable can be shared among all objects of a class. The vtable is referenced from each object via a *vtable pointer* which effectively acts as the type identifier of the object.

The vtable pointer is a hidden data member in the object; it is neither declared as a member nor can it be addressed from within the language. It is usually placed at the start of the object, but sometimes also at the end (for example, in Cfront [55].) A derived object and its base class object can share the vtable pointer, because their vtables contain pointers to the very same function implementations. Virtual functions introduced in the derived class are appended to the vtable. Those functions will never be invoked from the base class as it is not aware of the derived class.

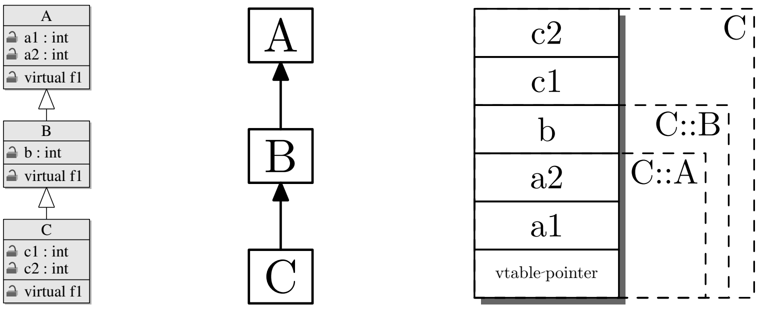


Figure 2.4: Object memory layout of a class with virtual functions.

Figure 2.4 shows an object layout of a class with virtual functions. Compared to an object of an otherwise identical class with no virtual functions, the object size increases. Also, depending on where the implementation stores the vtable pointer within the object, data members may be located at different offsets.

Given an object's address, invoking a virtual function on this object then involves the following three steps:

1. Read the vtable pointer from the object;
2. Read the address of the function from the vtable at the virtual function's index;
3. Call the function[2]

[2] Depending on the architecture this step and the previous step could be combined in one instruction.

Two indirections via memory are necessary to reach a virtual function, whereby the first indirection is dependent on the object address. The dynamic selection of the actual implementation also makes virtual functions unsuitable for inlining. Research has been performed to improve the performance of virtual function invocations. Several approaches favor replacing the dynamic function call with a switch statement and a number of static calls [2, 11, 29].

Multiple Inheritance

Multiple inheritance refers to a situation where a derived class has more than one direct base class. Subobjects of all direct base classes are placed in the derived object. The placement of those subobjects is not defined by the C++ standard. However, most compilers place base class subobjects in the order in which the base classes are listed in the derived class' specification. Figure 2.5 shows a class hierarchy with multiple inheritance and a possible object layout.

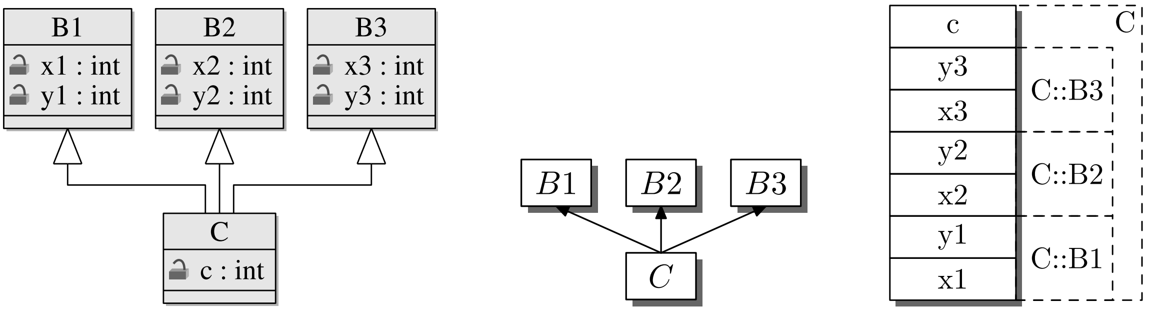


Figure 2.5: A C++ class hierarchy with multiple inheritance and the resulting memory layout of an object of the most derived class C.

In addition to the most derived object's vtable pointer, each base class object may have its own vtable pointer, too (not shown in Figure 2.5). The derived class can, however, share its vtable pointer with one of the base classes. Especially for small objects with many base classes the storage overhead for vtable pointers may be significant.

Multiple Inheritance with Common Base Classes

A class can be a direct base class of a derived class only once. However, it can be an indirect base class more than once. In such a scenario, multiple subobjects of the base class exist in the derived object, as is shown in Figure 2.6.

Each of those subobjects can be manipulated independently. Naming ambiguities, which arise when addressing members inherited from the common base class, must be resolved using the scope resolution operator :: to identify the intermediate base class whose subobject is meant.

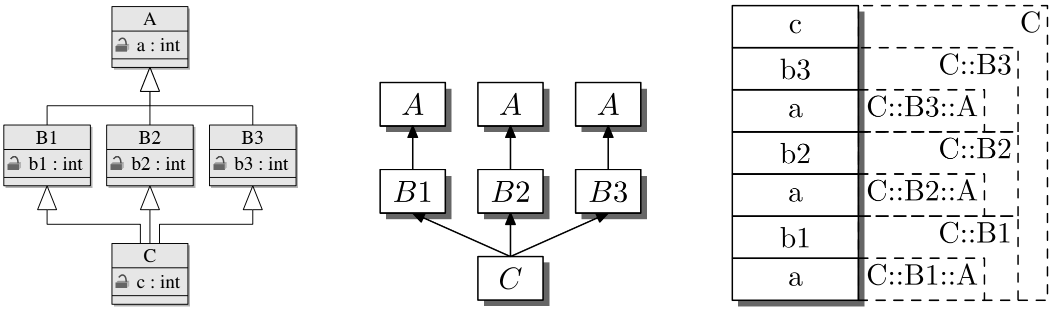


Figure 2.6: A C++ class hierarchy with multiple inheritance and a common base class and the resulting memory layout of an object of the most derived class C.

Multiple Inheritance with Virtual Base Classes

A class becomes a *virtual base class* when it is prefixed with the keyword `virtual` in a derived class' inheritance specification. Being a virtual base class is not an attribute of the base class; it is determined by how the class is inherited. Subobjects of virtual base classes exist only once in the derived object and are shared between all derived classes that inherit them. Figure 2.7 shows an example of virtual base classes.

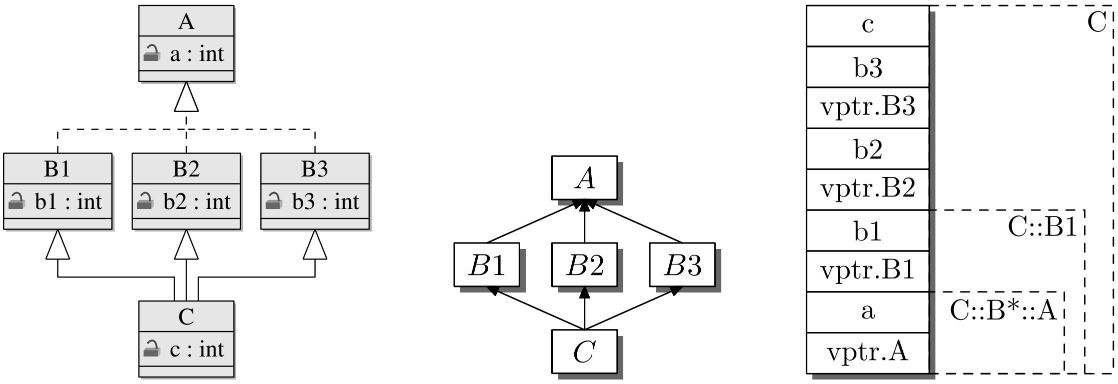


Figure 2.7: A C++ class hierarchy with virtual inheritance and the resulting memory layout of an object of the most derived class C. Subobject A is shared among objects of all classes that virtually inherit from A, directly or indirectly.

Subobject `A` is shared between the objects of `B1`, `B2`, and `B3`. Only one instance of `A` exists in `C`. `C`'s subobjects `B1`, `B2`, and `B3` each contain a pointer to a vtable. The subobjects of `B2` and `B3` span noncontiguous memory locations. `A` is part of `B2`, yet neither `b1` nor `B1`'s vtable pointer belong to `C::B2`; likewise for `B3`. Besides pointers to virtual functions the vtables contain the displacement of subobject A relative to the containing object (`B1`, `B2`, or `B3`). A reference from `B2` (or `B1` or `B3`) to `A::a` thus requires an indirection and an address calculation.

To complicate matters, a class can be a virtual base class and a nonvirtual base class at the same time, so that shared and unshared base class subobjects exist.

Summary

For simple classes, data members appear in the object in the same order as they appear in the class specification. Data members of derived objects, however, are first placed according to inheritance relationships and then by order of appearance. Furthermore, metadata may be included in the object. Consequently, the placement of members in objects of complex class hierarchies can not be influenced by the programmer as freely as in a simple class.

2.3 Program Transformations

Program transformation is the process of transforming one program into another. Two variants of program transformation can be distinguished: *rephrasing* transforms the program within the language whereas *translation* changes the language of the program. Compilation translates a program given in a high-level language into machine code so that the program can be directly executed by a processor. Rephrasing aims to improve a certain aspect of a program while maintaining its behavior.

This section describes the build process for C++ programs and the two program transformations class flattening and field reordering. It also discusses related work in the area of those transformations.

2.3.1 Build Process

For more than 25 years, the approach to turning OS source code into an executable has not changed much. C has been and still is the predominant programming language in all major desktop and server operating system kernels today [9, 36, 52, 53].

The build process for operating system kernels written in C or C++ does not differ much from the process of building applications in those languages. The most notable difference is that operating system kernels are highly self-contained whereas applications often rely on functionality provided in libraries.

Many programming languages support separate compilation. Source code modules or units, often stored in a separate file, are translated one-by-one into object files. All object files are then combined to the final program. Separate compilation can significantly reduce the build time of large programs by compiling only the modules that were modified or whose dependencies were modified.

The common build process for C and C++ programs (and OS kernels) consists of the following four stages:

preprocess Starting with the source file that is to be translated, the preprocessor replaces `#include` directives with the contents of the file they refer to. This process is recursive. The result is a single, linear representation of (at least) all the code that is necessary to perform the translation, such as data types, base classes, and declarations of functions or variables implemented in other source files that are referred to from the source current file. #include directives allow to share common code among multiple source files.

In a second step, macros are expanded. Macros can also take parameters that are substituted during expansion. Furthermore, the presence or value of macros can be tested using the #if directive. Depending on the expression used, all text between an #if directive and its corresponding #endif directive are included or removed, allowing for *conditional compilation*.

compile The compilation stage transforms source code from the high-level language to assembly language. This stage itself consists of several steps. A front end for the high-level language performs lexical, syntactical, and semantical analysis of the input and generates an intermediate representation of the program. A middle end performs further analysis (e.g., dependence analysis, alias analysis, pointer analysis) and optimizations (e.g., inline expansion, dead code elimination, constant propagation, loop transformation, register allocation.) Finally, a back end for the target emits assembly code for the optimized intermediate representation, involving instruction scheduling, selection of addressing modes, register and memory allocation, etc.

assemble The assembler translates assembly code into blocks of binary instructions, blocks of data, and address information, the symbol table. Symbolic local jump targets turn into offsets of jump instructions; function entry points and global data appear as public symbols; references to nonlocal memory objects or jump targets are marked as external symbols requiring relocation. The output of the assembler is an object file.

link During the link phase, all object files are combined into the final binary. External symbols of one object file are matched with public symbols of other object files. Code and data of all object files are merged into common sections. Object-file relative addresses are assigned addresses relative to the final binary, and addresses for external symbols are filled in. External symbols that cannot be found in the object files of the program are looked up in libraries for linking at program load time. The complete program is emitted as a final binary file that can then be loaded by the operating system for execution.

2.3.2 Class Flattening

Class flattening is a rephrasing program transformation. From a derived class in a class hierarchy (the *source class*), it creates a simple class (the *target class*), which does not use inheritance. The flattened target class directly contains all direct and inherited members of the source class and has no base classes itself, as illustrated in Figure 2.8.

Class flattening includes all direct and inherited members of the source class into the target class. This can be achieved by walking the inheritance graph, beginning at the

```
class A {
public:
    int a;
    void f() { /* A::f */ };
    void g() { /* A::g */ };
};

class B : A {
public:
    int b;
    void g() { /* B::g */ };
    void h() { /* B::h */ };
};
```

(a) Class hierarchy for B

```
class FlatB {
public:
    int a;
    int b;
    void f() { /* A::f */ };
    void g() { /* B::g */ };
    void h() { /* B::h */ };
};
```

(b) Class B flattened into FlatB

Figure 2.8: Class flattening creates a stand-alone class without base classes from a derived class' hierarchy. All members accessible in the derived class appear in the flattened class.

source class, in a breadth-first manner. All members that do not yet exist in the target class are copied there whereby the language's rules for hiding and overriding of inherited members define whether a member already exists.

The *flat form* of a class was introduced by Meyer [37]. 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. A first brief description of the flattening process was given by Meyer in 1997 [38] whereas a command line utility to create a flat form of Eiffel classes was already available in 1998 [37].

Independent of Meyer, Bellur et al. describe a class flattening tool for C++ [5]. The flattener manipulates preprocessed source code. The authors target class flattening as a means to *eliminating virtual functions* for which they describe two alternative approaches: In the stand-alone approach, the flattener copies all relevant member functions and data members from a selected class K and its base classes into a new class $FlatK$ and removes the inheritance relationship. In the parallel hierarchy approach, a separate class hierarchy with $Flat$-prefixed class names is created where all virtual functions are de-virtualized while the inheritance relationship is maintained. To use the flat version of a class, the programmer and/or the flattener can turn selected variables of type K or related types into the respective type for $FlatK$ (variable flattening). The flattener can also create copies of client functions, i.e., functions that have at least one parameter of type K or related type (client function flattening). These functions with parameters of type $FlatK$ instead of K overload the original client functions. They will be used whenever a client function is called with the flattened version of the class. An automated approach for variable flattening is suggested by the authors, but considered infeasible due to the high cost of a full data flow analysis. They report a 4.25-5.58 times faster execution for two example programs that are best cases for flattening: short, inlinable virtual functions; no use of dynamic polymorphism.

A second use of class flattening proposed by Bellur et al. [5] is *enhancing program understanding* by presenting a flat form of a class in a source code browser, like suggested by Meyer. Furthermore, the authors mention that class flattening can also ease debugging, because execution no longer jumps up and down in the class hierarchy.

Another use of class flattening can be found in the area of *software quality measurement*. A very detailed description of the class flattening process for C++ by Beyer et al. [6] has been implemented in the Crocodile measurement tool [34]. Beyer et al. also discuss the impact of inheritance on software metrics like size, coupling, and cohesion [7]. Comparing values for original and flattened versions of classes led the authors to an improved interpretation of values for the unflattened version. Furthermore, the additional data allowed to detect more candidates for restructuring, and gave insights into the use of inheritance, be it for source code sharing, definition of interfaces, or creation of type hierarchies.

Binder [8] applies class flattening to reduce complexity in the context of *software testing*. There, only a flattened class allows to define a reasonably sized class test plan for all features inherited.

All instances of class flattening in previous work create a second, flattened variant of the class definition, which coexists with the original. This work proposes *transparent class flattening* which *replaces* the original class definition with the flattened variant.

2.3.3 Field Reordering

Field reordering arranges fields of a compound data structure according to an optimization criterion. It is commonly used to improve the spatial locality of compound data structures to optimize cache usage when accessing them. Field reordering can be performed in the run-time system, in the compiler, or as a source-to-source transformation. The latter relies on a correlation between the order of fields in the type specification and the resulting object layout, such as described for C++ in Section 2.2.1.

In type-safe languages, field reordering is automatically safe because the object's in-memory representation is left to the compiler or run-time system. In other languages, for example in C or C++, the programmer can reason about the internal structure of an object and make static assumptions about the placement of fields. Arbitrarily reordering fields could therefore negatively impact program correctness.

Truong et al. [60] present *field reordering* as a technique to improve the cache behavior of dynamically allocated data structures in C. With field reordering, "fields of a data structure often referenced together are grouped together to fit into the same cache line." Truong et al. exploit the fact that nonstatic data members of a C data structure are assigned increasing addresses in the order they appear in the data structure declaration. This behavior is defined in the C language specification [26]. Consequently, changing the field order in the declaration influences their mapping to memory addresses and thus to cache lines. Truong et al. leave 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 another optimization technique, instance interleaving, Truong et al. obtain speedups of 1.08–2.53, reducing cache or TLB miss ratios by 35 to 96 percent.

Semi-Automatic Field Reordering

Chilimbi et al. [14] automate field reordering for C programs in so far that a tool produces recommendations for new field orderings. These recommendations need to be checked and eventually implemented by the programmer, because the authors deem automatically determining whether layout manipulations are safe with respect to program correctness impossible. Profiling information from a previous program run is combined with static analysis in order to, in the end, construct a field affinity graph for every structure type. In these graphs, nodes represent fields and edge weights are proportional to the frequency of the fields being accessed contemporaneously, with contemporaneously meaning being accessed within a certain time (100ms). All instances of a type are treated identically, as most instances were found to show "similar access characteristics (i.e., consecutive accesses to the same field in different (indistinguishable) instances, rather than different fields)." Chilimbi's approach places fields with high temporal affinity *near* each other; no assumption is made about structure alignment on cache-line boundaries, as this "can only be determined at run time". Chilimbi et al. reports performance improvements of 2 to 3 percent after reordering five of the most frequently used data structures in Microsoft's SQL Server.

Kistler and Franz [28] call field reordering "data member clustering". A second technique, referred to as "data member reordering" attempts to optimize the order of fields *within* a cache line. Memory interleaving and cache line-fill buffer forwarding are identified as source of different latencies for the words in a cache line after a cache miss. Like Chilimbi et al., Kistler and Franz build temporal relationship graphs (TRGs) for objects. Nodes in the graph correspond to fields in the object. The weights of the edges reflect the number of times fields were accessed subsequently within a specific number of disjunct memory references. "The TRG is created by collecting path profiling information and then stepping through each program path returned by the profiler." These graphs are then subjected to a graph partitioning algorithm that associates fields with cache lines. It is, however, left unclear how multiple potentially different TRGs for objects of the same class lead to an optimized layout for the class. "Finding an optimal order of fields within cache lines is done with an exhaustive search..." By limiting the application to type-safe programming languages such as Oberon, the optimization process can be fully automated. Kistler and Franz also discuss optimizing the layout of derived objects. Subobjects inherited from supertypes are considered inseparable. Layout optimizations are thus restricted to how fields introduced in the derived object are placed behind the inherited subobject. The likely but unstated reason for this limitation is polymorphism. The paper also states that, "Encapsulation of object types often leads to subtypes having only minimal access to their inherited fields, thereby reducing temporal relations between fields from different derivation levels." However, access patterns to fields of a data structure are actually not related to accessibility of inherited fields from derived types. Kistler and Franz report speedups of 3 to 96 percent for their layout optimization. How much of this speedup can be attributed to "data member reordering" (inside a cache line) on top of "data member clustering" (of fields into cache lines) is not described.

Zatloukal et al. [65] present a slightly different algorithm for finding an optimized field ordering. A program's access behavior is modeled in a per-structure member transition graph (MTG). The nodes in the graph represent the members. The edges between any members i and j carry transition and survival probabilities. The transition probability describes how likely member i is accessed immediately before member j. The survival probability describes how likely a cache line that was in the cache when member i was accessed is still in the cache when member j is accessed. The graph is built from data collected during the program's execution. Access to the compiler's type information allows gathering traces that include each memory address accessed, and for each access to a structure member, the name of the structure and its member. From the MTG, cache hit probabilities can be determined for any member ordering by a set of equations. The optimization algorithm uses a Branch-and-Bound algorithm [30] that considers all possible orderings for small structures, and a Local Search algorithm [1] that iteratively improves the initial ordering for larger structures. Finally, the ordering suggested by the optimization algorithm is subjected to a simulation step that reports cache miss rates for the reordered structures based on the initially collected trace. The result of the optimization is a new ordering for members that the programmer may choose to implement. Zatloukal et al. report a performance improvement of 1.3 percent after reordering seven data structures in Microsoft's SQL server. This is less than the results reported by Chilimbi et al. [14], because a limitation in the profiler did not allow to reorder any of the five most frequently used structures that were optimized by Chilimbi.

None of previous work on field reordering in type-unsafe languages has completely automated the process of field reordering. At best, a new ordering is suggested, which then still needs to be verified and — if found safe — be performed manually by a programmer by modifying the structure definition. Furthermore, field reordering has been applied to C, but not to C++ and therefore not to C++ class hierarchies either.

Complexity

There has been considerable research into the complexity of field reordering. For complete access traces an algorithm can find the optimal layout in exponential time: it simply tries all possible placements. Thabit [58] showed that optimal data packing using pairwise frequency information is NP-hard and that the optimal packing also depends on the cache's replacement policy. Likewise, Kennedy and Kremer [27] showed that the problem is NP-hard. Petrank and Rawitz [44] showed that no polynomial time method can guarantee a data layout whose number of cache misses is within $O(n^{1-\epsilon})$ of that of the optimal data layout, where n is the length of the trace. In addition, if only pair-wise information is used, no polynomial algorithm can guarantee a data layout whose number of cache misses is within $O(k-3)$ of that of the optimal data layout, where k is the size of the cache. The results hold even when the computation sequence is completely known, objects have the same size, and the cache is set associative. These general results, however, do not preclude effective optimization targeting specific (rather than all) data access patterns [66].

2.4 Kernel Portability Aspects

Porting is the process of bringing a software to an environment different to the one(s) it has been written for. Porting often requires modifications to the software in order to adapt to the new environment. Such modifications are either generalizations or (partial) reimplementations. Generalization replaces parts of the software that are incompatible with the new environment with a more generic variant whereas partial reimplementation adds variants that are compatible with the new environment. In contrast to rewriting a piece of software from scratch for every new environment, porting promises less effort through the reuse of parts that need not be adapted. Portability is a software metric expressing the ease of porting.

The challenge for portable software is configuration diversity. A configuration is characterized by numerous aspects. The most notable aspects are instruction set architecture, word width, and run-time environments. However, several more hardware properties as well as diversity of software and development tools add to the configuration space. This section discusses various aspects of the environment that a portable kernel needs to address.

2.4.1 Hardware

Porting software to a new hardware platform involves more than recompiling for the target's instruction set. Other aspects deserving consideration include the target's word width and endianess, and the topology of the system. Furthermore, implementations of conceptually similar hardware often differ more or less so that supporting them requires porting as well.

Instruction Set

An instruction set architecture (ISA) specifies the native data types of a machine, instructions and their binary representation (opcodes), addressing modes, registers, operating modes, and handling of internal and external events such as exceptions and interrupts.

All major processor vendors have developed their own ISA or ISA extensions. Over the years ISAs have evolved, although not always in a backwards compatible way. Examples are the reuse of removed instructions' opcode space for new instructions[3] and slightly different instruction behavior in newly added operating modes.[4]

High-level languages abstract from the instruction set; the mapping to actual instructions is performed by the compiler. Code written in the high-level language is generally portable provided it does not make assumptions about the underlying instruction set (as self-checking or self-modifying code would do.)

[3] For example, x86's `INC` and `DEC` instructions were replaced by x86-64's `REX` prefix.

[4] Compare default operand sizes in x86's real mode, protected mode, and long mode.

However, certain instructions that are required for system programming have no representation in the high-level language and thus need to be emitted separately, for example in embedded assembly fragments (inline assembly) or in assembly source files. Also, exceptions and interrupts influence the software-observable state of the machine and require operations (e.g., saving and restoring the context of interrupted code) that cannot be captured in the high-level language. Such code must be rewritten during porting.

Word Width

The width of a general purpose register limits the size of the data that can be manipulated efficiently with arithmetic and logic instructions. It is also the most common amount of data transferred between memory and the register set. The machine word width has been used to define the size of certain data types in high-level languages. For example, the C++ specification defines that `int` be as wide as the word width of the machine. Furthermore, it defines that `char`, `short int`, `int`, and `long int` each provide the same or more storage, in this order.

The width of memory addresses determines the size of the address space and thereby limits the amount of data directly accessible to an instruction. Often, the width of an address is identical to the width of a general purpose register. The C++ specification, however, leaves the size of a pointer as implementation-defined.

Assumptions, that the sizes of pointers, `int`s, and `long int`s are identical and that values of these types can be assigned without loss, are unportable. The size of compound data types and the offsets of members in classes may differ between machines of different word width. Relying on a particular word width when shifting bits "off the word" is not portable either.

Endianess

In most programming models, memory is considered a byte-addressable storage media. All words larger than eight bits require more than one byte for being stored in memory. The byte that stores the least significant bits is the least significant byte. The endianess of a system defines how these bytes map to consecutive memory addresses. A *little endian* system stores the least significant byte at the address of the word, with the remaining bytes following in order of increasing significance. In contrast, a *big endian* system stores the most significant byte first, followed by the remaining bytes in oder of decreasing significance.

A system's endianess is generally transparent to the C programmer. However, it becomes visible when aliasing memory using different types. For example, accessing the same memory location as both an `int` and a `char` yields different values on big and little endian systems.

Another difference concerning the declaration of bit-fields and how individual fields map to bits of the containing type manifests when working with unions of bit fields and integers. The C++ specification merely defines: *Bit-fields are assigned right-to-left on some machines, left-to-right on others.* It is, however, commonly observed that bit-fields are assigned right-to-left on little-endian machines, and left-to-right on big-endian machines. That is, the first bit field declared will be stored in the least significant bit of the first byte occupied by the bit-field in memory. Therefore, for a given bit-field layout to map to particular bits in an overlapping integer type, the declaration of the bit field must be adapted to the endianess of the target.

System Topology

The topology of a system, i.e., the number and type of cores, the caches, the buses, and the memory configuration, influence decisions with respect to system correctness and performance.

The amount of parallelism available in a system determines the selection of suitable kernel algorithms. While code on a uniprocessor system can safely assume that no other code is executing at the same time, this assumption can lead to incorrect results on a multiprocessor system. Mechanisms such as locking or inter-processor communication must be employed to coordinate access to shared state and parallel execution. Such multiprocessor mechanisms impose overheads that can be avoided in a uniprocessor configuration. Unlike Java, both C and C++ are inherently single threaded at the language level. Their execution models do not cater for concurrency. Hence, synchronization must be explicitly designed into algorithms.

Memory latency (ignoring caches) does not differ among the execution cores in simultaneous multithreading (SMT) and multicore systems. In ccNUMA systems, however, with multiple memory blocks backing different parts of the global physical address space, access latency is a function of the distance between the requester of a datum and the datum's location. Therefore, code and data require careful placement to minimize access latencies and avoid unnecessary bus traffic.

The structure of the memory hierarchy, especially the location of caches, is influential to decisions about what information can efficiently be shared between cores and where. Often, frequently updated, shared data can be turned into more local instances on which a higher-level cooperation protocol operates, thereby reducing cache-line bouncing. Exact information about the line size of caches allows to avoid false sharing [59].

Common Hardware

Across a wide range of configurations, there exists a variety of hardware to support concepts such as virtual memory, scheduling, and communication. While each concept is implemented in very similar ways, the details of the implementation differ, as the following three examples show.

All memory management units (MMUs) that translate virtual to physical addresses use a cache, the translation look-aside buffer (TLB), to speed up the translation. Some MMUs invoke a software handler on a TLB miss, others contain hardware to walk tables in memory to load the missing translation information. Common to both approaches is a software-managed structure in memory that is then looked up by the TLB-miss handler in software or by TLB-miss hardware. In the latter case, the format of the structure in memory is defined by the hardware, whereas with a software-loaded TLB only an interface for installing or removing single entries in the TLB is defined.

All timer devices have the following in common: they can be programmed to “tick” at fixed intervals (periodic timers) or once after a number of cycles has passed (one-shot timers), and they can generate interrupts for those ticks. However, the device registers to program the interval are very diverse across platforms.

Another prominent example of slightly different yet standard hardware are serial ports as could be found in PCs for the last two decades. Their NS16550-compatible controllers are programmed via a well-defined set of 8-bit registers. Device drivers for those controllers bear a high similarity across all platforms. They only differ in the method of reading and writing the controller registers, that is, at which address the registers are mapped and which instructions are used (e.g., memory or I/O space) to access them.

2.4.2 Software

Besides the hardware, also the software environment can differ between configurations. For applications, the run-time environment is part of the software environment. For a modular kernel that runs on the bare hardware with no support layers underneath it, the software environment is defined by the internal and external interfaces of the kernel. Aside from rather high-level kernel components, internal interfaces exist between different programming languages. The external interfaces of the kernel are its application programming interface (API) and its application binary interface (ABI).

Calling Conventions

A calling convention defines how parameters and return values are passed between the caller of a function in a high-level language and the function being called. It also defines who (caller or callee) is responsible for saving and restoring resources used by both. Most calling conventions use processor registers, the stack, or a combination of both. Calling conventions may also designate certain registers for particular purposes, such as defining a general purpose register as the stack pointer register.

Often, several incompatible calling conventions exist for a given architecture. They originate from different programming languages (e.g., Pascal vs. C), compiler vendors, or even compiler versions. If multiple conventions are supported by a compiler, one particular can often be chosen at the file scope or even for single functions.

Normally, the compiler takes care of generating the code for placing parameters in the correct registers or memory locations, for saving and restoring registers that must be preserved, and for extracting return values. A kernel programmer is exposed to the details of calling conventions when hand-written assembly code needs to invoke functions implemented in the high-level language.

Inline Assembly

In system programming it is sometimes necessary to access registers or invoke instructions that have no representation in the high-level language. Inline assembly is a convenient mechanism for embedding assembly code directly in the body of a function. The C++ standard sets aside the `asm(...)` construct for this purpose, whereby the details of `...` are vendor specific.

The assembly fragment is often specified with preconditions, post-conditions, and a clobber list. Pre- and post-conditions define input and output constraints which describe the mapping between elements of the high-level language (e.g., variables, object addresses, constants) and elements of the assembly language (e.g., registers, addresses, immediate values) before and after the assembly code fragment. The clobber list informs the compiler of what other resources are modified as a side effect of the assembly code such as further registers, flags or memory.

The following example illustrates the power of this mechanism. The inline assembly code embeds an x86 instruction `XCHG`, which atomically swaps its operands, into the current function. More specifically, the code atomically writes the value 1 into the memory location of variable L and stores the previous content of L in variable v, resembling a simple version of the lock acquire operation.

```
asm ("xchg    %0, %[lock]"
     :        "=r" (v)     // output  (%0, register, store in v)
     :        "0"  (1),    // input   (same as %0, preload with 1)
       [lock] "m"  (L)     // input   (memory location of L)
     : "memory");          // clobber
```

From this specification the compiler may generate the following assembly code in the body of the function:

```
         movl    $1, %eax         # 1 => tmp
#APP
         xchg    %eax, -12(%ebp) # tmp <=> L
#NO_APP
         movl    %eax, -8(%ebp)  # tmp => v
```

The constraints in the inline assembly string (`%0` and `%[lock]`) are substituted with the respective registers or address expressions (`%eax` and `-12(%ebp)`) allocated for the elements of the high-level language. Code to satisfy pre- and post-conditions is generated before and after the inlined assembly code which is enclosed between `#APP` and `#NO_APP` comments in the example above.

Except for the constraints, inline assembly code is a meaningless string to the compiler. Therefore, optimizations must treat it as a black box; that is, its instructions are not reordered or broken apart. The compiler, however, is free to schedule surrounding code as it sees fit as long as dependencies are honored.

Assembly code in dedicated files is quickly identified as being inherently nonportable. In contrast, inlined assembly code has the often unexpected potential to make a file or function written in a high-level language nonportable.

Member Offsets

When interfacing assembly code to higher-level languages, it may become necessary for hand-written assembly code to access nonstatic data members in objects of a high-level compound structure of which only the base addresses are available.

At least three approaches for determining the offsets of individual data members in composite structures can be distinguished: dummy sections, access to compiler tables, and machine generated header files. They are briefly described below.

Dummy sections For every composite structure, an empty section is generated with a section address of zero. Nonstatic data members of the structure are assigned symbolic names in this section at the appropriate offset. The address of a structure member can then be generated by adding the offset of its symbol in the section to the base address of the object.

Compiler tables Compilers often generate additional non-program information such as debug information, which may also include field offset information. An assembler may allow to programmatically refer to such information from assembly code. Field offsets can then be used as assembly language constants that are added to the object's base address.

Machine-generated header files In C and C++, the macro `offsetof(type, member)` returns the byte-offset of `member` in an object of type `type` as a constant expression. A special C file contains `offsetof` expressions for members of a class in a specific format: MACRO(symbol name, expression). The macro expands to easily recognizable inline assembly code including the symbol name whereas the constant expression is passed into as a constant. The file is fed to the compiler for transformation into an assembly file such that the constant expressions result in integer constant literals. The output is scanned for the inline assembly lines to generate a header file with lines of `#define SYMBOL VALUE` that can then be included from assembly files.

API Portability

An operating system kernel is the layer between the hardware and applications; a microkernel is the layer between the hardware and the operating system components and applications. On its upper interface, the kernel exposes the application programming interface (API). The application binary interface (ABI) maps the API to hardware primitives available on the particular target configuration.

An API is portable when its elements (e.g., the abstractions and mechanisms it defines) appear in a sufficiently similar fashion on all target configurations. The kernel implements the API through the ABI by means of the underlying hardware. Without a portable API, code reuse in the kernel would be minimal. Hence, a portable API is a necessary precondition for a portable kernel.

2.4.3 Tools

The third source of configuration diversity in the kernel's environment is the build system, including such tools as compiler, assembler, linker, and other programs. The main reasons for this diversity are the freedom in implementation of those tools and their evolution.

A vendor and version lock-in on development tools is often not desirable and should be avoided at reasonable cost. Tool openness is particularly important for an open source project, as it lowers the bar for exploration and experimenting that come before commercial uptake. The attractiveness of "it just builds" cannot be underestimated.

Although the build system is not directly reflected by kernel code, it still affects the kernel code base. The following list briefly discusses the influence of tools diversity on the kernel.

Freedom in implementation Language specifications often leave certain aspects to the implementation of the compiler or the run-time system. Compiler vendors have used this freedom to create incompatible solutions. While a strictly standards compliant program will translate with almost any compiler, the outputs of different compilers and even different versions of the same compiler are often not compatible.

Examples for such incompatibilities are different calling conventions, locations of vtable pointers, and object layouts in general. Another example can be found in the GCC compiler tool chain: version 3 and above use a different name mangling scheme than earlier versions, with the result that code compiled with different versions cannot be linked together successfully. While a kernel is usually compiled with only one compiler version at a time, hand-written assembly code that interfaces with C++ code needs to be aware of the mangling scheme and thus of the compiler version. Name mangling schemes also tend to differ largely between compilers of different vendors.

Compiler extensions Often, compiler vendors implement particular features that go beyond the language specification. Usually, such extensions address optimizations or ease of use. An example for the latter are the `<?` and `>?` operators in GCC 3.x versions. These minimum and maximum operators avoid multiple evaluation and are thus convenient to use. Examples for extensions targeting optimization are the `__builtin_expect` branch hint expressions that allow to specify the likely outcome of an expression and thus help the compiler generate more compact and thereby more efficient code.

Increasing standards compliance Over the last years, a trend towards stricter standards compliance can be observed among compiler vendors. Vendor-specific extensions have an increasingly hard time to survive. Code that makes use of such extensions needs to be adapted to either avoid them or use them only when available.

Incompatible concurrent specifications A prominent example for incompatible specifications are the AT&T syntax and Intel syntax for the x86 assembly language. Both describe the same object, mnemonics for processor instructions, yet they are incompatible (Source and destination operands are swapped; immediate values and operand sizes are specified differently, etc.) The parts of the kernel that are written in or contain assembly language need to be aware of the syntax being expected by the assembler.

Code quality Different compilers often generate more or less optimal code from the same input. The main reasons here are different advances in compiler optimization techniques between vendors. Also a change in a compiler's internal structure may allow more optimizations in later versions. Source code that translates to optimal code with a new compiler may be all but optimal when translated with an older compiler.

3 Case Study: L4Ka::Pistachio

The L4 microkernel is a second generation microkernel that was initially developed by Jochen Liedtke at GMD, Germany, IBM Research, New York, and Karlsruhe University, Germany. He implemented early kernel versions in MASM assembly code for x86 processors. Assembly versions for Alpha and MIPS processors, L4Alpha [51] and L4Mips [43], were implemented by the Operating Systems group at Dresden University of Technology, Germany, and the DiSy group at the University of New South Wales, Sydney, Australia. Due to licensing restrictions of the x86 kernel, Fiasco [42], a C++ version for x86 processors focussing on real-time properties, was implemented by Dresden University. High-performance C++ implementations by Karlsruhe University followed. Based on the latest Karlsruhe kernel, National ICT Australia branched off a version for embedded systems.

The L4Ka::Pistachio microkernel is a reference implementation of the L4 experimental Version X.2 API. Its major design goals are IPC performance, portability, and reusability [56]. Pistachio was designed and implemented by colleagues and me at the System Architecture Research Group at Karlsruhe University, Germany. The DiSy group has contributed several ports. At the time of writing, the kernel is available for 10 architectures and 23 platforms, and supports single- and multiprocessor configurations. Pistachio is written in C++. Expensive language features such as exceptions, run-time type information, and virtual functions are avoided in performance-critical parts of the kernel.

After introducing the L4 API in Section 3.1, I discuss the L4Ka::Pistachio microkernel's approach to portability in Section 3.2, showing how it addresses the portability aspects discussed in Section 2.4. In Section 3.3, I show how class hierarchies can be used to improve Pistachio's structure by building kernel data structures from a set of fine-grained, configuration specific classes, and Section 3.4 analyzes the runtime overhead of that approach.

3.1 The L4 X.2 API

The L4 X.2 API is the latest member of a series of L4 API specifications, representing more than ten years of microkernel research. The L4 X.2 API is described in the L4 eXperimental Kernel Reference Manual [57].

The L4 API defines two major abstractions and two mechanisms:

Threads are the abstraction for activities. An L4 thread is represented by its register state (a subset of processor registers and virtual registers introduced by the L4 API), a global, unique identifier, and an associated address space. All threads bound to a particular processor are scheduled according to their assigned static priority and time-slice length.

Address spaces are the abstraction for protection and isolation. Resource permissions are managed in address spaces. L4 address spaces are no first class object; they are indirectly identified via any thread associated to this particular space. All threads in an address space have the same rights and can freely manipulate each other.

Inter-process communication (IPC) is the mechanism for data transfer and controlled execution transfer between threads. A message transfer is synchronous and involves exactly two threads. Both sender and receiver have to agree on the time of the transfer as well as on the format of the message. Messages can transport simple words and more complex descriptors that are interpreted by the kernel.

Mapping is the mechanism for transfer of resource permissions between address spaces. Access to a resource is granted by transferring resource descriptors in an IPC message, thereby enabling user-level management of address spaces. Resource permissions can be duplicated or moved. The receiver's permissions can only be a subset of the sender's permissions. Mapping can be applied recursively. Revocation of resource rights is done asynchronously through the *unmap* primitive and does not require explicit consent from the receiver of the mapping operation.

Hardware generated events such as *exceptions* and *interrupts* are translated into kernel-generated IPC messages. On an hardware interrupt, the kernel synthesizes a message to a thread that is registered as the handler for that interrupt. The sender appears to be a thread with a special per-interrupt identifier. Similarly, hardware exceptions are mapped onto an IPC based fault protocol. In the name of the faulting thread, the kernel synthesizes a message with information about the cause of the fault and sends it to the faulting thread's exception handler. The faulting thread is then set to receive a reply message from the exception handler to resume execution. The IPC fault protocol is transparent to the faulting thread; its state is preserved by the kernel. On a page fault exception, the fault message is sent to the pager of the thread, expecting a memory mapping in the reply.

Portability

The L4 X.2 Kernel Reference Manual [57] defines the generic API for all 32-bit and 64-bit machines. The generic API is as such independent of specific processor architectures and also endianess-independent. It is complemented by processor-specific ABI specifications. The manual differentiates between several interfaces of different levels of abstraction.

The *Logical Interface* defines all concepts and logical objects such as system-call operations, logical data objects, data types and their semantics. Altogether, they form the logical L4 API. Binary representations of most data types and generic data objects are defined independently of specific processors (although there are two different versions, one for 32-bit and a second one for 64-bit processors). Both versions together form the *Generic Binary Interface* of L4. For ease-of-use and standardization reasons, the two fundamental interfaces are complemented by two more interfaces: The *Generic Programming Interface* defines the objects of the logical interface and the generic binary interface as pseudo C++

classes. The *Convenience Programming Interface* makes the most common operations more easily usable for the programmer. Convenience and ease-of-use, not completeness, is the criterion for this interface. The *Processor-specific Binary Interface* is not part of the generic L4 API specification. It defines the binary kernel interface by means of processor-specific features, e.g., methods of system call invocation, protocols specific to the processor, etc.

The strict separation between API and ABI allows to write portable applications. Every complete L4 language binding has to include the entire convenience programming interface so that portable applications can use this interface to abstract from machine-specific details. The L4 API makes one fundamental assumption, though: Addresses (pointers, function entry points, etc.) are as wide as the general purpose registers of the machine and hence the IPC message word.

3.2 Implementation

The API and ABI specification together define the interface of the microkernel. As such, they define how applications talk to the kernel and how the kernel exports its services to applications. However, it is left to the implementation how the kernel provides the services defined in the specification.

Common to all L4 kernels are three major data structures: thread control blocks (TCBs) representing an L4 thread, page tables representing an address space, and the mapping database for tracking mapping operations to enable recursive revocation using the unmap primitive.

The previous section discussed L4 at run time; this section discusses L4Ka::Pistachio at build time. It describes selected aspects of L4Ka::Pistachio's implementation thereby focusing on portability. I discuss configuration management, data types, system topology, multiple programming languages, and developer tool chains.

3.2.1 Configuration Management

Pistachio achieves its level of portability through configurability. Kernel functionality has been categorized to be either generic or specific to one (or a combination of several) *configuration dimensions*. Configuration dimensions are orthogonal degrees of freedom in system configuration. In Pistachio, three configuration dimensions are defined (see Figure 3.1), and in every dimension, one or more instances are defined: processor architecture (ia32, ia64, amd64, arm, alpha, powerpc, powerpc64, mips32, mips64, sparc64), platform (pc99, efi, pleb, malta, ipaq, etc.), and API (v4, v5e). Furthermore, combinations of instances of different dimensions can be defined (e.g., v4-ia32, v5e-arm, etc.)

Code (and declarations) for a particular instance of a configuration dimension or a combination is placed in its respective subdirectory. Code that differs for aspects not defined as configuration dimensions is either placed in different source files or surrounded by directives for conditional compilation.

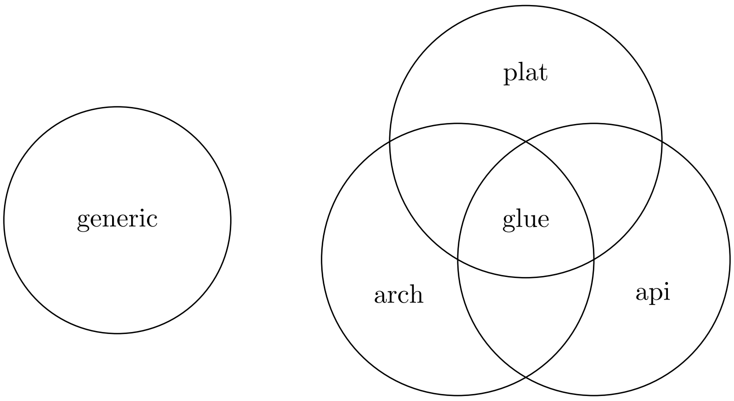


Figure 3.1: Pistachio distinguishes three configuration dimensions: the architecture, the API, and the platform. Code is either generic or it is specific to one or a combination of several configuration dimensions.

Pistachio uses the CML2 configuration management system [46]. The CML2 language allows to define menu items (booleans, one-of, numeric or text inputs), rules for visibility of items, and rules for derivations of values. The output of CML2 is a set of variable-value pairs. The CML2 system consists of three Python [50] programs. Run interactively, it presents a hierarchy of menus to the user to make selections. Run from a script it accepts command line arguments as input selections.

CML2 emits configuration information in several files and formats. One file contains lines of the form `VARIABLE=VALUE`. Another file contains `#define VARIABLE VALUE` and `#undef VARIABLE` directives that can be evaluated by the C++ preprocessor. In Pistachio, the first file is sourced by the build system to allow for the configuration-dependent selection of source files and configuration-dependent specification of options for tools. By use of a compiler command line option, the second file is read by the preprocessor before any source file is being parsed. Hence, configuration information is present in any source and header file of the kernel and can be used to control conditional compilation.

Often, code specific to one instance of a configuration dimension needs to refer to code or data from another configuration dimension without knowing the exact instance in that other dimension. For example, API-specific code may need to invoke an architecture-specific function. Pistachio uses *computed includes* to select the files that contain the desired functionality. With computed includes, the path name for an `#include` directive is produced by a macro: For example, the directive `#include INC_ARCH(foo.h)` will include the file `include/arch/ia32/foo.h` if the architecture has been set to `ia32`. Like configuration information, the necessary macro definitions are read by the compiler before any source file.

3.2.2 Data Types

In Pistachio, the kernel's type system is mostly decoupled from that provided by the compiler. The following basic data types are available to kernel code:

s64_t, s32_t, s16_t, s8_t, u64_t, u32_t, u16_t, u8_t are signed and unsigned integer types of the respective word width as indicated by the name. These types are guaranteed to not change across configurations.

word_t is an unsigned integer of the processor's general purpose register size. The word width changes with the architecture, i.e., it is 64 on 64-bit architectures and 32 on 32-bit architectures.

addr_t is a single-byte granular pointer type for safe address calculations. Since the L4 address space model has not yet been extended to handle physical address spaces that are larger than the virtual address space of the machine, kernel code does not support physical and virtual addresses of different sizes. They are assumed to be of equal width.

Above types are defined per architecture based on the types provided by the compiler. Details of compiler-specific type definitions (like special modes for large data types on narrower architectures) are thus effectively hidden from the kernel programmer. The data types are available in all C/C++ header and source files by inclusion of the architecture-specific header file via the compiler's command line.

As described in Section 2.4.1, the mapping of bit fields to the bits of the containing word depends on the endianess of the system. In Pistachio, a set of preprocessor macros hides this detail from the programmer. Depending on the endianess of the system (which is published as a derived configuration variable), the sequence of the fields in the declaration is reversed or not, resulting in the first field always containing the least significant bit of the containing word.

3.2.3 System Topology

Pistachio runs on uniprocessor and multiprocessor systems, supporting the following types of the latter: simultaneous multithreading (SMT), multi-core, symmetrical multiprocessing (SMP), as well as cache-coherent nonuniform memory access (NUMA), or some combination thereof.

Depending on system configuration and scalability requirements, a kernel object may exist per system, per node, per physical processor, per core, or per logical processor (hardware thread). Often, code accessing such an object may not need to be aware of the differences and can thus benefit from hiding them.

An example for such an object is the scheduler in Pistachio, of which one exists for every logical processor. Code invokes "the scheduler" by first calling the `get_scheduler()` function to obtain a reference to the appropriate scheduler object which is then used for the actual invocation. The `get_scheduler()` function could return a reference to the global scheduler object in a uniprocessor configuration or determine the current processor number to index into an array of scheduler objects. Pistachio, however, places the seemingly global (from the C++ language's point of view) scheduler object in a region of virtual memory that maps to different physical memory for every logical processor, referred to as the processor-local region. The indirection through the MMU eliminates the need for identifying the current processor number, independent of the actual configuration (single or multi-processor). The code can become very clean as it needs not be concerned about the level of parallelism in the system.

By the use of a macro at the place of definition, global objects (in the sense of C++) can be placed into particular data sections in the kernel's virtual address range. Some of these regions are backed with node-local (`.data.nodelocal`) or processor-local (`.data.cpulocal`) memory. Furthermore, static initialization of "global" objects in such regions is repeated for every node or logical processor whereas global objects not marked specifically are initialized once by the boot processor.

3.2.4 Mixed Programming Languages

Some parts of the kernel must be implemented in assembly language, such as accesses to control and status registers via special instructions. Another example are the entry and exit stubs for handlers of system calls, exceptions, and interrupts.[1] A kernel programmer may also choose to provide hand-optimized assembly implementations of common, performance-critical code paths.

In Pistachio, special instructions are wrapped into inline functions that contain inline assembly code. Likewise, the stack-based thread switch is encapsulated in an inline function. System call and exception handlers reside in assembly files in the respective architecture-specific directory. For several architectures, a hand-optimized assembly version of the common case IPC path exists, the is so-called IPC FastPath. Functions in C++ that are invoked from assembly code, such as the C implementation of the IPC path, are declared `extern "C"` to disable name decoration.

In all these cases, assembler code frequently references kernel objects defined in the high-level language. A compiler generates the correct field offsets relative to the base address of an object, and it continues to do so after the programmer changed the data structure declaration. Hand-crafted assembly code, however, is not updated automatically. Thus, unsafe references from assembler code to data structures defined in the high-level language are acceptable only if field offsets are not hard-coded but automatically derived.

Initially, the Pistachio build process generated a C file with constant expressions as initializers for elements of an array that was then compiled with the target compiler. The array elements were then extracted from the resulting object file to create a header file. However, recent compilers chose to generate code for initializing the array at run time

[1] Some compilers, e.g., GCC for the ARM, AVR, M68K, SH, and H8 architectures, provide extensions that generate prologue and epilogue of functions such that they can directly be used as interrupt handlers.

and thus rendered that approach unusable. Therefore, Pistachio uses a machine-generated header file that contains the appropriate offsets, as described in Section 2.4.2.

3.2.5 Tools

As an open-source project, Pistachio needs to build with popular open source development tools. Being able to build the kernel for the most popular target architecture, x86, with almost any version of the freely available GNU toolchain (GCC, Binutils, Make) has proven to be an invaluable property because it radically lowers the bar for experimentation and thus improves acceptance in the community of potential customers. Requiring some (arbitrary) version of an easily available toolchain is generally accepted whereas requiring a particular version of such a toolchain or a patched, custom-built version are less favorable. Yet, the more exotic the target architecture is, the more acceptable becomes the requirement for a particular version of the toolchain.

With the wide availability of the GNU toolchain on almost any *NIX-like operating system (or execution environment such as Cygwin [47]), requiring GCC (or compatibles) for building Pistachio has not raised any problems. Requests have been made to support the Intel C++ compiler; that work is underway. In an earlier experiment with the Hazelnut kernel, supporting Microsoft's C++ compiler for building user-land applications has not been positively acknowledged once.

The evolution of the GNU toolchain has left its marks in the Pistachio kernel code base. The trend towards stricter C++ standards compliance usually demands code that can also be built with a less strict (i.e., earlier) version. Newly introduced or removed compiler extensions, however, need to be either avoided completely or wrapped in macros whose definition depends on the compiler version. An example of such an extension is `__builtin_expect(expr, value)`, giving hints to the compiler about the most probable value of an expression (often used with `if` and `switch` statements.) It is wrapped in the `EXPECT_TRUE(expr)`, `EXPECT_FALSE(expr)`, and `EXPECT_VALUE(expr, value)` macros. If the extension is not available, these macros simply evaluate to their first argument.

3.3 Improving L4Ka::Pistachio with Inheritance

In terms of performance, Pistachio has fulfilled expectations: all assembly versions of L4 were discontinued soon after Pistachio's availability. They had been maintained in lack of alternative similarly efficient implementations in a high-level language. However, performance is only one of the design goals of Pistachio; portability and maintainability are the other two.

A general rule of thumb asserts that a project that was created by a single person with an effort of 1 can be made available to a small group in-house with an effort of 3 and to the public with an effort of 10. Pigoski [45] attributes approximately 80 percent of the time invested in a long-term software project to maintenance, of which 50 percent are spent reading and trying to understand the source code.

The source code infrastructure of the Pistachio microkernel has very well served its purpose of allowing to quickly build a portable microkernel for multiple architectures. Measured by the time required for porting, Pistachio can be considered highly portable: For example, the ports of Pistachio to the Alpha and SPARC64 processors were reportedly completed within just two weeks. The Pistachio code base has also been (ab)used as an OS construction kit, allegedly because of its carefully crafted and easily understandable data structures for x86 hardware as well as its lean, easily extensible structure.

Nevertheless, Pistachio can still be improved, especially in terms of configurability and code reuse, which in turn can improve performance and readability. The file-based selection of code limits the granularity at which code can be selected and configured. This problem is being worked around by unnecessary code duplication or excessive preprocessor use. Either the definition of data structure or a function is replicated for all configurations to allow one configuration to be implemented differently, or excessive use of preprocessor directives such as `#ifdef`, `#else`, and `#endif` is made to keep all variants in the same location, or text fragments of the definition are placed in separate files and combined using `#include` directives somewhere amid the definition. Readability of the alternatives severely decreases in the order listed. Source browsers, humans and especially machines, often have severe problems telling apart the many definitions for a class or function under the same name that coexist in different paths of a source tree.

An approach somewhere between these extremes defines “one-fits-all” data structures that contain the joint set of all members required by all configurations. However, such data structures potentially waste cache performance. With access characteristics for members differing across configurations, the object layout can be optimal for only one configuration. An example for such conflicting optimizations is Pistachio’s TCB class `tcb_t` that has been optimized for the `ia32` port, but the MIPS64 port developed by Nourai [40] requires several additional data members and a rather different layout for optimal IPC performance.

Furthermore, the ability to share code between configurations needs to be refined: For example, the `amd64` port and the `ia32` port bear so many similarities that large parts of the code could be shared. However, the `amd64` port was created by copying and modifying the `ia32` port. Merging the two ports has long been considered impossible with current the directory structure, and adapting the structure to allow the merge was beyond the scope of the `amd64` port. The result is increased maintenance, because bug fixes to common parts need to be applied to both ports separately. Even worse, some bugfixes on one side were not carried over to the other and thus led to diversions that will make merging harder.

3.3.1 Configuration-specific class composition

Pistachio’s structural problems described above can be solved by using inheritance for object composition as follows. The class of a kernel object is composed from multiple classes that each encapsulate functionality (and the associated data) specific to an instance of a configuration dimension or configuration feature, as shown in Figure 3.2.

The classes form a class hierarchy with the most derived class representing the actual kernel object. Throughout the hierarchy, implementations can be refined in more derived

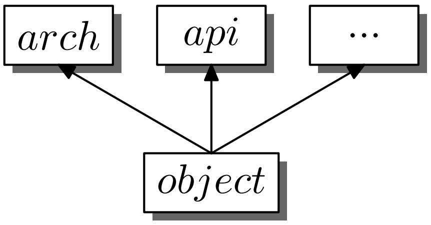


Figure 3.2: Kernel objects are composed from a classes that each encapsulate functionality and data specific to an instance of a configuration dimension.

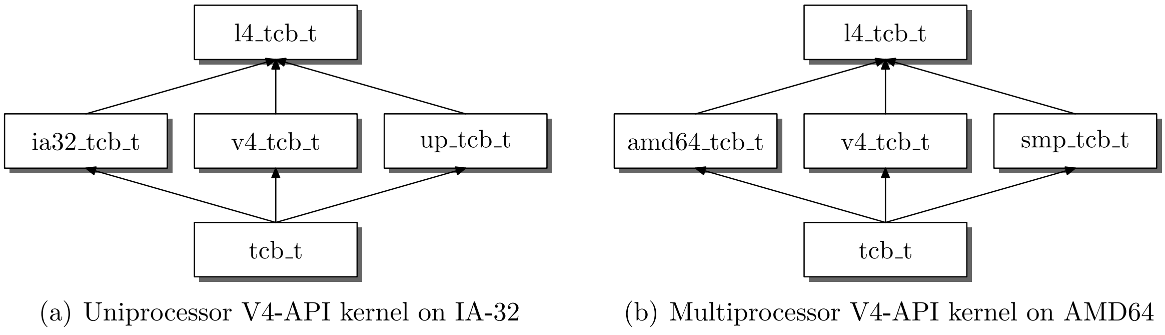
classes: member functions can be overridden and data members can be hidden with more suitable variants. In the most derived class direct as well as inherited members and methods can be accessed without knowing where in the hierarchy they are implemented. The internal structure of the most derived class becomes irrelevant to the code using the class.

There may also be superclasses that are specific to a certain combination of configuration dimensions. These would inherit from classes for each of their configuration dimensions and be in turn a base class for even more specific subclasses or the most derived class.

Inheritance relationships between the classes that form the hierarchy are derived from the target configuration. Thus, for different configurations, the hierarchy contains different superclasses. The most derived class, however, has a stable interface across all configurations. Only this most derived class is used throughout the kernel to instantiate and access objects of the class.

Functionality implemented in one configuration-specific superclass may need to invoke functionality implemented in another configuration-specific superclass. To allow methods to invoke methods implemented somewhere else in the hierarchy a common, top-level base class must be defined that declares the interface of the final, most derived class.

Figure 3.3 illustrates configuration-specific class composition for kernel objects by a simplified example of the thread control block (TCB) class `tcb_t`.



(a) Uniprocessor V4-API kernel on IA-32

(b) Multiprocessor V4-API kernel on AMD64

Figure 3.3: Two simplified examples of the `tcb_t` class hierarchy

Class `tcb_t` inherits part of its implementation from an architecture-specific mix-in (`ia32_tcb_t` or `amd64_tcb_t`), an API-specific mix-in (e.g., `v4_tcb_t`), and another mix-in that is either uniprocessor-specific (e.g., `up_tcb_t`) or multiprocessor-specific (e.g., `smp_tcb_t`).

3.3.2 Class Properties

Classes composed from a hierarchy of configuration-specific subclasses as developed in the previous section have the following properties:

- The class hierarchy has a common virtual base class at its root. At a minimum, this base class declares the interface between the configuration-specific mix-ins as pure virtual functions.
- Configuration-specific mix-ins add missing or specialize existing functionality. They also add the data members required for implementing this functionality.
- Neither the base class nor the mix-ins need to be complete classes and thus cannot be instantiated.
- The most derived class is not abstract. It is the only class in the hierarchy that can be instantiated.
- Methods should be declared virtual so as to allow overriding at arbitrary places in the hierarchy.
- Every method declared in the base class has at least one implementation in the class hierarchy. If there is more than one implementation, there must exist exactly one final overrider such that no ambiguities arise.
- Data members must have unique names throughout the class hierarchy, unless there is one final shadower that ensures no ambiguities arise.

The purpose of the class hierarchy is to flexibly compose the most derived class. The class hierarchy changes from configuration to configuration. Yet, independent of where and how functionality is implemented inside the class hierarchy, the interface of the most derived class remains stable and only this class is to be used by other code.

Code using the class must not make any assumptions about the internal structure of the class, i.e., it must not even assume the class was actually composed from a class hierarchy. Hence, code must refer to all members of the class without base class specifiers. There are also no base classes this class could possibly be compatible with.

3.4 Inheritance-related Overheads

Two types of run-time overhead can be attributed to inheritance: the overhead of indirect calls to implement polymorphic function invocations, and the overhead of a suboptimal object layout such as dictated by the inheritance relationship. In this section, I show the performance implications of introducing a class hierarchy for performance-critical classes in the Pistachio microkernel. I conducted two isolated experiments focusing on one type of overhead each.

Performance of the microkernel is evaluated by measuring IPC performance with the `pingpong` IPC microbenchmark. Two threads send messages back and forth and measure the round trip time. Those threads can reside in the same address space, in different address spaces, or on different processors. Throughout the benchmark, the length of the messages is increased. For each message size, the benchmark reports number of processor cycles for a single message transfer (i.e., half the cycles for a round trip) averaged over a large number of transfers.

3.4.1 Virtual Function Calls

To determine the performance impact of using virtual functions on the kernel's critical path, I modified the flat class `tcb_t` to be a class hierarchy with a virtual base class and a mix-in class, as shown in Figure 3.4.

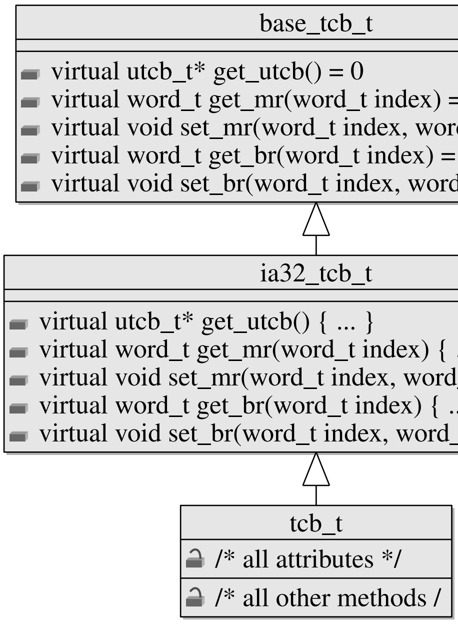

base_tcb_t
virtual utcb_t* get_utcb() = 0
virtual word_t get_mr(word_t index) = 0
virtual void set_mr(word_t index, word_t value) = 0
virtual word_t get_br(word_t index) = 0
virtual void set_br(word_t index, word_t value) = 0
ia32_tcb_t
virtual utcb_t* get_utcb() { ... }
virtual word_t get_mr(word_t index) { ... }
virtual void set_mr(word_t index, word_t value) { ... }
virtual word_t get_br(word_t index) { ... }
virtual void set_br(word_t index, word_t value) { ... }
tcb_t
/* all attributes */
/* all other methods /

Figure 3.4: Class `tcb_t` with several virtual functions, which are frequently called during an IPC operation

I declared the architecture-specific, performance-critical member functions `get_utcb`, `get_mr`, `set_mr`, `get_br`, and `set_br` in the base class as pure virtual functions and moved their definition from the most derived class into the mix-in class. These functions are called several times on the IPC path. All data members remained in the most derived class. The compiler generated a vtable pointer at the start of the `tcb_t` object. To ensure that the mapping of referenced data members to cache blocks did not change, an unreferenced member found in the first cache block was temporarily moved to the end of the structure to make room for the vtable pointer.

Figure 3.5 shows IPC performance of an empty message transfer between threads in the same address space for two different configurations: the unmodified kernel (*baseline*) and the kernel with the `tcb_t` class hierarchy (*hierarchy*). Measurements were performed on an Intel Pentium 4 processor with 2.8 GHz. The combined hardware costs for entering and leaving the kernel are indicated as the dark parts of the bars.

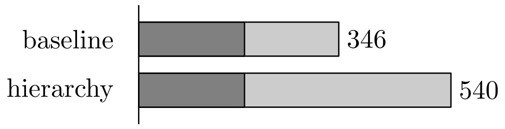


Figure 3.5: Introducing a class hierarchy effectively doubles the execution time (in processor cycles) of the IPC path in the kernel.

The introduction of virtual functions adds an absolute run-time overhead of 194 processor cycles to an IPC message transfer operation. This amounts to 56 percent overhead in total, or to 120 percent overhead in software. The overhead can be attributed to the non-predictable indirect calls used for invoking virtual functions.

3.4.2 Object Layout

To evaluate the influence of inheritance-dictated object-layout on microkernel performance, I modified the `tcb_t` class to inherit most of its data members from various base classes, as shown in Figure 3.6. No functions were moved to base classes, and no member functions are virtual functions.

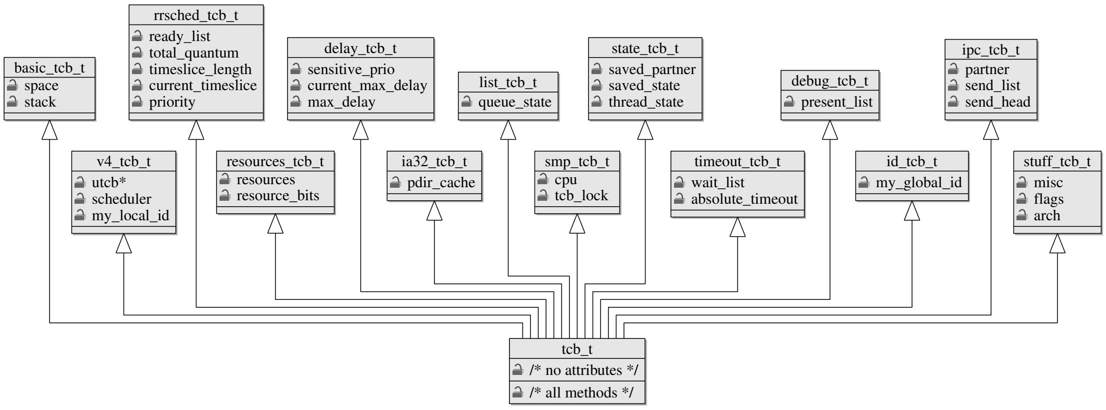


Figure 3.6: Class `tcb_t` inherits data members from several base classes.

The placement of data members in base classes follows a logical structure, i.e., debugging-related members are placed in `debug_tcb_t`, multiprocessor-related members reside in smp_tcb_t, etc. The placement of members in base classes also determines their placement in the object (see Section 2.2.2.) The access behavior of operations on the derived class, however, does not reflect the logical structure but is of a cross-cutting nature. Consequently, the data members that are referenced on the IPC path are spread across the object. Figure 3.7 visualizes the distribution of referenced data members for two tcb_t objects involved in an IPC operation.

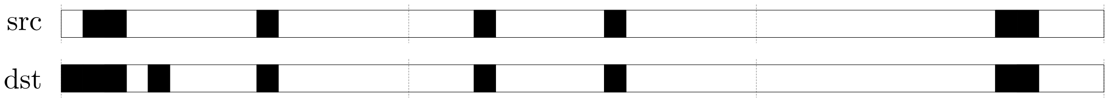


Figure 3.7: Two objects of class tcb_t — memory locations accessed on the IPC path are marked black. Addresses increase from left to right. Thin vertical lines indicate 64-byte cache-block boundaries.

The 28 bytes of the seven referenced data members in the source TCB and the 36 bytes of the nine referenced data members in the destination TCB would fit in one cache block per object. Instead, they are spread out over three cache blocks. As this applies to both sender and receiver TCB, the cache footprint is four cache blocks larger than optimal. The influence of this subobtimal object layout on IPC performance is shown in Figure 3.8.

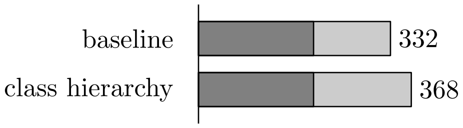


Figure 3.8: IPC performance of a baseline kernel and a kernel with a suboptimal object memory layout due to inheritance.

The run-time overhead for an empty IPC message transfer between two threads in the same address space amounts to 36 cycles or 11 percent.

In the IA-32 instruction set, offsets smaller than 128 Bytes can be encoded with an 8-bit displacement, whereas larger offsets require a 32-bit displacement. Due to different offsets of tcb_t data members in the two kernels, the assembler used different addressing modes for some instructions accessing those members. Hence, the length of those instructions varies. For the experiment, however, the instruction count and the instruction cache footprint of the IPC path remained constant.

4 Eliminating Portability Overheads

In the previous chapter I showed how inheritance can be used to improve the software structure of the Pistachio microkernel along with experiments that hint at the run-time overheads to be paid when doing so. These overheads have always been a popular argument against the use of inheritance in performance-critical scenarios such as microkernels.

For simple classes, field reordering can maximize spatial locality by rearranging fields according to access characteristics. However, this does not apply to classes composed using inheritance, because the field order is dictated by inheritance. Class flattening transforms derived classes into simple classes and thereby makes those classes suitable for field reordering. It also removes the overhead of virtual functions. *A key contribution of this work is to combine class flattening and field reordering to optimize class hierarchies for maximum performance on a kernel's critical path.* Field reordering for kernel objects is driven by field access patterns that are best gathered through profiling. The process of collecting this profiling information can be aggressively tailored towards optimizing microkernel objects by incorporating knowledge about microkernel peculiarities, generating complete yet compact information that allows to find an optimal reordering.

The remainder of this chapter describes my approach to completely eliminate the run-time overheads of inheritance in a microkernel. Section 4.1 introduces the optimization methodology in detail. Section 4.2 discusses class flattening as an enabling technology for field reordering of class hierarchies. Section 4.3 presents previously unconsidered strategies for field reordering, some of which are specific to kernel data structures. Section 4.4 describes a customized approach to determining field access characteristics to drive field reordering. An example field reordering algorithm using such collected access characteristics is sketched in Section 4.5. Section 4.6 shows how transparent flattening and field reordering work in concert as a transparent optimization in the build process.

4.1 Optimizing Performance-Critical Classes

The memory layout of compound data structures such as structs and classes determines the cache footprint when accessing their fields. For a microkernel, a small cache footprint is crucial for performance of the kernel and of the system as a whole. Poor cache usage on the kernel's critical path results in costly cache misses in the kernel and once more in the application to reload the evicted data.

In the light of the high number of possible target configurations of a portable microkernel, optimizing the memory layout of critical kernel data structures by manually reordering their fields in the class declaration is a tedious task and the result is not portable. Fields accessed on one architecture may remain unreferenced on another architecture. Similarly, one algorithm may access fields that another, alternative algorithm does not access. Furthermore, the set of fields accessed may also depend on the input set, that is, on the behavior of the applications on top of the microkernel.

The set of fields that are referenced on the kernel's critical path depends on several factors. The access sequence to these fields also depends on the quality of the compiler and the optimization level: redundant loads may be avoided; independent field accesses may be performed in different order; and field accesses may be merged or split.

On top of that, classes composed using inheritance expose properties that make field reordering impossible altogether, as the placement of fields inherited from base classes is governed by the inheritance relationship. However, composing classes from a number of small configuration-specific superclasses is an elegant solution to the problem of code selection in a portable microkernel. Code duplication can be avoided and portability and maintainability are improved.

All overheads associated with inheritance, sub-optimal cache usage and indirect function calls, can be eliminated with a combination of class flattening and field reordering. Class flattening converts a class hierarchy into a single, flat class, thereby removing inheritance and thus virtual functions and any notion of subobjects. Subsequently, field reordering can freely arrange the fields in a cache-optimal way.

The mapping of fields to memory addresses and thus to cache blocks depends on the order in which the fields appear in the class declaration, the inheritance relation, and on the size of the cache blocks, which in turn may vary among different members of an architecture family or even different implementations of the same processor.

The vast number of possible constellations and resulting optimal field placements calls for an automated approach to optimizing performance-critical kernel data structures. The optimization should be transparent to the programmer; a kernel developer should focus on designing efficient algorithms and reusable code (and specialized, optimized code where appropriate), but not fiddle with internal layouts of data structures or the cost of function calls, both of which are details of the language implementation.

Field reordering requires information about field access patterns to arrange fields in a cache-optimal way. Since access patterns depend on many factors including the workload running on the kernel, the best time for precisely determining these patterns is during the execution of the kernel in the target system. The field reordering optimization is driven by feedback from the kernel being executed. Initial execution of the kernel under load generates field access information that can be used to reorder fields of performance-critical classes in subsequent, optimizing kernel builds. Knowledge about the peculiarities of microkernels allows to tailor the process of collecting access patterns to the task of field reordering, in particular it enables low-footprint yet full-detail traces of the critical path to be captured that are a requirement for finding an optimal field order [16].

Safety of Transformation

In the closed code biotope of a microkernel, both class flattening and field reordering can be automated if certain conditions are met: Flattening is a safe and transparent transformation when inheritance is used only as a means to efficiently compose the most derived class and code using the class does not make any assumptions about the class beyond its interface. Such a requirement is also strongly supported by the principle of encapsulation.

Field reordering is safe for classes that are not externally visible (data types for API elements), that do not interface with predefined structures (page tables, descriptor tables, etc.), and where no code makes static assumptions about their internal layout. Code that automatically adapts to changes in internal layout is unproblematic.

Whole-program Optimization

Both class flattening and field reordering are whole-program optimizations by nature. All parts of the program must use the same definition of a class to produce compatible objects. For manageability reasons, however, most larger C++ programs — and that includes microkernels — are broken down into many source files that are then translated separately.

Nevertheless, separate compilation does not preclude such whole program optimizations. As long as those optimizations are deterministic and applied equally to the same classes in all translation units, they should produce the same results. Hence the effect of a whole program optimization is kept and objects of optimized classes in separately translated files remain compatible.

Location of Optimization

Class flattening and field reordering can both be applied in different places: outside or inside the compiler. Both techniques could be added to optimization frameworks already present in todays compilers and leverage the existing infrastructure. For example, GCC can use basic block profiling data generated with an instrumented version of a program to optimize branches and place basic blocks optimally during subsequent compilations of the program.

The downside of integrating the techniques into a compiler is that — unless they become standard optimizations — they require the use of a modified compiler. Standard compilers are often preferred over custom-build compilers. Modifying commercial compilers may not be trivial due to lack of source access. The alternative to integrating both techniques into the compiler is to implement them as source-to-source transformations that are applied before compilation. Most build environments allow for such processing to be performed before compilation.

By keeping class flattening and field reordering separate from the compiler, the optimization is not tied to a particular compiler and can be combined with nearly any build system. Therefore, the optimization described in this work performs both class flattening and field reordering as source-to-source transformations.

4.2 Transparent Class Flattening for Field Reordering

Inheritance defines an “is-a” relationship between a derived class and a superclass: An object of the derived class can be treated as an object of the superclass. This implies that fields inherited from a superclass form a subobject of the superclass within the object of the derived class and that this subobject has the same layout as an object of the superclass. Consequently, the freedom of placing fields in an object of a derived class is restricted by the inheritance relationships used to compose the class.

When a derived class is flattened, however, all fields become direct members of the class. The memory layout of a flattened (i.e., simple) class is directly related to the order in which the fields appear in the class definition and is therefore suitable for subsequent field reordering. In this sense, class flattening acts as an enabling technique for field reordering of classes composed with inheritance.

This work proposes *transparent* class flattening, which *replaces* the original definition of a class with its flattened version. The scope of the flattening transformation is thereby limited to the class definition itself. No use place of the class must be transformed, which greatly reduces the complexity of the flattening algorithm. The direct benefit is that no code using the class needs to be adapted to use the flattened variant of the class, and that existing confidence in that code is maintained.

Section 4.2.1 describes transparent class flattening and Section 4.2.2 states the preconditions that enable transparent flattening. Section 4.2.3 discusses fidelity aspects of class flattening whereas Section 4.2.4 shows how the restrictions that enable transparent flattening can be enforced.

4.2.1 Transparent Flattening

All instances of class flattening in previous work create a second, flattened variant of the class definition, which coexists with the original. To make use of the flattened variant, code must be modified to refer to the new class name. Variable declarations must be changed to the flat class and functions with a parameter of the class’ type must be duplicated to also accept the flattened class.

In contrast, transparent class flattening *replaces* the original class definition with the flattened variant. The original and the flattened version of the target class expose the same interface — under the same class name. Behavior with respect to this interface does not change either. Thus the flattened variant can be used in place of the original class. The fact that the class has been flattened is *transparent* to code using the class.

Unsurprisingly, the mechanics of (traditional) and transparent class flattening are very similar. The main difference is that inherited members are copied into the source class instead of a newly created target class. Furthermore, the inheritance specification is removed from the most derived class’ definition. Figure 4.1 illustrates the transformation of a class by transparent class flattening.

As flattening removes inheritance, there is no need to declare member functions as virtual functions in the target class. The keyword `virtual` is removed during flattening. As a

```
class A {
public:
    int a;
    void f() { /* A::f */ };
    void g() { /* A::g */ };
};

class B : A {
public:
    int b;
    void g() { /* B::g */ };
    void h() { /* B::h */ };
};
```

(a) Class hierarchy for B

```
class A {
public:
    int a;
    void f() { /* A::f */ };
    void g() { /* A::g */ };
};

class B {
public:
    int a;
    int b;
    void f() { /* A::f */ };
    void g() { /* B::g */ };
    void h() { /* B::h */ };
};
```

(b) Class B flattened transparently

Figure 4.1: Transparent class flattening modifies the most derived class. Inherited members visible in the most derived class are copied into the class and the inheritance specification is removed.

result, the compiler can generate direct calls to member functions and even inline them. The runtime overhead of virtual functions is thus eliminated. Likewise, the compiler will not include a vtable pointer in the object, eliminating the space overhead of inheritance.

Definitions of base classes can be deleted after flattening if they were used exclusively to compose the most derived class.

4.2.2 Preconditions

Not all classes are suitable for class flattening, though. The preconditions that enable transparent class flattening are as follows:

No dynamic polymorphism Objects of the flattened class are not compatible to objects of a former base class. Thus, objects of the flattened class must not be treated as objects of a base class.

No ambiguities Ambiguous members would result in conflicting definitions after flattening that can no longer be resolved by identifying the base class the member was inherited from. Thus, the inheritance hierarchy of the target class must not contain ambiguous members.

No access to hidden members Flattening removes members that are hidden by a definition in a more derived class. Thus, code must not access hidden members through identification of the respective base class.

No partially virtual base classes If a base class appears multiple times in the inheritance hierarchy of the target class, it must always be a virtual base class. Otherwise, flattening would result in conflicting definitions.

A typical scenario that meets above conditions is when inheritance is used for composition of classes, i.e., with abstract hierarchies.

To enable combined class flattening and field reordering for selected classes in the Pistachio microkernel, the class structure introduced in Section 3.3.2 to improve Pistachio's portability fulfills above preconditions. By design, those classes are composed and to be used so as to ensure they can be flattened transparently.

4.2.3 Flattening Fidelity

Transparent class flattening replaces the hierarchical definition of the target class with an equivalent, flat definition. Equivalence is maintained as far the interface of the class is concerned, whereby the interface consists of member functions and fields that are accessible without further qualification (such as scope operators.) Member functions will produce the same results and side effects (within the language) in the flat class as they did in the original class. To this extent, both versions of the class expose identical behavior.

Former base classes of the flattened class could be removed completely if they are not referenced anywhere else in the code. Automating this step would require analysis of the complete code including inline assembly fragments. It should be noted, however, that removing unused base classes is not an inherent part of transparent class flattening.

Valid C++

Although it may seem obvious, the class must be a valid C++ class prior to flattening. Transparent class flattening transforms a most derived class in a class hierarchy into a semantically equivalent, flat class. This transformation can only produce acceptable output only if the input is acceptable, too.

Furthermore, allowing a class to be valid only after flattening would defeat one of the purposes of introducing powerful, well-known programming techniques in the microkernel to enhance the software structure for improved readability and maintainability. Developers would additionally have to understand the effects of class flattening, thereby adding to the complexity of microkernel construction instead of reducing it.

Visibility

The lookup algorithm of C++ defines the visibility of entities. Entities can be hidden behind other entities with the same name that the algorithm considers earlier. For class flattening, two cases are relevant.

In a class hierarchy, inherited fields and functions can be hidden. Hidden fields are shadowed by fields with the same name defined in a more derived class. Hidden functions are overridden by functions with the same name, not only the same signature, further down in the hierarchy.

In objects of a class constructed from superclasses, inherited yet invisible fields continue to exist in the subobjects of the base class in the derived class. They can still be accessed via explicit qualification through the scope operator referring to the base class and so can hidden functions. Class flattening removes base classes and thus the possibility to refer to members of base classes. Hidden members become inaccessible and can therefore safely be removed. They also need not exist for compatibility of subobjects, because the target class does not inherit any subobjects.

Access Control

Access to an inherited member is governed by the access specifier (`private`, `protected`, `public`) preceding the member declaration in the class where it is defined and by all access specifiers in the base class specifiers of subclasses.

Access to members in the flattened class must be at least as good as in the original class. Improved access is, however, tolerable: The code must be valid with and without flattening and so cannot rely on potentially improved access after flattening.

Taking this to the extreme, class flattening can remove all access specifiers and mark all members `public` without loosing protection. Consequently, data member declarations in the flattened class are not separated by access specifiers. This is important for the subsequent field reordering stage as the allocation order of fields is only defined to be sequential within the reach of one access specifier, and implementation-defined otherwise.

4.2.4 Enforcing Restrictions

The preconditions that enable transparent class flattening impose restrictions on the structure of target classes. In a microkernel, these restrictions can be enforced easily. The key is to build the kernel twice, with and without class flattening.

Building a working kernel without the optimization ensures that the input of class flattening is valid C++ according to the language definition of the compiler being used. Here, the tendency of most compilers toward the C++ Language Specification [25] is very helpful. After flattening, the compiler will flag most violations of the described restrictions as errors in the source, as they manifest in invalid C++ constructs. Any remaining cases can be detected and flagged by the flattening process.

Any polymorphic use of the target class by client code will result in incompatible assignments after flattening, because the base class is no longer a base class of the target class. Uses of the scope operator in client code will result in dangling base class references, because the flattened class does not have any base classes. Ambiguous fields and functions with the same signature would result in invalid code (redefinition) and are flagged by the compiler after flattening. However, functions with the same name but different signatures would appear as overloads of the name and may result in unexpected behavior instead of a compile error. Ambiguous method definitions must therefore be detected by the flattening process. The flattening process must also flag partially virtual base classes.

4.3 Field Reordering Strategies

This section describes various factors driving field reordering that have not been considered by previous work. These factors lead to higher optimization potential or can simplify the field reordering algorithm. Not all factors are necessarily applicable at the same time, depending on hardware configuration and usage scenarios. The resulting strategies may also, at least partially, contradict each other.

For objects that span multiple cache lines, the mapping of fields to these cache lines depends on the location of the field in the object and the location of the object relative to the cache line boundaries. Since the offset of a field is the same for all objects of a class, object placement/allocation is a key concern for field reordering. Three reasons support well-controlled placement of objects.

First, when objects can be allocated at arbitrary addresses, the location of cache line boundaries within the object is unpredictable. Packing all referenced fields tightly will produce a layout that, in the best case, maps these fields to the minimum number of cache lines. In the worst case it maps the fields to one cache block more than the minimum. The minimum can only be achieved reliably when objects are aligned at cache line boundaries, or at a constant offset to them.

Second, alignment requirements of fields determine the alignment requirement of the containing object. To ensure proper alignment of a field in an object, the compiler places the field at an offset that is a multiple of the field's alignment requirement and demands that the object is aligned accordingly.

Third, 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. This technique is very popular in kernels. For example, the L4 kernel uses the memory behind a TCB as per-thread stack space. A mask operation yields the base address of the current thread's TCB from the value of the stack pointer register. Similarly, Linux maintains a thread's kernel stack as part of the per-thread `task_struct` object. The `task_struct` for the currently executing thread can efficiently be found without any memory reference by masking the stack pointer. Therefore, aligning objects at their size (or the next higher power of two) is beneficial.

Alignment of objects at cache block boundaries can result in internal fragmentation. For objects spanning multiple cache lines, which are the focus of this work, fragmentation should be rather small. There is certainly a trade-off to be made between memory consumption and cache performance. In a microkernel, however, the performance gained by aligning objects at cache line boundaries may well make up for the waste of memory.

Aligning objects on at least cache block boundaries allows precise placement of fields to minimize cache footprint.

4.3.1 Object Roles

Previous work does not distinguish objects of the same class when reordering fields. Based on the observation that objects of a class show similar access behavior, statistics used to drive field reordering for a class are built from all accesses to all objects of the class. This observation certainly holds true for programs that operate on a large number of objects such as on nodes in a tree.

A microkernel, however, typically manipulates only very few objects during its short, performance-critical operations, and not all instances of a class referenced during an operation necessarily have identical access characteristics. Depending on the role of the various instances, the access pattern to fields in objects of the same class may be very different. Consequently, the sets of accessed fields may differ largely across instances, justifying separate treatment of instances for field reordering.

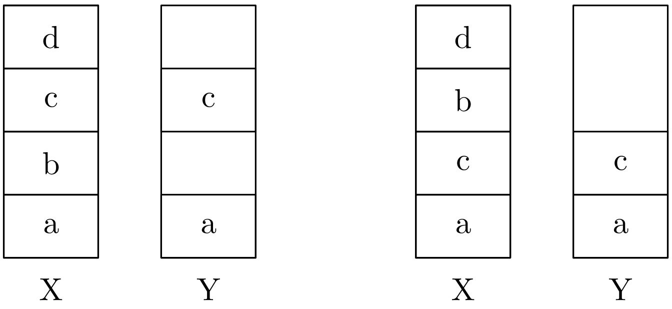


Figure 4.2: Although objects of the same class have a common layout, the cache footprint for accessing them can be minimized by field reordering if the objects' access patterns are considered separately.

The extreme but illustrative example in Figure 4.2 shows the effects of keeping objects separate. Assume an operation on two objects, `X` and `Y`, of the same class references fields `a`, `b`, `c` and `d` in the first object and only fields `a` and `c` in the second object. For purpose of illustration, the fields be each half a cache line in size and accessed in alphabetic order. A straightforward approach would pack fields in sequential order `a`, `b`, `c`, and `d`. The resulting layout will use two cache lines for each object. Differentiating between objects, the fields could as well be packed `a`, `c`, `b`, `d`, so that `a` and `c` map to one cache line and `b` and `d` to a second. This layout would use only one cache line for object `Y`.

Less extreme cases can be found in the microkernel's critical operations. For example, analysis of a particular IPC system call in the L4Ka::Pistachio microkernel reports nine fields accesses in the destination thread control block but only six of these fields are accessed in the source thread control block.

4.3.2 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.

In a write-back cache, write accesses only update the cache, marking the corresponding cache line as dirty. In contrast to a clean line, i.e., a cache line that was not the target of a write hit, the contents of a dirty line must be written back to memory before the line can be replaced. This implicit write-back operation adds to the latency of the miss causing the replacement. Write-back operations may also be triggered by other bus agents such as another processor or an I/O device accessing the memory locations cached in a dirty line.

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 as it reduces pressure on the memory hierarchy.

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. For example, analysis of a particular IPC system call in the L4Ka::Pistachio microkernel reports nine fields accessed in the thread control block but only three of these fields are written.

4.3.3 Field Alignment

Proper alignment of fields can be a matter of performance or, worse, a matter of correctness. For example, optimization guidelines for IA-32 processors [23] recommend to align all 32-, 64-, and 128-bit data such that their base addresses are a multiple of four, eight, and 16, respectively. Accesses that span cache line boundaries are likely to incur large penalties (cache line split). On the ARM processor [3], the LDR instruction will rotate the 32-bit value read from an unaligned memory location according the lower two bits of the address. The STR instruction, however, simply ignores the two lower bits of the address. Accessing a 32-bit word on an unaligned address will thus read a garbled value and store to a different location. In some system configurations, unaligned accesses may also trigger alignment exceptions. In general, primitive data types should be naturally aligned, that is, such that their base address is a multiple of their size (or the next higher power of two.)

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. When the architectural word width is smaller than the field, operations are often executed as multiple operations on parts of the field. Some instruction set extensions, such as AMD's 3D!Now and Intel's SSEx, contain instructions to efficiently manipulate larger words or multiple words at once, and optimizing compilers make use of these instructions for manipulating wide words.

Analyzing the kernel's critical path can not guarantee that fields are never accessed in larger chunks anywhere else where improper alignment may result in incorrect behavior or exceptions being raised. Slight performance penalties due to unaligned accesses may be neglected if they happen off the critical path. Yet, the requirement for natural alignment can still be relaxed when the generated code is *known* to be safe, i.e. by telling the compiler to not use complex instructions.

Avoiding multimedia instructions in the kernel may also be beneficial for other reasons, since these instructions usually utilize resources of the floating point unit (FPU). The FPU is often multiplexed (lazily) among user threads. Using it in the kernel would involve expensive state management that outweighs the potential performance gains by orders of magnitude.

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.

4.3.4 Locks and Data

Often a lock variable is stored with the data it protects. In Linux (version 2.6.16) many examples can be found, such as in block device request queues, in the directory entry cache, in socket structures, and in network device driver structures. Java implementations store object locks in the object header or with the fields of the object. The multiprocessor version of the L4Ka::Pistachio kernel stores a lock for synchronized access to thread control blocks in the thread control block itself. A spin lock's contention rate could be used to determine whether a lock is preferably placed in the same cache-line as the data it protects or in a different one to avoid false sharing.

A high-contention lock variable is frequently read by the processors trying to acquire the lock. The cache line containing the lock variable is usually marked shared in all caches. When the processor holding the lock frequently writes data to this cache line, the line is first invalidated in all other caches to exist exclusively in this processor's cache. Then the processor updates the line with the new value, but immediately after that the line will be written back to memory because the other processors try to read the lock variable and experience a cache miss. Therefore, placing frequently written data in the same cache line as the high-contention lock protecting it will result in cache-line bouncing.

In contrast, a low-contention lock variable usually resides in a cache line marked exclusive on the processor that holds the lock or held it last, or it is not in any cache. A processor acquiring the lock will fetch the cache line and mark it exclusive. Placing data that is protected by the lock in the same cache line will automatically transfer that data to the processor that is about to modify it.

Thus, a high-contention lock variable should not be stored in the same cache line as the data it protects, while it may be beneficial to store a low-contention lock variable along with the data it protects. Which fields of a data structure a lock in the same data structure protects can be specified using annotations. Alternatively, such information may be inferred automatically from code by static analysis [18].

4.4 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.

This section describes how field access patterns of performance-critical microkernel operations can be determined, and how they can be determined precisely and efficiently. First, several techniques for analyzing program behavior and their suitability for obtaining field access information are discussed in Section 4.4.1. Then, Section 4.4.2 describes characteristics of microkernels in the light of memory access tracing. In Section 4.4.3 a profiling approach to determining field access patterns for selected classes in a microkernel is discussed.

4.4.1 Method Review

Several techniques can be used to analyze program behavior. The following list briefly describes each technique and discusses its suitability for the problem of determining field access information for the critical path in a microkernel.

Static source code analysis The data and control flow information produced by analyzing the source code can be used to identify all fields of a particular class that are possibly accessed during the execution of a particular function.

Static analysis cannot identify the critical path in the kernel. The reported set of accessed fields may be much larger than the set of actually accessed fields, because the latter can be very dependent on the input, i.e., on the usage scenario of the kernel.

Static source code analysis can also not determine the exact sequence of field accesses because, to a large degree, this sequence depends on the compiler. Optimization may reorder independent field accesses compared to their order in the source code. The effective width of field accesses is also compiler-dependent. For example, when the code reads several bits in a group of bit fields, the compiler could generate many small byte-sized loads for every bit or a single wide load and extract the bits from the register one by one.

Source code analysis is usually limited to one programming language, or at least to programming languages at the same level. It fails to analyze inline assembly and assembly code that implements system call and exception handler stubs or even complete system calls.

As such, static source code analysis can only provide incomplete and conservative estimates of field access information in a kernel.

Sampling Event-based or random sampling uses hardware to generate deterministic or random events while the code to be analyzed is executing. It lends itself to analyzing a target system under a realistic workload. Random sampling can identify frequently executed code paths by recording the instruction pointer at the time of the event.

Event-based sampling can identify hot spots for resources provided the resource can trigger events.

Sampling imposes a low runtime overhead and is unintrusive as far as the target code is concerned. However, it requires substantial support infrastructure in hardware and software: hardware to trigger events at arbitrary times, the trigger event handler, and a facility to extract sampling data from the target system.

Since events are asynchronous to the instruction stream under analysis, context information must be extracted by the event handler. To determine the target of memory accesses made by the analyzed code, the handler must identify and decode the current instruction at the time of the event and calculate the addresses of memory operands based on register contents at the time of the event. Event-based sampling is not necessarily precise, especially in processors with many pipeline stages.

Logging data is collected inside the running kernel on the target system. Thus, the data must be extracted from the target system to be available to the optimization process on the build system. Data can be extracted either by the system running on top of the instrumented kernel or by a remote system via a kernel debugging facility. Both approaches require adding substantial amounts of infrastructure to the target system.

While a rich set of performance counters can be found in many server and desktop processors, their availability is often rather limited in embedded processors. Furthermore, the kernel may export them to applications or use them for its own purposes, so they may not be available for sampling. The latter is especially true for timer devices.

Statistical sampling is best used for analyzing hot spots of resource usage or hot paths in programs. However, due to its statistical nature it is unsuitable for recording exact traces of program execution in general and memory references in particular.

Instrumentation Instrumentation adds code to synchronously invoke logging functions for interesting events during execution. These events can be function entry and exit, basic block boundaries, but also memory references. Such code can be added by the compiler during the build process or later by binary instrumentation tools such as Atom [54], EEL [31], and Etch [48]. Due to its synchronous nature, instrumentation can provide exact information to the logging facility.

Instrumentation can impose significant run-time overhead, depending on the level of detail required. The instrumented code can also be much larger than the uninstrumented version. The logging infrastructure must be reachable from the instrumented code. Usually the functions are called directly and must thus reside in (and pollute) the same space as the target code.

Like with sampling, logging data is accumulated in the target system and needs to be extracted from there before it can be used for field reordering. Logging code generated by instrumentation tools often assumes to be run as an application on an operating system, e.g., it opens files or registers handler functions to be executed at program exit.

Simulation Full system simulators such as SimOS [49], Simics [62], QEMU [4], or Bochs [32] can execute the kernel with its intended workload on top. An extensible simulator allows to interpose on almost arbitrary points in the simulation, for example at the memory interface of the processor core. Assuming a faithful simulation, all memory accesses can be captured in exactly the order and width like a real target processor would issue them.

The simulator can run an unmodified kernel and workload. However, the slowdown due to simulation can be enormous. A simulator may also not be able to sufficiently represent the target system, for example exotic devices.

Using a simulator for the analysis requires that the target system is available as a simulation target. Simics is a good example as it offers a wide variety of simulation targets. The simulator API is consistent across targets, so a customized profiling extension can be reused.

Field access information can be extracted from the simulator in a format suitable for optimization. Invocation of the simulator can be made a part of the build and optimization process.

Profiling the actual kernel with workload on top has a major advantage over analyzing the source code: the actual programming language, compiler, optimization level, etc. used to generate the kernel binary are irrelevant. They are part of the kernel build process, but they are of no concern for the process of gathering field access information. What is analyzed is the combination of all, i.e., what will be running on the target system.

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. In such cases replacing the actual workload with a workload simulator causing a representative mix of kernel activities could help. System call and event statistics collected with low overhead in an instrumented kernel executing the workload on real hardware would fuel such a workload simulator.

In summary, of the methods reviewed above, executing the kernel in a full system simulator is the most suitable method for collecting field access information because it provides precise information about the memory references on the critical path.

4.4.2 Microkernel Specifics

The key to reducing the amount of memory trace data that needs to be collected is to aggressively customize the tracing process for the purpose of collecting field access information in a microkernel. That can be achieved by incorporating knowledge about the microkernel. Part of this knowledge is inherent in the way microkernels are used, part is available in the kernel source and/or configuration information. This section presents observations about microkernels that can be used to tailor the tracing process to field reordering in a microkernel.

Processor Mode

Kernel objects store state information pertaining to API objects or kernel-internal resource management. Kernel objects are accessed by kernel code.[1] That is, 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.

For an instrumented kernel this restriction is automatic, because only kernel code is instrumented to log accesses with the tracing facility. When simulation is used, the memory tracing simulator extension can ignore any accesses in unprivileged mode. Determining the simulated processor's privilege mode is a standard feature of a simulator API.[2]

Path Length

A microkernel is the lowest software layer in a microkernel-based system. As such, it acts as a service provider to applications. The kernel offers its services through system calls. Performance-critical system call handlers are rather short, typically in the order of tens or hundreds of instructions. Microkernel invocations can be thought of as separate, short runs of the program "microkernel", interspersed with longer executions of user code. With a limited code path length like this, a complete trace of one kernel invocation is limited in size, too. Furthermore, such short traces expose a high level of similarity. 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.

The short path length through the kernel, the resulting small trace size for one kernel invocation, and the high similarity of traces of kernel invocations suggest online processing of the trace at the end of the path rather than after many invocations at the end of a tracing session. Such short traces are similar in spirit to the hot data streams [13] with the difference being that they are readily identified by the start and end of kernel code execution whereas hot data streams are extracted from a program's complete data reference trace.

The start and end of a kernel invocation are detected the easiest by the memory tracing approach using a simulator. An instrumented kernel would require annotations of all possible kernel entry and exit points to find the start and the end of a trace.

[1] The kernel may export kernel objects so that unprivileged code can access them. However, optimizing user-level accesses to kernel objects is outside the scope of this work.

[2] Simics exposes the SIM_processor_privilege_level() function [63], bochs defines the function get_CPL, and QEMU uses a masking operation on a bitfield of flags.

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 thread control blocks, the performance-critical kernel objects, in an array. At the time of writing, only one L4 kernel [40] allocates thread control blocks dynamically from the kernel heap. However, it then stores their addresses in a statically allocated table.

When address ranges of objects of target classes either are known in advance or can be determined easily at run time, as is often the case, the trace data can be filtered immediately to contain only references to objects of target classes.

Addresses of dynamically allocated objects in a running kernel can be found by monitoring updates to statically allocated indirection tables (like those described above.) Address ranges of statically allocated objects can be statically configured in the tracing facility.

Number of Objects

Often-called and thus performance-critical microkernel calls typically reference only very few kernel objects. More complex operations involving many kernel objects, such as address space deletion, tend to be invoked less frequently. For example, a simple IPC message transfer between two threads in the L4Ka::Pistachio microkernel involves two, at most three thread control blocks. The pingpong benchmark running on the kernel references at most four thread control blocks during one kernel invocation, in the startup phase when address spaces are created.

4.4.3 Precise Tracing for Field Reordering

Full memory reference traces of programs are precise in the sense that they do not omit information. Usually they are huge (in the multi-Gigabyte range for a few seconds of program execution) and require post-processing to extract the interesting information. They often contain a high percentage of useless information. In contrast, field affinity graphs [12] and member transition graphs [28] 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 [44], and has suggested to keep complete traces when the sequence of memory references is short.

The remainder of this section describes the design of a tracing facility 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. It heavily exploits the microkernel specifics described in Section 4.4.2 to customize tracing. The facility collects only the information that is necessary to drive field reordering with the (additional) strategies described in Section 4.3.

For static customization, the tracing facility uses information from various sources such as definitions in the kernel source, addresses from the kernel binary's symbol table, and debug information from the kernel binary. These are 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.

Address Filtering and Type Inference

Memory references are filtered by processor privilege mode and address ranges as detailed 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.

Since address information of statically and dynamically allocated, typed kernel objects is used to filter memory references, the type of the kernel object that was referenced can be inferred from the address range the reference was made to. Along with the information about the memory access, the address filter delivers the base address and the type of the referenced object.

To find references to objects in (potentially padded) arrays of objects, address filtering employs a number of range-stride-size checks. The address is first compared with the lower and upper bounds of the array, then the offset into the array is taken modulo the distance between array elements, the stride, and compared with the object size. Address ranges occupied by single objects, that is, static objects and dynamically allocated objects, are stored in data structures supporting fast lookup. Addresses of statically allocated objects (that are known at build time) are inserted at the start of tracing.

Supporting large padding between objects in an array is necessary as this space is often used for different purposes: Most L4 kernels keep the kernel stack of a thread in the unused part of the 1KB or 2KB block that is allocated for the TCB in the linear virtual array of TCBs. Likewise, the Linux kernel stores a thread's kernel stack in unnamed space behind the thread control structure, for which 8KB are allocated in total.

Address Abstraction

Accesses to fields of different objects need to be tracked separately to allow optimizing for differing field usage patterns. However, the actual addresses of referenced objects are not relevant for field reordering.

To distinguish between the kernel objects that are used during a microkernel invocation while abstracting from their addresses, the tracing facility assigns sequential object numbers to objects as they 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 as described abstracts from the actual object in favor of an "object role."

The tracing facility can store addresses of objects sequentially in an array, using the array index as the object number. Linear search is efficient because typically the expected number of referenced objects during one performance-critical microkernel invocation is very small (2-3). Kernel invocations that reference more objects than fit into the statically sized array are labeled as problematic and ignored or reported to the kernel builder.

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.)

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.

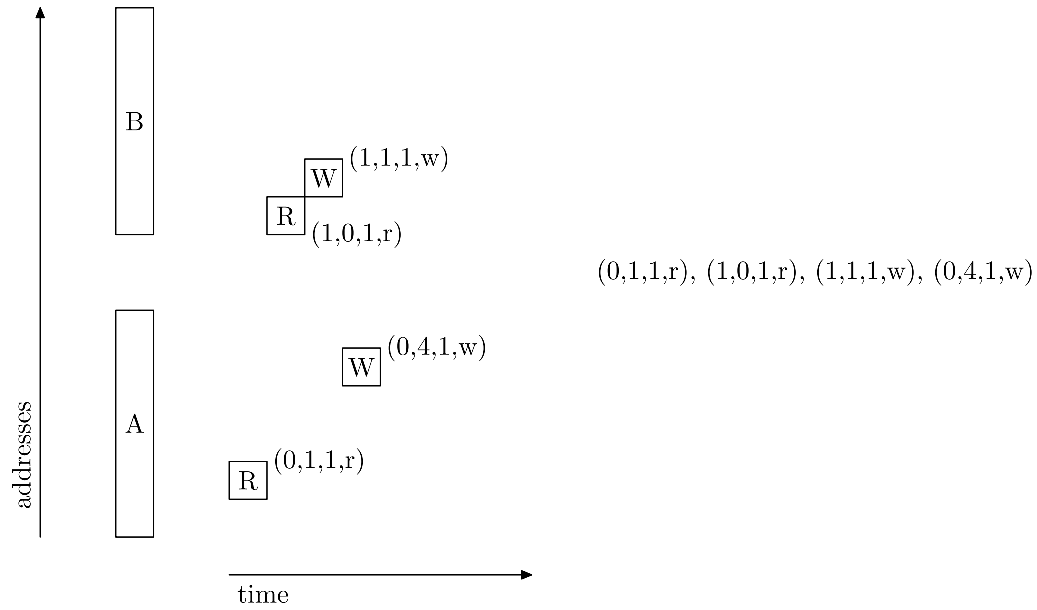


Figure 4.3: Two objects (A and B) of a class, a time line of references to fields of these objects (R=read, W=write) between kernel entry and kernel exit, and the resulting quadruple sequence.

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. For example, sequences recorded in the L4Ka::Pistachio microkernel during a run of the pingpong IPC benchmark are between 2 and 85 quadruples long.

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 indicates a possibility for merging both sequences, thereby further increasing the weight of the more frequent sequence.

4.5 Field Reordering Algorithms

Section 4.3 discussed a number of strategies for placing fields in objects. These strategies can be incorporated into algorithms for finding an optimal member order based on the precise field access information collected by the tracing approach described in the previous section. Which strategies should be applied, and with what priorities in case of conflicts, depends to a large extend on the hardware configuration (e.g., caching strategies, allocation policies) as well as the direct and indirect costs of accesses (e.g., access latency, delayed background operations.) Devising a generic algorithm that covers all scenarios is beyond the scope of this work. Therefore, this section merely presents an example of a field reordering algorithm.

The following algorithm targets a scenario with a single object of the target class being accessed on the critical path. The algorithm further assumes a write-back cache, i.e., modified cache lines are more expensive than unmodified lines due to the delayed write-back operation. All fields of the target class that are modified constitute the *rw* set; all fields that are only read constitute the *ro* set.

1. Move the *rw* set to the start of the class definition.

2. Move the *ro* set behind the *rw* set.

3. Order the members of each set by decreasing field size.

4. Fill any alignment-related gap between the two sets by moving suitably-sized members of the *ro* set to the start of the *ro* set.

The result is a cluster of all referenced members at the start of the (cache-line aligned) object. At the start of this cluster, which maps to the minimum number of cache lines, the modified members form another cluster, which maps to a minimum number of *dirty* cache lines, as shown in Figure 4.4.

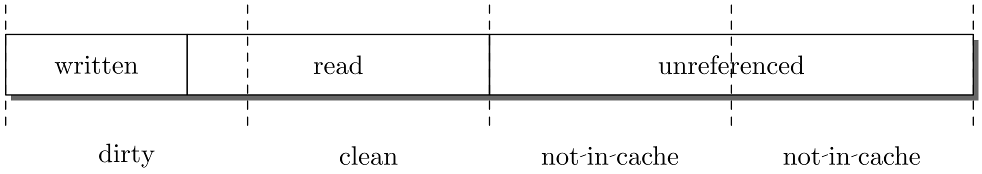


Figure 4.4: Field groups after reordering

4.6 Optimization Process

The optimization proposed in this chapter comprises transparent class flattening followed by field reordering, whereby field reordering is driven by profiling data collected from a previous execution of the kernel to optimize. Figure 4.5 illustrates the various stages of the optimization process.

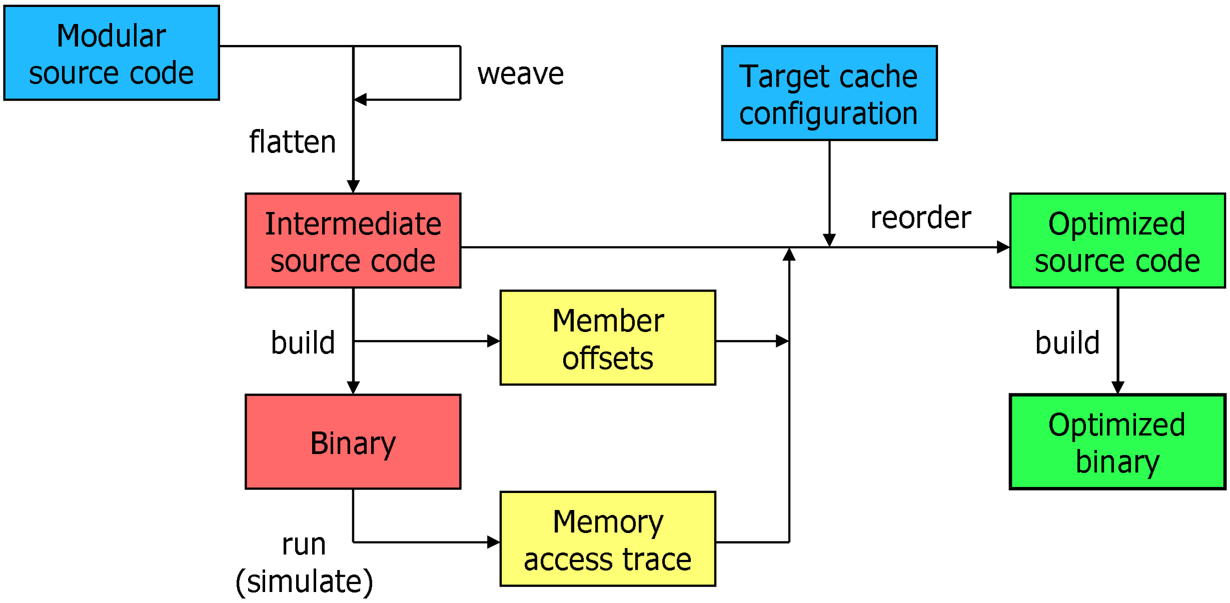


Figure 4.5: The optimization process

First, the modular source code that uses inheritance to compose kernel objects from a number of small, configuration-specific classes is preprocessed and target classes are flattened.

The resulting intermediate source code is then built to produce a kernel binary. During the build process, information such as the offsets of fields in target classes is extracted. The kernel with the envisioned workload is executed in a simulator to obtain field access information for those classes that should have their fields reordered.

Field access information, field offsets, and information about the target system's cache such as the cache line size drive the field reordering process that rewrites the declarations of target classes in the intermediate source code.

Finally, the rewritten, optimized source code is built to produce a kernel binary that uses optimized field placements. The intermediate source code is not stored between both kernel builds but recreated instead. Reordering is only active during the second, optimizing build when the necessary data is available.

The combination of class flattening and field reordering also lends itself to optimizing aspect-oriented code. Weaved code could thus also serve as input to the flattening process. The weaving process typically includes a preprocessing phase itself so that the preprocessor then need not be run prior to flattening.

Transparent Optimization

A kernel developer should not need to run flattening or reordering tools separately. Instead, they should be an integral part of the build process and thus become a mostly transparent optimization. Applying both optimizations during every build run adds only minimal overhead, allowing them to be "always-on".

Class flattening and field reordering are best applied after preprocessing the source code, as shown in Figure 4.6. Depending on the actual build system, the compiler driver is launched such that it either invokes the three stages in place of the preprocessor or assumes an already preprocessed source with the three stages being run prior to compiler invocation.

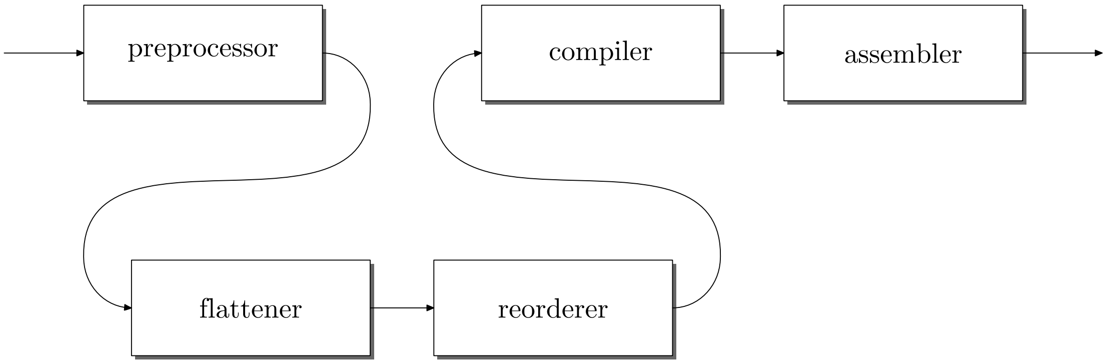


Figure 4.6: The standard build process for a source file is extended by class flattening and field reordering between the preprocessing and the compilation stage.

In the Pistachio microkernel, the selection of classes that are good candidates for optimizing is portable across all configurations. Therefore, the list of classes to be optimized can be set statically in the build environment, so that the common kernel developer does not need to be concerned with it. Input for field reordering can be picked up automatically by the reordering stage once the kernel has been profiled.

5 Evaluation and Discussion

The optimization technique described in this thesis is aimed at enabling object-oriented programming in microkernels without negatively impacting their performance. A kernel with a class hierarchy that has been optimized should therefore be compared with the original kernel, which does not use inheritance.

Evaluation of the optimization technique was performed in the context of the Pistachio microkernel. A microbenchmark reveals the impact of the optimization on the kernel's cache footprint and IPC performance. When an optimization is to be performed the costs of the optimization must be considered as well. Such costs include the application costs of the optimization, but also those to create and maintain the optimization tools.

The hardware and the software tools that were used to perform the evaluation are described in Section 5.1. Section 5.2 evaluates the effectiveness of the proposed optimization technique in terms of function call overhead, cache footprint and execution performance of the kernel's IPC path; it also discusses side effects of the technique. The costs of the automated optimization technique are discussed in Section 5.3, followed by a comparison with manual optimization in Section 5.4.

5.1 Evaluation Environment

Evaluation of the optimization technique involved measuring microkernel performance on hardware as well as analyzing the tools that perform the various steps of the optimization process. In this section, I first describe the hardware used for carrying out the experiments and then briefly introduce the tools.

Except for Section 5.2.1, the system used for the performance analysis contains a 3.6 GHz Intel Pentium 4 processor with a 16 KB write-back L1 data cache and a trace cache with a capacity of 12 Kμops. The shared L2 cache of 2 MB is sectored with a line size of 64 Bytes and two lines per sector. Furthermore, the system contains 2 GB of PC-4300 266 MHz DDR2 main memory. For this hardware, Table 5.1 lists the cost of memory accesses and kernel entry and exit operations in processor cycles. Memory latencies were measured with the CPU-Z tool [61]. Kernel entry and exit use the SYSENTER and SYSEXIT instructions, respectively.

Profiling runs to determine access patterns for member reordering and the cache analysis were executed in the Simics full-system simulator [62], version 3.2.17. Simics offers simulation targets for many architectures and popular platforms. It has a well-documented API [63] which can be accessed from modules written in C or Python.

Operation	Cycles
L1 hit	4
L1 miss, L2 hit	30
L2 miss	318
kernel entry+exit	183

Table 5.1: Costs of various operations in terms of processor cycles.

Performance of the microkernel is evaluated by measuring IPC performance with the `pingpong` IPC microbenchmark, in which two threads send messages back and forth and measure the round trip time. Those threads can reside in the same address space or in different address spaces. Throughout the benchmark, the length of the messages is increased. For each message size, the benchmark reports number of processor cycles for a single message transfer (i.e., half the number of cycles for a round trip) averaged over a large number of transfers.

The Class Flattening Tool

Class flattening is implemented as a source-to-source transformation in a stand-alone class flattening tool called `collapse`, which is described in more detail by Oberländer [41]. `collapse` modifies the declaration of classes whose names are specified via command line options. It performs the following steps:

1. The input file is parsed into an abstract syntax tree (AST). The parser is based on the GCC grammar. Using the C AST library `libcast`, it creates an internal representation of the source code that can be easily manipulated: Duplicating or moving code are operations on subtrees, and renaming identifiers (e.g., class names) involves only replacing a string pointer.

2. Various visitors walk the AST to perform the following operations: They determine the class hierarchy, clone base class members into the target class, remove the keyword virtual, adjust scope names of stand-alone members, remove the inheritance specifier and optionally any base classes.

3. Another visitor then writes the textual representation of the updated AST as C++ source code into the output file.

The class flattening process operates on preprocessed source code and is therefore to be executed between the preprocessing and the compilation stage.

The Profiling Extension

The profiling simulator extension collects data member access characteristics for target classes during execution of the kernel under workload. It is written in C and interfaces with the Simics API. It refers to header files from the Pistachio kernel source tree to statically configure address ranges, for example, for TCBs.

The extension interposes at the interface between the processor core and the memory hierarchy where it can snoop all memory accesses generated during program execution. It does so by registering a function that is invoked for every memory access with information about the direction (read vs. write), the cause (data vs. instruction), the logical and the physical address, and the size of the access.

Accesses during one kernel invocation are filtered by processor mode and address range as described in Section 4.4.3. Actual addresses are abstracted into an object number in the current invocation and an offset. Every access to an object of a target class is stored as a quadruple (object number, offset, size, read or write) in a per-class sequence. The sequences are recorded and eventually processed at the end of the invocation, i.e., when switching back to user mode. The profiling extension keeps track of distinct access sequences and counts their occurrences.

```
Paths recorded: 17722
Paths dropped:  0
    1  0,128,4,r 0,128,4,r 0,128,4,r 1,0,4,r 1,0,4,w 1,128,4,r 1,0,4,r 1,0,4,w 1,12,4,w 1,128,4,w 1,16,4,w 1,180,4,w 1,20,4,w 1,176,4,w
       1,0,4,w 1,4,4,w 1,24,4,w 1,184,4,w 1,188,4,w 1,192,4,w 1,36,4,w 1,72,4,w 1,88,4,w 1,92,4,w 1,80,4,w 1,84,4,w 1,96,4,w 1,100,4,w
       1,112,4,w 1,116,4,w 1,122,2,w 1,120,2,w 1,28,4,w 1,124,4,w 1,4,4,w 1,128,4,w 1,32,4,w
    1  0,128,4,r 0,128,4,r 1,0,4,r 1,0,4,w 1,128,4,r 1,0,4,r 1,0,4,w 1,12,4,w 1,128,4,w 1,16,4,w 1,180,4,w 1,20,4,w 1,176,4,w 1,0,4,w 1,4,4,w
       1,24,4,w 1,184,4,w 1,188,4,w 1,192,4,w 1,36,4,w 1,72,4,w 1,88,4,w 1,92,4,w 1,80,4,w 1,84,4,w 1,96,4,w 1,100,4,w 1,112,4,w 1,116,4,w
       1,122,2,w 1,120,2,w 1,28,4,w 1,124,4,w 1,4,4,w 1,128,4,w 1,32,4,w
    2  0,128,4,r 0,128,4,r 1,0,4,r 1,128,4,r
    2  0,128,4,r 0,128,4,r 1,0,4,r 1,0,4,w 1,128,4,r 1,0,4,r 1,128,4,r 1,128,4,r 1,12,4,r 1,4,4,r 1,128,4,r 1,128,4,r 1,4,4,r 1,12,4,w
       1,0,4,r 1,8,2,w 1,128,4,r 1,32,4,w 1,12,4,r 1,28,4,w 1,28,4,r 1,28,4,w 1,28,4,r 1,28,4,w 1,28,4,r 1,28,4,w 1,28,4,r 1,28,4,w 1,28,4,r
       1,28,4,w 1,28,4,r 1,28,4,w 1,28,4,r 1,28,4,w 1,28,4,r 1,28,4,w 1,16,4,w 1,20,4,w 1,12,4,r 1,176,4,w 1,180,4,w 1,124,4,w 1,0,4,r
    1  0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 0,36,4,r 1,72,4,r 1,72,4,w 0,68,4,w 0,64,4,w 0,36,4,w 0,20,4,w 0,16,4,w 0,24,4,r 0,32,4,r 0,28,4,w
       0,16,1,r 0,36,1,r 0,48,4,r 0,112,4,r 0,52,4,w 0,48,4,w 0,36,4,r 0,36,4,w 1,16,1,r 1,4,4,r 1,32,4,r 1,28,4,r 1,24,1,r 1,132,4,r
       1,132,4,w 1,24,4,r 1,180,4,r 1,12,4,r 1,140,4,r 1,144,4,r 1,148,4,r 1,152,4,r 1,176,4,r 1,20,4,w 1,16,4,w 1,176,4,w 1,180,4,w
    1  0,12,4,r 0,72,4,r 1,16,4,r 1,20,4,r 0,0,4,r 1,16,4,w 0,16,4,w 0,24,4,r 1,4,4,r 1,32,4,r 0,32,4,r 0,28,4,w 1,28,4,r 1,24,4,r 1,36,1,r
       1,36,4,r 1,64,4,r 0,72,4,w 1,68,4,w 1,64,4,w 1,36,4,w 0,20,4,w 0,12,4,r 0,176,4,r 0,16,4,w 1,16,4,w 1,12,4,r 0,112,4,r 1,112,4,r
       0,36,4,r 1,12,4,r 1,4,4,r
    1  0,12,4,r 1,0,4,r 1,16,4,r 0,16,4,w 0,20,4,w 0,24,4,r 0,32,4,r 0,28,4,w 2,16,1,r 2,4,4,r 2,32,4,r 2,28,4,r 2,24,4,r 2,16,4,w 2,20,4,r
       2,12,4,r 2,4,4,r
    1  0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 1,20,4,r 1,20,4,w 0,12,4,r 1,12,4,r 1,12,4,r 1,176,4,r 1,16,4,w 0,16,4,w 0,12,4,r 1,112,4,r
       0,112,4,r 1,36,4,r 1,112,4,r 1,48,4,w 0,52,4,r 1,52,4,w 0,52,4,r 0,48,4,w 0,52,4,w 1,36,4,w 1,112,4,r 1,112,4,r 0,12,4,r 0,4,4,r
    1  0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 1,20,4,r 1,20,4,w 0,12,4,r 1,12,4,r 1,12,4,r 1,176,4,r 1,16,4,w 0,16,4,w 0,12,4,r 1,112,4,r
       0,112,4,r 1,36,4,r 1,112,4,r 1,48,4,w 0,52,4,r 1,52,4,w 0,52,4,r 2,48,4,w 0,52,4,w 1,36,4,w 1,112,4,r 1,112,4,r 0,12,4,r 0,4,4,r
    1  0,12,4,r 0,72,4,r 0,16,4,w 0,20,4,w 0,24,4,r 0,32,4,r 0,28,4,w 0,16,1,r 0,36,1,r 0,48,4,r 0,112,4,r 0,48,4,r 0,52,4,r 1,52,4,w
       0,52,4,r 2,48,4,w 0,36,4,r 0,36,4,w 1,16,1,r 1,4,4,r 1,32,4,r 1,28,4,r 1,128,4,r 1,12,4,r 1,180,4,w 1,16,4,w
   32  0,128,4,r 0,128,4,r 0,16,4,r 0,20,4,w 0,12,4,r 0,140,4,w 0,144,4,w 0,148,4,w 0,152,4,w 0,156,4,w 0,20,4,r 0,176,4,w 0,16,4,r 0,180,4,w
       0,12,4,r 0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 1,20,4,r 1,20,4,w 0,12,4,r 1,12,4,r 1,12,4,r 0,20,4,w 0,16,4,w 0,24,4,r 1,4,4,r 1,32,4,r
       0,32,4,r 0,28,4,w 1,28,4,r 1,128,4,r 1,24,4,r 1,36,1,r 1,16,4,w 1,20,4,r 1,12,4,r 1,4,4,r
   32  0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 1,20,4,r 1,20,4,w 0,16,4,w 1,16,4,w 1,0,4,r 0,20,4,w 0,160,4,w 1,12,4,r 0,12,4,r 1,12,4,r 0,12,4,r
       1,12,4,r 0,12,4,r 1,128,4,r 0,128,4,r 0,24,4,r 0,36,1,r 0,132,4,r 0,132,4,w 1,12,4,r 0,72,4,r 0,16,4,w 0,20,4,w 0,24,4,r 1,4,4,r
       1,32,4,r 0,32,4,r 0,28,4,w 1,28,4,r 1,128,4,r 1,24,4,r 1,16,4,w 1,20,4,r 1,12,4,r 1,4,4,r 1,12,4,r 1,140,4,r 1,144,4,r 1,148,4,r
       1,152,4,r 1,176,4,r 1,20,4,w 1,180,4,r 1,16,4,w 1,156,4,r 1,176,4,w 1,180,4,w
    2  0,128,4,r
    1  0,12,4,r 1,0,4,r 1,16,4,r 0,16,4,w 0,20,4,w 0,24,4,r 0,32,4,r 0,28,4,w 0,16,1,r 0,36,1,r 0,48,4,r 0,112,4,r 0,48,4,r 0,52,4,r 1,52,4,w
       0,52,4,r 1,48,4,w 0,36,4,r 0,36,4,w 1,16,1,r 1,4,4,r 1,32,4,r 1,28,4,r 1,128,4,r 1,12,4,r 1,180,4,w 1,16,4,w
    1  0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 1,20,4,r 1,20,4,w 1,12,4,r 0,20,4,w 0,16,4,w 0,24,4,r 1,4,4,r 1,32,4,r 0,32,4,r 0,28,4,w 1,28,4,r
       1,128,4,r 1,24,4,r 1,36,1,r 1,16,4,w 1,20,4,r 1,12,4,r 1,4,4,r
    1  0,128,4,r 1,12,4,r 0,128,4,r 1,0,4,r 1,128,4,r 0,12,4,r
 2000  0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 1,20,4,r 1,20,4,w 1,12,4,r 0,20,4,w 0,16,4,w 0,24,4,r 1,4,4,r 1,32,4,r 0,32,4,r 0,28,4,w 1,28,4,r
       1,128,4,r 1,24,4,r 1,16,4,w 1,20,4,r 1,12,4,r 1,4,4,r
15641  0,12,4,r 1,0,4,r 0,0,4,r 1,16,4,r 1,20,4,r 1,20,4,w 0,12,4,r 1,12,4,r 1,12,4,r 0,20,4,w 0,16,4,w 0,24,4,r 1,4,4,r 1,32,4,r 0,32,4,r
       0,28,4,w 1,28,4,r 1,128,4,r 1,24,4,r 1,16,4,w 1,20,4,r 1,12,4,r 1,4,4,r
```

Figure 5.1: Profiling output for class tcb_t after partial execution of the pingpong benchmark. Lines have been wrapped for completeness of presentation.

At the end of the simulation, the profiling extension dumps its output into per-class files. Figure 5.1 shows an example of such a file. These files reside on the machine running the simulation which is usually also the machine used for building the kernel. The results of profiling are thus readily available for the layout optimizer.

The Layout Optimizer

The layout optimizer, called `shuffle`, determines a new member ordering for a class. It uses profiling data and the class' object layout information as input. `shuffle` is a standalone tool and operates on one class at a time.

In the profiling output, the sequence with the highest count represents the critical path and drives optimization. When there is no single predominant sequence, the optimizer tries to merge sequences that have nonconflicting optimization goals. That is the case when a sequence references a subset of the members referenced by another sequence. Alternatively, the sequence to govern optimization can also be chosen manually.

Member offsets and sizes are extracted from debug information generated during the build process of the profiled kernel. The object-relative addresses in the elements of the sequence are translated to member names.

The layout optimizer employs one or more strategies to determine a new member ordering. The cache configuration and the actual strategies and their respective parameters are specified via the optimizer's command line. The result is a list of member names in the order in which they should appear in the optimized class' declaration.

The Member Reordering Tool

Like class flattening, member reordering is implemented as a source-to-source transformation. The reordering tool `reorder` manipulates the order in which members appear in a class declaration. Since `reorder` and `collapse` are very similar, they share large parts of the code base and the processing. However, `reorder` applies different visitors to manipulate the AST. In the AST, all members in a class declaration form a linked list of subtrees. Therefore, reordering members merely involves sorting this list accordingly. Padding members are inserted by creating a subtree for an array of bytes of the respective size.

The new ordering of members for a class (as determined by the layout optimizer) is given to `reorder` via a command line option. For each target class, the list of the first members is specified. For example, `A:a3,a1,a2` requests that class A be modified such that the list of members starts with a3, followed by a1, and then by a2. The order of the remaining members is left untouched. Naturally, by moving specified members to the start of the declaration, all other members move automatically to the back. Padding space can be inserted by specifying its size a number in the member list. For example, `A:a3,a1,8,a2` would insert a new member, an array of 8 bytes, between members a1 and a2. Knowing the names of all members in the class, `reorder` can find a name for the padding array(s) without producing conflicts.

The reordering tool expects preprocessed source code. As such, it is invoked between the preprocessing and the compilation stage of a standard build process. It reads an input file and generates an output file, but it can also be used as a pipe to form arbitrary tool chains.

Build Process Integration

The optimization described in this thesis is a whole program transformation. In a project consisting of multiple source files, the optimization must be applied equally to all source files.

The make-based infrastructure of the Pistachio source tree specifies a rule for transforming a C++ source file into an object file. It invokes the compiler driver gcc, which generates an object file from a source file through preprocessing, compilation, and assembly. This rule has been replaced with the command chain shown in Figure 5.2.

```
gcc -E file.cc | collapse -c tcb_t | reorder -r tcb_t:... | gcc -o file.o
```

Figure 5.2: C++ file build command line with optimization tools (most options omitted)

For each file, the compiler driver gcc is invoked to merely preprocess the source file. Its output is sent to collapse, the class flattening tool. collapse flattens the tcb_t class hierarchy as specified by a command line option. The output of collapse is then fed to the reorder data member reordering tool. If a reordering specification file for class tcb_t is available, reorder is invoked with the specification for reordering the tcb_t class members; otherwise, reorder performs an identity transformation. The output of reorder is eventually passed to gcc to execute the compilation and assembly stages, whereby gcc's input is declared as pre-processed C++ source code.

5.2 Automatic Class Optimization Results

To evaluate the automatic optimization approach described in this thesis for its effectiveness in eliminating the overheads of inheritance, I compare a kernel with an optimized tcb_t class hierarchy with an original kernel, which does not use inheritance for the class.

I modified the build process of the Pistachio kernel to include both class flattening and field reordering and transformed the formerly simple tcb_t class into a class hierarchy. The reordering tool shuffle was configured to move all members to the beginning of the class declaration in the order they were referenced. For this purpose, shuffle used data member offsets extracted from the kernel binary and traces collected with the profiling simulator extension executing while the kernel with the pingpong benchmark.

In this section, I first show that class flattening removes the runtime overhead of virtual functions. I then compare the cache footprints of the optimized and the original kernel on objects of the tcb_t class, followed by a comparison of IPC performance of both kernels. Furthermore, I discuss side effects of the optimization.

5.2.1 Virtual Functions

To show the effectiveness of class flattening in removing the run-time overhead of virtual function calls, I continued the experiment described in Section 3.4.1. There, I converted the flat class tcb_t into a class hierarchy with several virtual functions that previously were inline functions in class tcb_t.

I then applied transparent class flattening to the class, turning the hierarchy into a flat class again. Figure 5.3 shows the performance of the IPC message transfer mechanism.

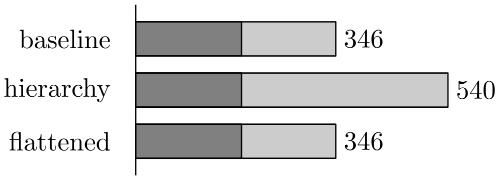


Figure 5.3: Overhead of virtual functions removed by transparent class flattening

The IPC performance of both the baseline and the flattened kernel is identical. In fact, the binaries for both kernels differ merely in the string that embeds the build time and compiler version into the binary. Through flattening the virtual functions became nonvirtual and were as good candidates for inlining as they were in the baseline kernel.

5.2.2 Cache Footprint

An IPC message transfer between two threads involves two `tcb_t` objects, the sender thread's TCB and the receiver thread's TCB labeled *src* and *dst*, respectively in the following figures. A `tcb_t` object occupies 192 bytes in memory, is 1K-aligned and therefore spans three cache blocks of 64 bytes.

Figure 5.4 and Figure 5.5 both show the mapping of referenced members to cache blocks in *src* and *dst* during an IPC message transfer between threads in different address spaces. Thin vertical lines mark 64-byte cache block boundaries. Figure 5.4 shows the mapping for the original kernel whereas Figure 5.5 shows the mapping for the optimized kernel.

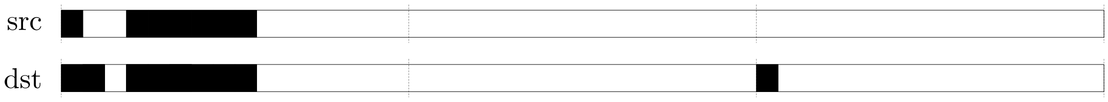


Figure 5.4: Referenced members in `tcb_t` objects of the original kernel

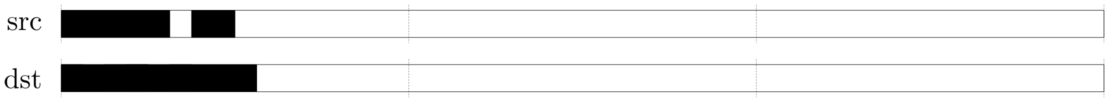


Figure 5.5: Referenced members in `tcb_t` objects of the optimized kernel with fields reordered for minimum cache footprint

The original kernel uses three cache lines for the two `tcb_t` objects accessed during an IPC operation: one for the sender thread's TCB and two for the destination thread's TCB, whereby a single member is referenced in the third cache line of the destination thread's TCB.

In contrast, the object layout of the optimized `tcb_t` class starts with a compact hot part so that the accessed members span only one cache block per object. Field reordering has achieved the minimum cache footprint of `tcb_t` objects on the kernel's critical path.

5.2.3 Performance

Class flattening removes inheritance and thereby the overhead of virtual function calls. Field reordering optimizes the object layout of the flattened class for the configuration. Performance of a kernel with inheritance is therefore expected to be no worse than that of the original kernel. An optimized object layout may even improve performance in configurations that the original, static layout was not optimal for.

To simulate cache pressure from applications, I extended the classic pingpong IPC benchmark to execute a cache thrashing loop before every message transfer. The loops in both the ping and the pong thread each reference 32KB of memory (twice the size of the L1 data cache) in 64-byte increments, thereby effectively replacing the whole L1 cache contents.

I analyzed the following three configurations: the standard pingpong IPC benchmark on a standard Pistachio kernel (*vanilla*); the cache thrashing pingpong benchmark on a standard kernel (*baseline*); and the cache-thrashing pingpong benchmark on a kernel with an optimized `tcb_t` class hierarchy (*optimized*.) For each configuration, Table 5.2 shows the average transfer time for a 0-word message between two threads in the same address space (*intra*) and in different address spaces (*inter*). Figure 5.6 visualizes those numbers with the overhead of the cache thrashing loop factored out.

	inter	intra
vanilla	1329	401
baseline	4713	3659
optimized	4677	3643

Table 5.2: IPC performance in terms of processor cycles. *baseline* and *optimized* include the overhead of the cache thrashing loop in the benchmark application.

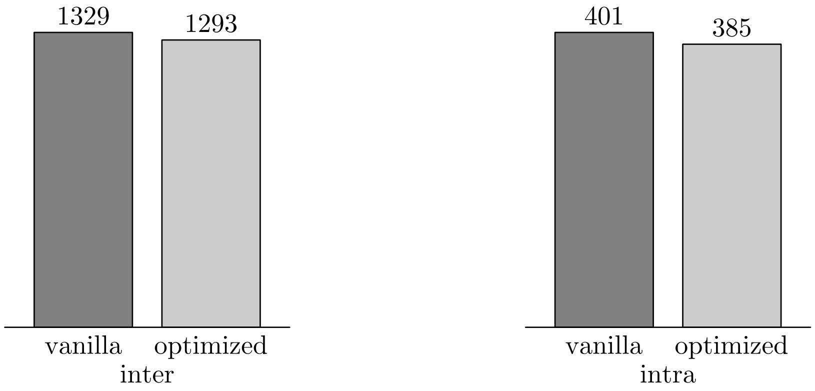


Figure 5.6: IPC performance in terms of processor cycles. The costs of the cache thrashing loop have been factored out of the *optimized* numbers.

Despite the introduction of a class hierarchy and virtual functions in the `tcb_t` class, the measurements show the performance of the kernel with the optimized `tcb_t` class hierarchy to be no worse than that of the original kernel with a flat `tcb_t` class.

Most importantly, this result confirms my thesis, that object-oriented programming can be used even in such performance-critical cases as microkernel code without any extra costs. The run-time overheads of inheritance shown in Section 3.4 have been eliminated completely by the optimization described in this work.

Furthermore, the result shows that the optimization was able to improve on the IPC performance of a kernel with a hand-optimized, supposedly optimal object layout. For IPC between threads in different address spaces, the optimization saved 36 cycles or 2.8 percent. For communication in the same address space the savings were 16 cycles or 4.2 percent. The improvements can in part be attributed to the reduced cache footprint observed in Section 5.2.2, in part to side effects discussed in the next section. That the improvements are not a multiple of the costs given in Table 5.1 can be explained with out-of-order effects.

5.2.4 Side Effects

Side effects of the optimization beyond manipulation of object layouts can also influence performance of the kernel. A full cache analysis helped revealing these. The numbers of data and instruction cache blocks accessed during a round-trip IPC (i.e., two message transfers) are shown in Table 5.3.

(a) Raw values

	inter	intra
vanilla	25/20	15/20
baseline	1051/21	1039/21
optimized	1049/19	1039/19

(b) Cache thrashing factored out

	inter	intra
vanilla	25/20	15/20
optimized	23/18	15/18

Table 5.3: Data and instruction cache footprint of a 0-word IPC message round-trip. The numbers represent 64-byte cache blocks consumed by data/code.

Whereas the data cache footprint is the same for both kernels (vanilla and optimized) when threads in the same address space communicate, the footprint is reduced by one cache line per message transfer in case of a cross-address-space message transfer. The thread switching code for IA-32 accesses the `space` member of the destination TCB only when the threads reside in different address spaces. In the baseline kernel, this member at offset 128 maps to a different cache block than the remaining accessed members (Figure 5.4.) In the optimized kernel, all referenced members in the object occupy the same cache block.

With the optimized kernel, the code cache footprint is two blocks smaller (i.e., one block per IPC) than in the original kernel. While the instruction sequence is identical for both kernels, the offsets of members in the objects changed and hence the addressing modes of instructions accessing the members can differ. The `space` member of the `tcb_t` class is at offset 32 in the optimized kernel and can be accessed with an 1-byte immediate displacement in the instruction. In contrast, an offset of 128 in the baseline kernel requires a 4-byte displacement. The reduced instruction length for accesses to the `space` member shrunk the containing functions such as the IPC function by 3 bytes per such instruction.

For another member that happened to be moved from offset 0 to offset 4, the compiler also generated more compact code in the optimized kernel. As a result, the cache footprint of the IPC path taken in the benchmark case was reduced by one cache block, and IPC performance improves.

In an early experiment, the `space` member was manually swapped with the unreferenced `cpu` member near the start of the `tcb_t` object. The new offset allowed for a one-byte displacement whereby the hand-crafted assembly version of the IPC path shrunk by three bytes. This caused a jump target to be no longer aligned, which effectively resulted in worse performance. After aligning the jump target performance improved as expected.

Most of the observed side effects can be attributed to the variable instruction length of the x86 architecture. However, even architectures with a fixed instruction length may require additional instructions when member offsets and hence displacements change across the limits of addressing modes.

5.3 Optimization Costs

To employ the optimization described in this thesis, class flattening and field reordering are added to the build process. Given the commonly low frequency of kernel builds, the moderate increase in build time is insignificant. The simulator run at the end of the day to produce input for an optimizing kernel build happens even less often.

The real costs of the optimization described in this thesis lay in the initial implementation of the tools, the integration into the build system, and the adaptation to new target configurations.

5.3.1 Build Time Overhead

On today's development systems, building a microkernel such as the Pistachio kernel from source code takes only a fraction of a minute. This is almost negligible compared to the time a programmer spends thinking, writing code, and debugging.

The class flattening and field reordering source-to-source transformations as performed by the `collapse` and `reorder` tools each parse, manipulate, and emit the complete preprocessed source code before it is seen by the compiler, adding to the execution time of each source file's build process. Figure 5.7 shows the time to build a Pistachio kernel with and without flattening and reordering. The measurements were performed with nonparallelizing builds using GCC 4.1.2 on an openSUSE 10.2 system with two 2.2 GHz AMD Opteron 248 processors, 2 GB main memory, and a FiberChannel-attached disk storage array.

Adding the class flattening and the field reordering tools to the build process increases build time by 21 percent. Yet, with the commonly low frequency of kernel builds, this increase is negligible.

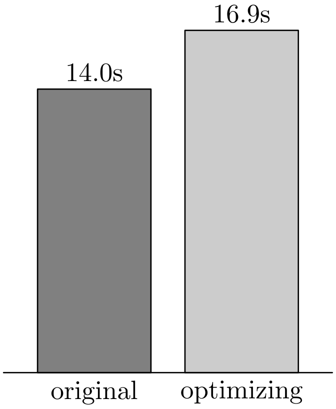


Figure 5.7: Kernel build time overhead

5.3.2 Code Size

The source-to-source transformation tools `collapse` and `reorder`, which share a common code base, have been built around the `libcast` C++ parser library originally developed for the IDL compiler IDL4. The profiling extension uses the Simics API. The offset extraction tool is based on `libdwarf` for access to debug information.

	source lines of code	language
collapse + reorder	2002	C++
simulator extension	308	C
field offset extractor	245	C
build system integration	40	Makefile

Table 5.4: Code size of optimization software components

Table 5.4 lists code sizes for the various software components involved in the optimization process. The numbers state actual lines of source code as reported by the `sloccount` utility [64].

5.3.3 Retargeting

Initially, the optimization process was implemented for IA-32 targets of the Pistachio microkernel. To determine the cost of applying the optimization to a different configuration, I adapted the process to the MIPS32 target.

Neither the source-to-source transformation tools nor their integration with the kernel build process required any changes. All modifications were confined to the simulator profiling extension. There, the detection of privilege levels needed minor adjustment: for some targets, the Simics API function `SIM_processor_privilege_level` returns the numeric value of the processor's current privilege level, for other targets it returns one of the predefined constants `SIM_CPU_Mode_User` or `SIM_CPU_Mode_Supervisor`.

Altogether, the required changes amounted to 20 lines of code. Adapting the optimization to further target configurations is expected to require even less work.

5.3.4 Maintenance

The Pistachio kernel is a research project under active development. Since the `libcast` parser was implemented on an as-needed basis, introduction of previously unused language constructs in Pistachio can require updates. However, this situation is rare as the parser is mostly complete and thus the list of unused but useful language constructs is nearly exhausted.

The Simics simulator used for collecting field access information is also evolving continuously. With previous major upgrades, part of the architecture-specific API functionality was marked deprecated in favor of more generic functionality. Such changes, however, had only little impact on the profiling extension. It is expected that the trend towards more generic internal interfaces eases future adaptation of the simulator extension to new target configurations.

Based on the impact of changes seen so far, the maintenance burden for keeping the optimization tools up to date with their dependencies is estimated at a few hours per year.

5.4 Comparison with Manual Optimization

The object layout optimization part of the technique described in this thesis automatically adapts to the kernel configuration and the workload on top of the kernel. It is thereby superior to manual layout optimizations that would have to be repeated for every such combination of kernel configuration and workload.

When classes for kernel objects are composed from multiple base classes, such as proposed in Section 3.3, manual optimization of object layouts is impossible. The class hierarchy needs to be flattened before manual layout optimizations can be applied.

Even when the cost of manual optimization would not matter, the result is not guaranteed to be optimal, as shall be illustrated by the following example. Figure 5.8 visualizes the result of a cache analysis of the IPC path for an inter-address space IPC at the time this work was started, after the kernel had been manually optimized for Pentium III and Pentium 4 processors.

In such a kernel, the `space` member lies at offset 128 and therefore maps to an additional cache line. It is referenced during a cross-address-space IPC, which is the dominant IPC operation in a microkernel-based multiserver operating system. Swapping `space` with the unreferenced member `cpu` at offset 8 would have reduced the cache footprint. This was one of the performance bugs discovered throughout the course of this work.

While optimizing for the Pentium III processor, it was observed that reading the currently active page table base address from control register CR3 was faster than reading it from the current thread's TCB. Later, while optimizing for the Pentium 4 processor, however, reading from memory turned out to be faster. The inline assembly code in the

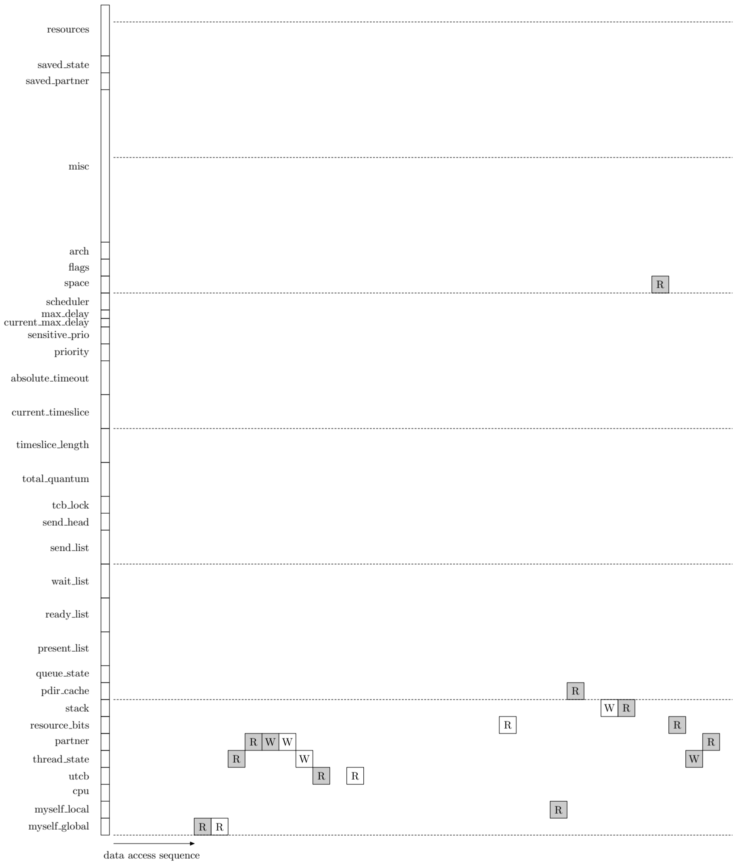


Figure 5.8: Accesses to members of the tcb_t class on the IPC system call path. Time progresses in units of memory accesses from left to right. The object layout is shown vertically, with thin horizontal lines marking cache block boundaries of 32 bytes. R=read, W=write, gray=source TCB, white=destination TCB.

thread switching function `switch_to()` and the hand-optimized assembly implementation of the IPC path were therefore implemented in two variants of which one was selected by conditional compilation via preprocessor symbols derived from the current target processor selection in the kernel configuration.

The observed effect can be explained with the results of the cache analysis. On a Pentium III with a cache line size of 32 bytes, the cache footprint of a 0-word IPC message transfer is 4 cache lines. The member `pdir_cache` at offset 32, which holds the page table base address for the thread, maps to a separate cache line. Not reading it and retrieving the value from the CR3 register reduces the cache footprint by one line. On a Pentium 4 processor with a 64 byte cache line size this member is in the same cache line as all other members (except for the `space` member at offset 128) and thus hits in the cache. Reading the current page table base address from CR3 is then more expensive than reading it from memory.

6 Conclusions

My thesis concludes by summarizing the thesis contributions in Section 6.1 and giving suggestions for future work in Section 6.2.

6.1 Contributions of This Work

This thesis makes several research contributions:

- Class flattening has been used for enhancing program understanding, for eliminating the run-time overhead of virtual function calls, and for software quality measurements. I use class flattening to convert *a hierarchy of classes* to a single flat class that *can be optimized with member reordering*, and to convert virtual functions into normal functions.

- Class flattening has previously produced flattened versions of the original class that coexisted with the original. Code modifications were required to use the flattened version. I propose *transparent class flattening*, which transparently changes the definition of the original class and thus requires no changes to the code using the class.

- Member reordering has previously been used for improving the cache performance of *application* programs. I apply member reordering to *kernel* data structures to minimize the kernel's cache footprint on the critical path.

- Previously, member reordering for C has only produced suggestions that had to be applied *manually*. I use source-to-source transformation to *automatically* rewrite data structure declarations prior to compilation.

- Member reordering for C has been limited to *struct-like* data structures. I extend its applicability to classes composed from a *hierarchy using inheritance*.

- Previous approaches to member reordering were driven by *pair-wise access frequencies* extracted from profiling information, which has been shown to be insufficient for finding an optimal layout. I propose a profiling approach that can collect *complete and thus precise yet compact* access traces by exploiting knowledge about the properties of microkernels, thus allowing for optimal ordering.

- I propose several *novel strategies* for member reordering that can increase the optimization potential.

Together, above contributions enable the careful use of inheritance and virtual functions in a microkernel, for example to improve modularity and portability, without the run-time overhead traditionally associated with them. Selected performance-critical classes can be automatically optimized for the target system configuration and workload.

The following, additional contributions were made:

- For proof of concept, the `reorder` tool was designed and implemented, which allows reordering the data members of selected C++ classes as devised by an external specification. Reordering is performed as a source-to-source transformation to be applied prior to compiler invocation.
- For proof of concept, a profiling extension for the Simics full system simulator was designed and implemented to collect member access information on a microkernel's critical path.
- For proof of concept, class flattening and profile-driven member reordering were integrated into the build process of the Pistachio microkernel.
- For proof of concept, the class representing a thread in Pistachio was modified for one configuration to be composed of several configuration-specific base classes.

The optimization technique described in this thesis is not limited to objects in a microkernel. It can be applied whenever inheritance is used for composition of classes and dynamic polymorphism is not needed. An example are libraries that use a class hierarchy to share code between various classes, but implement only most derived classes and operations on them.

6.2 Suggestions for Future Work

For this work, a proof-of-concept conversion of Pistachio's TCB class `tcb_t` into a class hierarchy was performed for one target configuration. Future work should complete this conversion for all configuration dimensions and also devise a mechanism to create the inheritance relationship automatically from the target configuration. Furthermore, other classes than the TCB could benefit from this optimization. The `space_t` class seems to be a potential candidate.

Set of classes that benefit from the optimization is rather static today for a microkernel like Pistachio, making the selection portable across all target configurations. Approaches to automatically identifying candidate classes could help dynamically adapting to future design changes of the kernel.

Minimizing the kernel's cache footprint has been a focus of this work. Field reordering could also be beneficial in the absence of caches such as in deeply embedded systems, when the memory latency of subsequent accesses might depend on the sequence of addresses.

The extension of the optimization technique beyond the microkernel, towards applications and libraries might lead to improved overall system performance.

Finally, with automatic and safe inference of candidate classes, the technique could be integrated into compilers as a standard optimization.

7 Bibliography

[1] E. H. L. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization.* J. Wiley & Sons, New York, 1997.

[2] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. *Lecture Notes in Computer Science*, 1098:142–166, 1996.

[3] ARM Ltd. *ARM Architecture Reference Manual*, June 2000.

[4] Fabrice Bellard. QEMU – a full system emulator. Project homepage at `http://fabrice.bellard.free.fr/qemu/`, 199x–2007.

[5] 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 1992.

[6] Dirk Beyer, Claus Lewerentz, and Frank Simon. Flattening inheritance structures — or — Getting the right picture of large OO-systems. Technical Report I-12/2000, Institute of Computer Science, Brandenburg University of Technology, Cottbus, November 2000.

[7] 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.

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

[9] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2005.

[10] 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, USA, 1992. IEEE Computer Society.

[11] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, OR, 1994.

[12] Trishul M. Chilimbi. *Cache-Conscious Data Structures — Design and Implementation.* PhD thesis, University of Wisconsin, Madison, 1999.

[13] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 191–202, New York, NY, USA, 2001. ACM Press.

[14] 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.

[15] 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.

[16] Uwe Dannowski. Automated object layout optimization in a portable microkernel. In *Proceedings of the MIKES 2007: First International Workshop on MicroKernels for Embedded Systems*, pages 22–28, Sydney, Australia, January 2007.

[17] ECMA International. *Standard ECMA-335 Common Language Infrastructure (CLI)*, third edition, June 2005.

[18] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[19] Free Software Foundation, Inc. *GCC online documentation.* 51 Franklin St, Fifth Floor, Boston, MA 02110, USA, September 2006.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, fourth edition, 2006.

[21] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[22] IBM Corp. *IBM XL C/C++ Enterprise Edition V8.0 for AIX Online Documentation*, October 2005.

[23] Intel Corp. *IA-32 Intel Architecture Optimization Reference Manual*, April 2006.

[24] Intel Corp. *Intel C++ Compiler Documentation*, September 2006.

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

[26] International Organization for Standardization (ISO). *ISO/IEC 9899:1999(E) Programming Languages — C*, January 2005.

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

[28] 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.

[29] Bradley M. Kuhn and David Binkley. An enabling optimization for C++ virtual functions. In *Selected Areas in Cryptography*, pages 420–428, 1996.

[30] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[31] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. *SIGPLAN Not.*, 30(6):291–300, 1995.

[32] Kevin Lawton. bochs: The open source IA-32 emulation project. Project homepage at `http://bochs.sourceforge.net/`, 1998–2007.

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

[34] Claus Lewerentz and Frank Simon. A product metrics tool integrated into a software development environment. In *ECOOP'98: Workshop on Object-Oriented Technology*, pages 256–260, London, UK, 1998. Springer-Verlag.

[35] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 237–250, New York, NY, USA, 1995. ACM Press.

[36] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture.* Prentice Hall, 2006.

[37] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[38] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

[39] Microsoft Corp. *Microsoft Visual C++ Online Documentation*, September 2006.

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

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

[42] Dresden University of Technology Operating Systems Group. The Fiasco microkernel. Project homepage at `http://os.inf.tu-dresden.de/fiasco/`, 1998.

[43] University of New South Wales Operating Systems Research Group. L4/MIPS. Project homepage at `http://l4mips.sourceforge.net/`, 2000.

[44] 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.

[45] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment.* J. Wiley & Sons, New York, 1996.

[46] Eric S. Raymond. *CML2 Language and Tools Description*, June 2000.

[47] RedHat, Inc. and others. Cygwin: A Linux-like environment for Windows. Available from `http://www.cygwin.com/`, 2000–2007.

[48] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian N. Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop 1997*, pages 1–8, August 1997.

[49] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.

[50] Guido Van Rossum and Fred L., Jr. Drake (Editor). *The Python Language Reference Manual.* Network Theory, 2003.

[51] S. Schönberg. L4 on Alpha, design and implementation. Technical Report CS-TR-407, University of Cambridge, 1996.

[52] Amit Singh. *Mac OS X Internals: A Systems Approach.* Addison-Wesley Professional, 2006.

[53] David A. Solomon. *Inside Windows NT.* Microsoft Press, second edition, 1998.

[54] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[55] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

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

[57] System Architecture Group. *L4 eXperimental Kernel Reference Manual, Version X.2.* Karlsruhe University (TH), August 2006.

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

[59] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[60] 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.

[61] Unknown author. CPU-Z: a freeware system information utility. Available from `http://www.cpuid.com/cpuz.php`, May 2007.

[62] Virtutech Inc. Simics — a full system simulator, 1998–2007.

[63] Virtutech Inc. *Simics Reference Manual (PAL)*, December 2005. Simics Version 3.0.

[64] David E. Wheeler. Estimating Linux's size, November 2000.

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

[66] 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.