JOHANNES KEPLER
UNIVERSITY LINZ | JKU

Christian Wimmer

Automatic Object Inlining in a Java Virtual Machine

A thesis submitted in partial satisfaction of
the requirements for the degree of

Doctor of Technical Sciences

Institute for System Software
Johannes Kepler University Linz

Accepted on the recommendation of

Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck
Johannes Kepler University Linz

Univ.-Prof. Dipl.-Ing. Dr. Michael Franz
University of California, Irvine

Linz, February 2008

Abstract

Object-oriented Java applications allocate many small objects linked together by field references. Such fields are frequently loaded and thus impose a run-time overhead. *Object inlining* reduces these costs by eliminating unnecessary field loads. It embeds referenced objects into their referencing object. The order of objects on the heap is changed in such a way that objects accessed together are placed next to each other. Then their offset is fixed, i.e. the objects are *colocated*. This allows field loads to be replaced by address arithmetic.

Array inlining expands the concepts of object inlining to arrays, which are frequently used for the implementation of dynamic data structures. Therefore, the length of arrays often varies, and fields referencing such arrays have to be changed. An efficient code pattern detects these changes and allows the optimized access of such array fields.

These optimizations are integrated into Sun Microsystems' Java HotSpot™ virtual machine. The analysis is performed automatically at run time, requires no actions on the part of the programmer, and supports dynamic class loading. It is embedded into the just-in-time compiler and garbage collector and does not need a global data flow analysis. The evaluation shows that the dynamic approach successfully identifies and optimizes frequently accessed fields for several benchmark suites with a reasonable low compilation and analysis overhead.

Kurzfassung

Objektorientierte Java-Anwendungen verwenden viele kleine Objekte, die über Felder miteinander verbunden sind. Solche Felder werden häufig geladen und erhöhen die Ausführungszeit. Objekt-Inlining reduziert die Kosten, indem unnötige Feldzugriffe entfernt werden. Die Reihenfolge von einander referenzierenden Objekten wird so geändert, dass verbundene Objekte im Speicher hintereinander liegen. Dadurch ist der Abstand zwischen den Objekten konstant und Feldzugriffe können durch Adressberechnungen ersetzt werden.

Array-Inlining erweitert die Optimierung von Objekten auf Arrays. Diese werden oft für dynamische Datenstrukturen verwendet. Dadurch ist ihre Länge nicht bekannt, und Felder, die auf Arrays zeigen, werden häufig geändert. Da diese Änderungen effizient erkannt werden können, ist die Optimierung solcher Felder möglich.

Diese Optimierungen sind in die Java HotSpot™ VM von Sun Microsystems integriert. Die Analyse wird automatisch zur Laufzeit ausgeführt, benötigt keine speziellen Zusatzinformationen und unterstützt das dynamische Laden von Klassen. Sie ist in den Just-in-Time-Compiler und den Garbage Collector integriert und verwendet keine globale Datenfluss-Analyse. Die Evaluierung zeigt, dass dieser dynamische Ansatz mit geringem Laufzeit-Aufwand wichtige Felder erkennt und optimiert.

Table of Contents

Acknowledgements

I want to thank all of the many people that supported this work. First and foremost, I thank my advisor Hanspeter Mössenböck for his constant support and encouragement and his diligent comments on algorithms, papers, and this thesis. Likewise, I thank Michael Franz from the University of California in Irvine to take the responsibility of the second thesis advisor and dissertation committee member.

The Java HotSpot compiler team at Sun Microsystems continuously supported and funded the project. I thank Kenneth Russell, Thomas Rodriguez, and David Cox for the numerous discussions and helpful comments on all parts of the Java HotSpot virtual machine.

Special thanks go to my former colleague Thomas Kotzmann, who also worked on the Java HotSpot VM. Our discussions in the early phases constituted the project and influenced the overall architecture of the implementation. Additionally, his comments on the thesis improved the quality. Likewise, I thank all my current colleagues, especially Thomas Würthinger and Markus Löberbauer for the numerous discussions. Both provided valuable feedback on the thesis.

Finally, I am most grateful to my parents, my brother, and my sister for their love as well as their encouragement and support during my whole studies.

Chapter 1

Introduction

This chapter starts with an introduction to the object-oriented programming language Java. Java applications are executed using a virtual machine. This imposes a run-time overhead but also allows novel feedback-directed optimizations that cannot be implemented in a static compiler. Object inlining is such an optimization, which utilizes the just-in-time compiler and the garbage collector of the VM. The chapter closes with a history of the research collaboration that facilitated this project.

Object-oriented programming encourages developers to decompose applications into a large number of small classes with a well-understandable and well-testable functionality. *Encapsulation* hides the implementation details of a class from a client, *inheritance* allows subclasses to specialize the behavior of a superclass, *polymorphism* allows objects of subclasses to be treated like objects of the superclass, and *dynamic binding* ensures that a method call is dispatched to the overridden method of the receiver's dynamic type. These concepts improve modularity, extensibility, and reusability of software components.

However, a proper decomposition of an application into classes can have a negative impact on performance. It leads to a large number of small objects on the heap that are linked together by object fields. This increases the number of loads for referencing fields, i.e. the number of memory accesses of the application. This thesis presents an optimization that places certain objects next to each other on the heap and replaces field loads by address arithmetic.

1.1 Java

The Java programming language [Gosling05] was developed by Sun Microsystems as a general-purpose object-oriented language. The syntax is similar to C++, however the complex and unsafe features of C++ were omitted. Instead, concepts such as garbage collection and thread synchronization were introduced to simplify software development. Java was designed as a portable language, so applications compiled once run on various host architectures. Additionally, Java allows a secure delivery of software components.

The emerging World Wide Web contributed much to the success of Java. While the interactivity of plain HTML pages is limited, the integration of small Java programs, so-called *applets*, into web pages allows the development of interactive applications that are integrated into the web browser. Transferring executable code over an untrusted network such as the Internet requires checks before execution, in order to guarantee that no malicious code is executed on the client.

Today, Java is used on a wide variety of platforms. Small embedded systems such as mobile phones and PDAs can be programmed easily without special knowledge about the target architecture using the *Java Platform Micro Edition* (Java ME). Midway in the spectrum, the *Java Platform Standard Edition* (Java SE) provides an environment for desktop applications, supporting the developer with a library for graphical user interfaces, network programming, XML processing, and multimedia applications.

The development of component-based multi-tier enterprise applications is facilitated by the *Java Platform Enterprise Edition* (Java EE), providing a large framework that significantly simplifies the development of secure and transaction-oriented server applications. Using one programming language for all types and sizes of systems is an advantage over specialized languages and can reduce the time and costs of software development.

To enforce the portability and platform independence, Java applications are not distributed as machine code for a specific hardware platform. Instead, the concept of a *virtual machine* (VM) is used for abstraction. Java source code is compiled to a compact binary representation called *Java bytecodes*. The application is stored in a well-defined binary format, the *class file format*, containing the bytecodes together with a symbol table and other supplemental information. The Java virtual machine [Lindholm99] is defined independently from the Java programming language. The *Java HotSpot VM* is Sun Microsystems' implementation of this specification [SunHotSpot]. High performance is achieved by using just-in-time compilation [Cramer97].

1.2 Feedback-Directed Optimizations

Compared to statically compiled binaries, the execution of applications in a virtual machine has several advantages, such as portability, safety, automatic memory management, and dynamic loading of code. However, it requires new optimization algorithms to achieve high performance. Instead of optimizing the code at compile time using a long-running and highly optimizing static compiler, optimizations are performed at run time in parallel with the execution of the application. This requires fast optimization algorithms that focus on the relevant parts of an application.

Optimizing at run time has several benefits: It allows one to collect live profiling data from the currently running workload to guide optimizations specifically to the *hot spots*, i.e. the most frequently executed parts of an application. The hot spots can be optimized in a better way than in a static compiler, which leads to a higher overall performance. Optimizations based on profiling data are called *feedback-directed optimizations* [Arnold05]. While the actual algorithms vary, most of them use the following pattern:

- *Profiling*: It is necessary to collect accurate profiling data with a low overhead. The two most commonly used techniques are sampling and program instrumentation. Sampling collects a new data set periodically using a timer (see e.g. [Anderson97]). Program instrumentation modifies the application so that relevant events trigger, for example, a counter increment (see e.g. [Yasue03]). Usually, sampling has less overhead, but program instrumentation is more accurate. Combined approaches (see e.g. [Arnold01]) can achieve both goals.
- *Optimization*: It is necessary to check whether the optimization is allowed in the context of the currently loaded code. If all preconditions are satisfied, the optimization is applied and the optimized code is executed.
- *Run-time checks*: With dynamic loading of code, it is possible that an optimization that has already been applied becomes unsafe, e.g. when a new class is loaded. In this case, the optimization must be undone.

Examples for such optimizations are method inlining based on call profiles [Hölzle96], optimizations of the machine code layout [Pettis90], multiversioning of code based on type profiles [Chambers91], and optimizations to improve the memory locality and cache behavior [Chilimbi98]. The algorithms described in this thesis follow the feedback-directed optimization pattern to perform object inlining for important fields selected by a counter-based profiler.

1.3 Problem Statement

Figure 1.1 shows a snapshot of the object graph that could be part of an object-oriented drawing application. A `Polyline` object uses the Java collection class `ArrayList` to maintain a dynamic list of points. Internally, the `ArrayList` stores its data in an `Object[]` array. When new points are added and the size of the array does not suffice, the array is replaced by a larger copy. The array elements reference the points that store the actual coordinates. The line color of the `Polyline` is maintained by a separate `Color` object. This follows the encapsulation principle: the methods that manipulate a color are separated from the `Polyline` class because they are generally usable in other parts of the application or in other applications.

The object-oriented structure requires three objects and one array to model the figure. Several memory accesses are necessary to load a single data element. For example, the field `lineColor` must be loaded before the actual data field `rgb` is accessible. When the objects are scattered on the heap, the cache performance is also affected negatively.

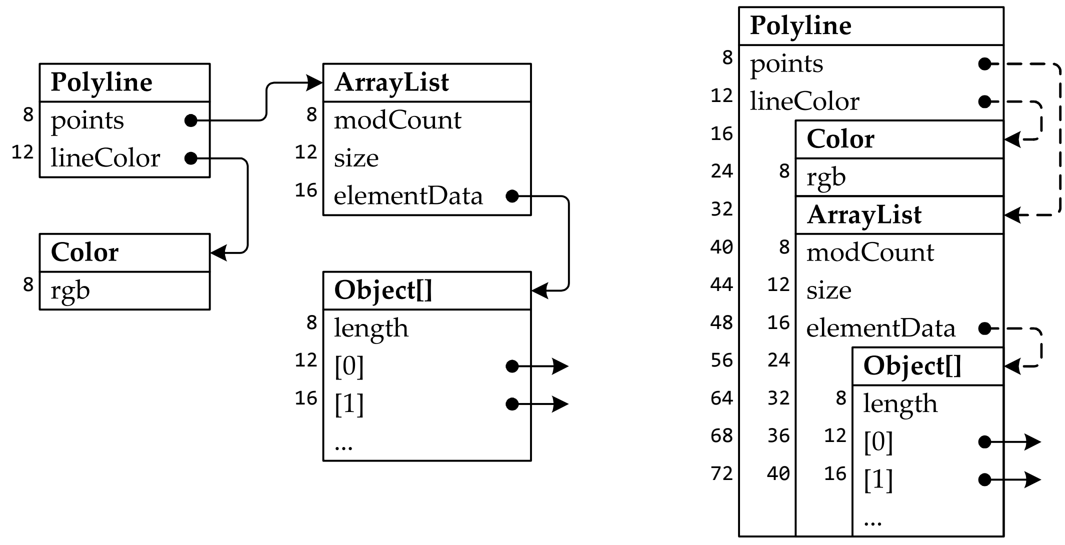


Figure 1.1: Motivating example for object inlining

Object inlining combines the objects to a larger group so that the field `rgb` can be loaded or stored with a single memory access relative to the beginning of the `Polyline` object. All objects are aligned at 8-byte boundaries. The numbers on the left hand side show the field offsets in bytes. For example, the field `rgb` can be accessed with the offset 24 relative to the `Polyline` object. Without object inlining, it would be necessary to access the field `lineColor` with the offset 12 of the `Polyline` object and then the field `rgb` with the offset 8 of the `Color` object.

Using the standard collection class `ArrayList` for the management of the point list has several advantages over using a simple array. The collection provides a generally usable, well-known, and thoroughly tested implementation for a dynamic list of elements. It is also easy to change the array-based data structure to a linked list by just replacing the `ArrayList` with a `LinkedList`.

The additional field loads for accessing the `ArrayList` object and the `Object[]` array are removed by object inlining. In the optimized object structure, the address of a point can be loaded with a single array access, using a larger offset relative to the `Polyline` object. Arrays like the `Object[]` array of the example play an important role in many object-oriented applications. While objects are used to statically decompose a program into small parts, arrays are ideal for the implementation of dynamic data structures. It is likely that the object fields `points` and `lineColor` are assigned only once in the constructor of the `Polyline`, while the array field `elementData` is changed whenever the list is resized. Object inlining must take this dynamic behavior into account.

Implementing object inlining as a feedback-directed optimization at run time has several advantages over a static compile-time optimization. First, the optimization is fully transparent for both the programmer and the user of the application. It is neither necessary to specify at the source code level which fields should be optimized nor to invoke a special bytecode optimizer after compilation. All existing Java applications are automatically optimized when they are executed with the improved Java VM. Secondly, modular applications that are compiled, deployed, and installed in several parts are globally optimized. At run time, all classes are treated equally, regardless of their actual source. Thirdly, code that exists in libraries but is never actually executed does not affect the optimization because unused classes are not loaded into the VM. If such classes contain code that invalidates object inlining, a static approach must be conservative, while the run-time approach can perform the optimization optimistically. Finally, inlining of array fields that can be changed at run time is not possible at all in statically compiled languages such as C++, because it requires support from the garbage collector.

The drawback of the run-time approach is that the optimization algorithms must be fast. Otherwise, the time spent for applying the optimization outweighs its benefits. Additionally, the analysis is based on the set of currently loaded classes and must always expect that newly loaded classes will invalidate the optimization. A global data flow analysis would be complex and slow, so our implementation uses mainly local information collected during just-in-time compilation.

Several programming languages such as C++ or C# allow the developer to influence the mode how objects are allocated. *Reference objects* are allocated on the heap, while *value objects* are handled like scalar values and are either placed on the stack or inlined into a reference object. Java does not offer value objects at the language level in favor of a simple object model.

In C++ [ISO14882], the programmer can decide freely between reference objects and value objects, i.e. the same class can be used to define both kinds of objects. This is error-prone because of the different semantics, e.g. when variables are assigned. Regarding our example, it is possible to inline the `Color` object and a collection object into the `Polyline` manually using value objects. However, it is not possible to inline the data array of the collection class because its maximum size is not known when the `Polyline` is allocated. Only a run-time optimization is capable of inlining such dynamic data structures.

C# [ISO23270] avoids the confusing semantics of C++. The programmer must explicitly distinguish between *classes* for reference objects and *structures* for value objects by using either the keyword `class` or `struct`. This prohibits a flexible re-use of classes. The programmer must decide early in the development cycle whether `Color` should be a structure or a class, i.e. whether it should be inlined into another object or not. The collections of the class library are only available as classes, so it would not be possible in our example to inline the point list manually. Because C#, like Java, is compiled to bytecodes executed by a virtual machine [ISO23271], automatic object inlining similar to our approach is possible.

1.4 State of the Art

Our feedback-directed object inlining combines two different areas of previous research: object inlining for static compilers and feedback-directed object colocation. To the best of our knowledge, we present the first implementation and evaluation of the combined approach. This section gives a brief overview of some existing algorithms, while Chapter 9 on page 117 contains a detailed comparison.

Dolby et al. integrated object inlining into a static compiler for a dialect of C++ [Dolby97, Dolby98, Dolby00]. When objects are inlined, all methods that might access these objects are cloned so that different methods for optimized and unoptimized objects are available. They use an expensive data flow analysis to detect and optimize all possible inlining candidates. The resulting high compilation time is acceptable for a static compiler. As a precondition for inlining a group of objects, their algorithm requires that this group can be allocated together and that it is not separated afterwards.

Laud implemented object inlining for Java in the CoSy compiler construction framework, a static compiler for Java [Laud01]. This algorithm can detect when a field is changed, i.e. when an object of a group is replaced after allocation. It is, however, not allowed that an object inlined into an object `A` is also referenced by some other object `B`. Lhoták et al. compared the approaches of Dolby and Laud and evaluated the number of fields that are inlinable for several Java benchmarks [Lhoták05].

Huang et al. developed a system for feedback-directed object colocation called *online object reordering*, implemented for the Jikes RVM [Huang04]. They use the adaptive compilation system of Jikes, which periodically records the currently executed methods. Hot fields accessed in these methods are traversed first in their copying garbage collector. This changes the object order on the heap. The decision which fields are hot is based on a static analysis of the methods. By using the existing interrupts of Jikes, their analysis has a low run-time overhead.

Chilimbi et al. use generational garbage collection for cache-conscious data placement in the object-oriented programming language Cecil [Chilimbi98]. They use a counter-based profiling technique to construct an object affinity graph that guides a copying garbage collector. However, they do not distinguish between different fields of the same object.

1.5 Project History

Our project is part of a long-standing and successful collaboration between Sun Microsystems and the Institute for System Software (named Institute for Practical Computer Science before 2004) at the Johannes Kepler University Linz. The project targets several parts of the Java HotSpot VM. It started in 2000 when Hanspeter Mössenböck extended the original client compiler [Griesemer00] with an intermediate representation in static single assignment (SSA) form [Mössenböck00]. Michael Pfeiffer started the implementation of a linear scan register allocator [Mössenböck02].

This work was continued in the context of the author's master's thesis [Wimmer04]. Several optimizations of the algorithm improved the quality of the generated machine code [Wimmer05] and therefore the peak performance of the Java VM. Nevertheless, they were fast enough to have no negative impact on the startup speed. Finally, the algorithm was integrated into the product version of the Java HotSpot VM and is part of the product since Java 6 [SunJava6]. The source code is available as part of the OpenJDK project [SunOpenJDK].

Thomas Kotzmann experimented with porting the Java HotSpot client compiler from C++ to Java [Kotzmann02]. For his PhD thesis, he extended the client compiler with a fast algorithm for escape analysis [Kotzmann05a]. It detects allocation sites of objects that do not escape a single method or a single thread. The first allows the objects to be replaced by scalar variables, while the latter allows the objects to be allocated on the method stack and synchronization on them to be removed [Kotzmann05b]. Support for garbage collection and the deoptimization framework of the Java HotSpot VM required also changes of the run-time system [Kotzmann07].

To simplify debugging of the client compiler and to show the compilation phases, the *Java HotSpot Client Compiler Visualizer* displays the intermediate representation of the compiler both in textual and in graphical form [C1Visualizer]. The project was started by the author as a tool to visualize the data structures of the linear scan register allocator. Then it was extended in the context of several student projects. Bernhard Stiftner ported the application to the NetBeans Platform [Stiftner06], Thomas Würthinger added a graphical view for control flow graphs [Würthinger06], Stefan Loidl a graphical view of the data flow [Loidl07], and Alexander Reder a textual view of the bytecodes [Reder07] and machine code [Reder08]. The tool is now available as an open source project and generally useful for the client compiler community.

Thomas Würthinger joined the project to implement array bounds check elimination for the client compiler [Würthinger07b, Würthinger08b]. The algorithm detects *fully redundant checks* that are proven to never fail, and also moves *partially redundant checks* out of loops. For his master's thesis, he implemented a visualization tool for the intermediate representation of the Java HotSpot server compiler [Würthinger07a, Würthinger08a].

Several papers document the development of the thesis at hand. The project started with an implementation of automatic object colocation that improves the cache behavior [Wimmer06]. It uses the garbage collector to place related objects next to each other on the heap. Object inlining guarantees that objects connected by a certain field are colocated and then replaces loads of this field by address arithmetic [Wimmer07]. Special support for arrays allows the inlining of array fields even when arrays are dynamically resized [Wimmer08].

1.6 Structure of the Thesis

This thesis is organized as follows: Chapter 2 presents the structure of the Java HotSpot virtual machine without object inlining. It describes the main subsystems that are relevant for this thesis: the object layout, garbage collector, just-in-time compiler, and the deoptimization framework. Chapter 3 introduces the overall system structure of our feedback-directed object inlining system. It presents the design principles and architectural decisions and closes with a walk-through of the optimization process.

The subsequent chapters contain the detailed algorithms of the optimization. Chapter 4 describes the detection of hot fields using read barriers, Chapter 5 the object colocation algorithms that are embedded into generational garbage collection, and Chapter 6 the object inlining algorithms that are embedded into the just-in-time compiler. Chapter 7 generalizes object inlining to array inlining and illustrates the differences between objects and arrays.

Chapter 8 evaluates our implementation using several benchmark suites and discusses the results. Chapter 9 presents related projects for object inlining as well as run-time optimizations for the improvement of the cache behavior. Additionally, some dynamic profiling techniques are described. Finally, Chapter 10 concludes the thesis with a recapitulation of the essential parts and a summary of our contributions.

Chapter 2

The Java HotSpot VM

This chapter introduces the Java HotSpot VM, which forms the basis for the object inlining algorithm of this thesis. The garbage collector of the VM uses a stop-and-copy algorithm for the young generation and a mark-and-compact algorithm for collections of the entire heap. The just-in-time compiler uses two different intermediate representations for global optimizations and register allocation. Additionally, the basic object layout with two header words as well as the deoptimization framework are presented. These subsystems of the VM are prerequisites for the following chapters.

The *Java HotSpot Virtual Machine* [SunHotSpot] is developed by Sun Microsystems and provides the foundation of their Java Development Kit. The VM is available on a wide variety of platforms and operating systems. Sun supports the Sparc architecture, the Intel IA-32 architecture, and the 64-bit extensions of AMD and Intel, running with different operating systems such as Sun Solaris, Microsoft Windows, and Linux. Versions for other platforms and operating systems, such as Apple's Mac OS X, are also available through Java technology licensees. This guarantees the platform-independent execution of Java applications on all major architectures.

The entire virtual machine is written in C++. This is necessary for the low-level functionality of the runtime environment. However, many of the complex features of C++ are used with caution, e.g. multiple inheritance and templates. Most of the source code is independent from the target architecture and the operating system. Only small parts must be changed to port the VM to a new platform.

2.1 System Structure

Figure 2.1 shows the VM configuration for the execution of interactive desktop applications, which is the basis for our implementation. The bytecodes of all methods are loaded by the class loader and initially executed by the interpreter. Because interpreting bytecodes is comparatively slow, frequently executed methods are compiled by the *client compiler* [Kotzmann08] to optimized machine code when their invocation counters reach a certain threshold. This is called mixed-mode bytecode execution [Agesen00].

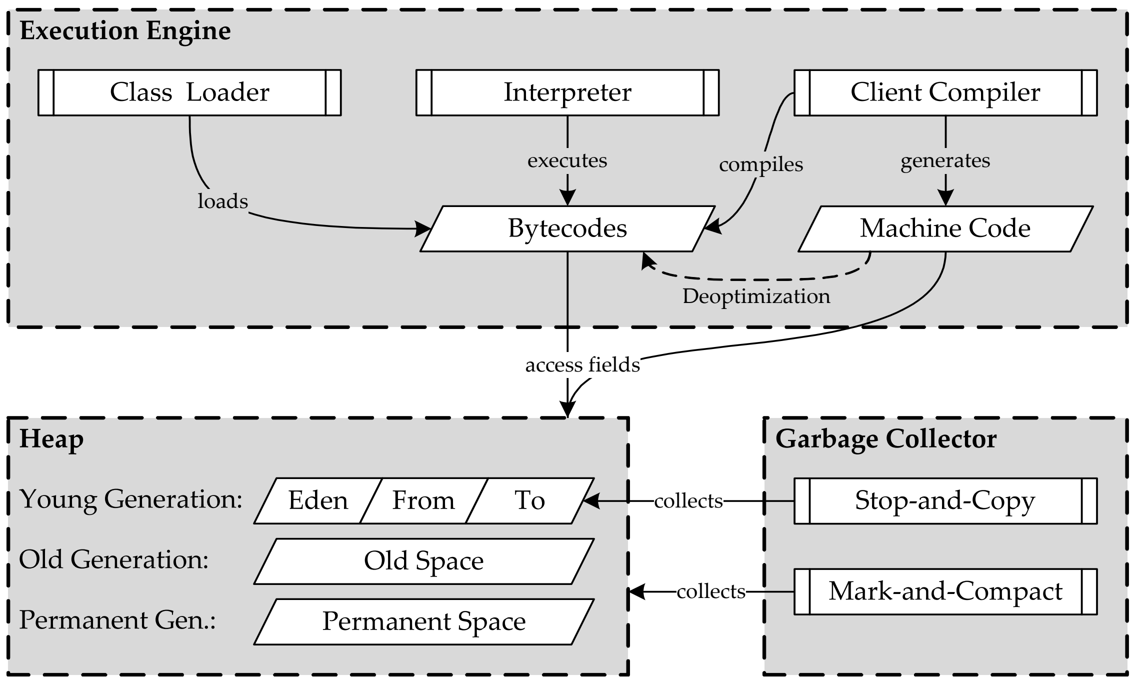


Figure 2.1: System structure of the Java HotSpot VM

Compiling all methods before their first execution would not be beneficial, because typical applications spend most of their time executing only a small fraction of the methods. For the majority of the methods, the time necessary for their compilation would be higher than the time spent interpreting them. The overall performance would be worse because the compile time adds to the run time. Therefore, only methods whose invocation counters exceed a certain threshold are compiled. If a method contains a long-running loop, switching from interpreted to compiled code is also possible in the middle of the method using *on-stack-replacement* [Hölzle94, Fink03].

Compilation is done in the background by a separate thread. The client compiler uses a graph-based high-level intermediate representation (HIR) with an explicit control flow graph for global optimizations, and a low-level intermediate representation (LIR) for linear scan register allocation. It performs aggressive optimizations such as inlining of dynamically bound methods. If an optimization is invalidated later, e.g. due to dynamic class loading, the VM can *deoptimize* [Hölzle92] the machine code and continue the execution of the current method in the interpreter.

The Java programming language is highly object-oriented and encourages the creation of objects even for small intermediate data structures, so the VM must support fast access to objects. To minimize costs of subtype checks and calls to virtual methods, they are highly optimized: subtype checks are implemented with caches covering nearly all checks [Click02], and virtual calls are optimized using polymorphic inline caches [Hölzle91].

The client VM uses a generational garbage collector [Ungar84, Jones96] with two generations, called *young* and *old* generation. Newly created objects are allocated in the *eden space* of the young generation. Since each thread has a separate thread-local allocation buffer (TLAB), object allocations require only few machine instructions in the common case and are multithread-safe without any synchronization (see Section 2.4.3).

When the eden space fills up, the young generation is collected using a stop-and-copy algorithm, which copies all live objects of the eden space and the *from-space* into the *to-space*. Because most objects die young, only few objects survive several copying cycles. Such objects are then promoted to the old generation. When the old generation fills up, the entire heap is collected using a mark-and-compact algorithm.

The *permanent* generation contains internal metadata of the VM such as class descriptors, method descriptors together with the bytecodes, and string constants. Some of these objects are not accessible by Java code, but are nevertheless processed by the mark-and-compact garbage collector. This is important for the unloading of classes.

2.1.1 Client and Server Configuration

The Java HotSpot VM is available for multiple platforms and operating systems. It scales from interactive applications on desktop hardware up to highly parallel applications on multi-processor servers. To achieve this scalability, two different just-in-time compilers, the *client compiler* and the *server compiler*, as well as multiple garbage collection algorithms are available.

The server compiler [Paleczny01] is a highly optimizing compiler tuned for peak performance. It uses a graph-based intermediate representation where control flow, data flow, and memory flow are uniformly modeled by edges between instruction nodes [Click95]. Machine code is generated by a bottom-up rewrite system (BURS, [Pelegri88]) that is based on an abstract architecture description. The sophisticated optimizations based on data flow analyses require much compilation time, which leads to a high overhead during startup when many methods are compiled. This is acceptable for long-running server applications because it impairs performance only during the warm-up phase. If multiple cores are available, several methods can be compiled in parallel.

In contrast to server applications, interactive desktop applications with a graphical user interface demand a low response time, so long delays caused by just-in-time compilation are unacceptable. The client compiler meets this objective by applying only a selected subset of fast and high-impact optimizations. Additionally, the internal structure of the client compiler is easier to understand because the control flow graph and the data flow graph are separated.

The small heap used for client applications requires other garbage collection algorithms than a multi-gigabyte heap used for large server applications. Therefore, the Java HotSpot VM contains various garbage collection algorithms [SunMemory]. All use exact garbage collection techniques, i.e. all objects and all pointers to objects are known and can be traversed by the garbage collector. This permits copying and compacting algorithms that move objects.

The default algorithm for desktop applications is neither parallel nor concurrent, which is sufficient for small heaps. For servers, the parallel algorithms distribute the work among multiple threads to utilize all processors, and the concurrent algorithms allow the user program to continue its execution while dead objects are reclaimed. The selection of the best algorithm is always a tradeoff between the maximum pause time, i.e. the time where no user threads are running, and the throughput, i.e. the overall percentage of time spent in the garbage collector.

2.2 Object Layout

The Java HotSpot VM uses a uniform and handleless memory model for all sorts of objects, including arrays and internal data structures. Implementing object references as direct pointers without using handles provides fast access to instance variables, but requires additional effort during garbage collection. Each object has a header of two machine-words, i.e. 2 * 4 bytes on 32-bit architectures: the *mark word* for internal status information and the *class pointer* as a reference to the type of the object.

2.2.1 Mark Word

The mark word is used as a bit field for multiple purposes such as synchronization and garbage collection. Figure 2.2 shows the usage of the bits in different situations. The two least significant bits are used to implement a *thin locking scheme* [Agesen99, Bacon98]. The most significant bits contain the *identity hash code* of the object. These bits are unused until `System.identityHashCode()` is called on an object.

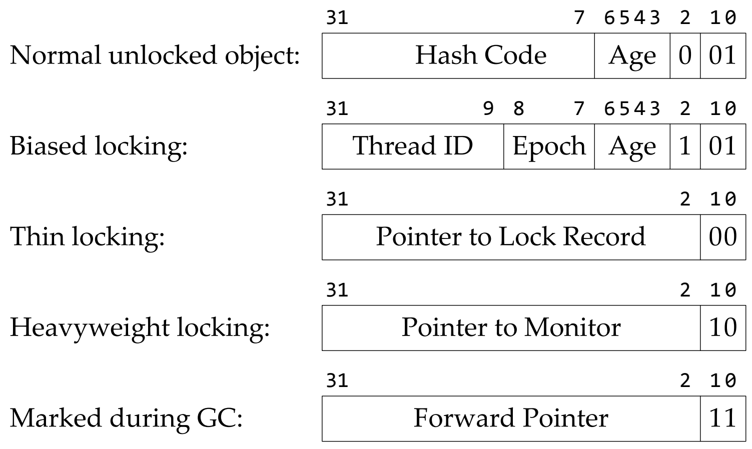


Figure 2.2: Bit usage of the object mark word

Four bits are used to store the *age* of an object, i.e. the number of times it has been copied by the stop-and-copy garbage collection algorithm. When this age reaches a certain threshold, the object is promoted to the old generation. The mark-and-compact algorithm does not use the age bits, but marks objects that are reachable by setting the two least significant bits to 11. When an object is moved by the garbage collector, the new object address, called the *forward pointer*, is also stored in the mark word. If the hash code bits are used, they are rescued before they are overwritten.

In Java, any object can be locked in order to synchronize its access. In the normal unlocked state, the two least significant bits have the value 01, and the remaining bits are free for other purposes. If the object is locked by a single thread only, a pointer to a lightweight data structure located on the stack of the locking thread is stored in the mark word, and the two bits are cleared. If multiple threads lock the object concurrently, a heavyweight monitor that manages the queue of waiting threads is created and a pointer to the monitor is stored in the mark word. In this case, the two least significant bits have the value 10. While an object is locked, the age and hash code are displaced into the locking data structures.

Thin locking is based on the assumption that most objects are not locked concurrently by multiple threads. It saves the VM from creating an expensive monitor data structure, but still requires atomic machine instructions because two threads could try to lock an object simultaneously. *Biased locking* [Russell06], which uses concepts similar to [Kawachiya02], avoids atomic instructions in case the object is always locked by the same thread. The object is biased once towards a thread by installing the *thread id* into the object header. This thread can now lock and unlock the object without atomic instructions. If another thread locks the object, the bias is revoked and the normal locking code is executed. The *epoch* bits are used to make the bias revocation efficient. If the identity hash code is requested for the object, the bias is also revoked because the hash code and the thread id use the same bits of the mark word.

2.2.2 Class Hierarchy

All objects in the garbage collected heap reference a class. The class hierarchy contains not only all currently loaded Java classes, but also the classes for internal metadata objects. Classes are also stored as objects on the heap, therefore they have an 8-byte header and a class pointer themselves.

Figure 2.3 shows the object structure for the classes of a Java instance object, which is shown on the left hand side. The Java object has one instance field that points to another Java object. All objects are aligned at 8-byte boundaries. The object size is padded to be a multiple of eight. In the source code of the virtual machine, *class* is always written as *klass* to avoid the naming conflict with the reserved C++ keyword.

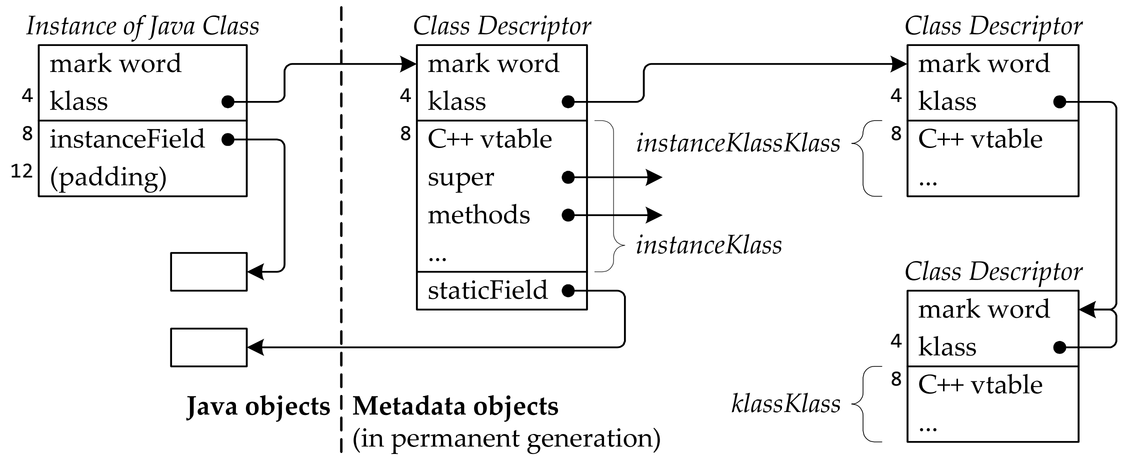


Figure 2.3: Class hierarchy for a Java object

The metadata objects are located in the permanent generation of the heap. All classes are represented by class descriptors that encapsulate different C++ objects, e.g. the class descriptor for a Java object encapsulates an *instanceKlass* object. The C++ object header points to the virtual method table. This allows calling virtual methods on the C++ level, e.g. the virtual method that computes the size of a given object. The size computation for a Java object is implemented in the *instanceKlass*. The class structure is recursive, i.e. the class descriptor has a class pointer itself. For example, the *instanceKlassKlass* computes the size of the instance class descriptor. The end of the recursion is a class descriptor that references itself. The possibility to compute the size for each object on the heap is used e.g. during garbage collection to iterate over all heap objects.

An *instanceKlass* references all data loaded by the class loader, e.g. the super- and subclasses as well as the interfaces implemented by the class. Several other kinds of metadata objects maintain additional information, for example about methods (including the bytecodes of the methods), the constant pool of a class, or symbolic names. The static fields of a class are stored at the end of its class descriptor.

2.3 Garbage Collection

Generational garbage collection is based on the hypothesis that most objects die young [Jones96]. In this case, it is beneficial to separate young objects from long-living objects and collect the area for young objects more frequently. Objects that have survived several collection cycles of a generation are considered long-living and promoted to the next generation. To allow an efficient collection of a young generation, pointers from an older generation to a younger generation must be treated like root pointers. Such pointers are collected in *remembered sets*.

The garbage collection framework of the Java HotSpot VM supports generations collected by different garbage collection algorithms. The default configuration of the client VM uses two generations: the young generation, which is collected by a stop-and-copy algorithm, and the old generation, which is collected by a mark-and-compact algorithm. The remembered set for the young generation is maintained using a card-marking scheme [Hosking93]. Whenever a pointer on the heap is changed, the memory block (card) that contains the pointer is marked as dirty. The stop-and-copy algorithm scans all dirty blocks of the old generation for pointers to the new generation and adds them to the remembered set.

2.3.1 Stop-and-Copy Algorithm

The young generation is separated into three spaces: *eden space*, *from-space*, and *to-space*. The stop-and-copy algorithm copies all live objects of the eden space and the from-space into the to-space. Then, the roles of the from-space and the to-space are exchanged. The size of the from-space and the to-space is equal, the eden space is usually larger. Because most objects die young, only few objects are copied and a small to-space is sufficient for all live objects of both the eden space and the from-space.

Algorithm 2.1 shows the basic StopAndCopy algorithm, which follows the principle described in [Cheney70]. First, all objects referenced by root pointers are copied to the to-space using CopyObject. Allocating memory in the to-space requires only an increment of the end pointer. Each object that has been copied stores a forward pointer to its new location, which is encoded in the mark word (see Figure 2.2). All objects referenced by copied objects are also alive and must be copied as well. The algorithm uses the to-space as a queue and scans all copied objects in sequential order. The forward pointer is used to prevent copying an object twice.

```
StopAndCopy
    toSpace.end = toSpace.begin
    for each root pointer r do
        r = CopyObject(r)
    end for

    obj = toSpace.begin
    while obj < toSpace.end do
        for each reference r in obj do
            r = CopyObject(r)
        end for
        obj += obj.size
    end while

CopyObject(obj)
    if obj.forwardPtr is set then
        return obj.forwardPtr
    end if

    newObj = Allocate(obj, obj.size)

    memmove(obj, newObj, obj.size)
    obj.forwardPtr = newObj
    return newObj

Allocate(obj, size)
    if obj.age < threshold then
        newObj = toSpace.end
        toSpace.end += size
        obj.age++
    else
        newObj = oldGeneration.end
        oldGeneration.end += size
    end if

    return newObj
```

Algorithm 2.1: Stop-and-copy algorithm for collection of the young generation

This breadth-first copying scheme is simple and efficient, but it leads to a random order of objects in the to-space. An object is copied when the first reference to it is scanned. A depth-first copying scheme, where all referenced objects are copied immediately after the object itself, would require an explicit stack of objects.

When the object has survived several copying cycles, it is promoted to the old generation. The threshold for the age is based on heuristics that use the age distribution of the young generation. Allocation in the old generation is also a simple pointer increment because there are no gaps between objects in the old generation due to the mark-and-compact algorithm.

2.3.2 Mark-and-Compact Algorithm

The mark-and-compact algorithm processes the entire heap of the VM. Java objects of the young and the old generation are compacted into the old generation, and the metadata objects of the permanent generation are compacted within the permanent generation, i.e. they are not mixed with Java language objects. After the compaction, all live objects are concentrated at the beginning of the old and the permanent generation. In contrast to the stop-and-copy algorithm, the order of objects is preserved.

```
MarkAndCompact
    MarkLiveObjects
    ComputeNewAddresses
    AdjustPointers
    MoveObjects

MarkLiveObjects
    for each root pointer r do
        MarkAndPush(r)
        FollowMarkStack
    end for

FollowMarkStack
    while not markStack.empty do
        obj = markStack.pop
        for each reference r in obj do
            MarkAndPush(r)
        end for
    end while

MarkAndPush(obj)
    if not obj.marked then
        obj.marked = true
        markStack.push(obj)
    end if

ComputeNewAddresses
    newObj = space.begin
    for each marked object obj do
        obj.forwardPtr = newObj
        newObj += obj.size
    end for

AdjustPointers
    for each root pointer r do
        r = r.forwardPtr
    end for

    for each marked object obj do
        for each reference r in obj do
            r = r.forwardPtr
        end for
    end for

MoveObjects
    for each marked object obj do
        newObj = obj.forwardPtr
        memmove(obj, newObj, obj.size)
    end for
```

Algorithm 2.2: Mark-and-compact algorithm for full collection

Algorithm 2.2 shows the four phases of the mark-and-compact algorithm. In the first phase MARKLIVEOBJECTS, the heap is traversed starting with the root pointers to mark all live objects. A stack of objects is used to avoid recursive calls. The mark word of an object contains the marking state and the forward pointer (see Figure 2.2), therefore it is saved to an auxiliary data structure in case a hash code or information for biased locking is present.

In the next phase COMPUTENEWADDRESSES, the new addresses for the marked objects are computed. All marked objects of the heap are iterated. New addresses are assigned increasingly so that the gaps between live objects are removed. Therefore, objects can only move towards the beginning of the heap, i.e. newObj is always less than or equal to obj. The new address is encoded as the forward pointer in the mark word. The third phase ADJUSTPOINTERS updates all root pointers and pointers inside objects to the new addresses, which are temporarily stored in the forward pointers of the referenced objects. Finally, MOVEOBJECTS copies the contents of the objects to the new location. The memory of the new location can be overwritten without precautions because objects move only towards the beginning of the heap.

2.4 Client Compiler

The client compiler [Kotzmann08] aims at a low compilation time and a small memory footprint. Therefore, global optimizations that would require complex data flow analyses or that could bloat the size of the generated machine code are omitted. The compiler performs only cheap and high-impact global optimizations, such as global value numbering or register allocation [Muchnick97]. Figure 2.4 shows the structure of the client compiler. The front end builds the *high-level intermediate representation* (HIR) and performs global optimizations. The back end uses the *low-level intermediate representation* (LIR) for register allocation and finally emits the machine code.

2.4.1 Bytecodes

Java source code [Gosling05] is first compiled to platform-independent bytecodes. This frees the VM from the time-consuming task of parsing and analyzing plain-text source code. The bytecodes provide a binary representation of the class that can be directly executed by an interpreter. It also simplifies validity checks, as defined in the bytecode specification [Lindholm99].

The example in Figure 2.5 shows the Java source code fragment of a class `Polyline` with a method `addPoint()`, and the Java bytecodes created for this method. The bytecodes are stored in a compact binary form. The number to the left of each bytecode refers to its index relative to the beginning of the method. It is called the *bytecode index* (bci).

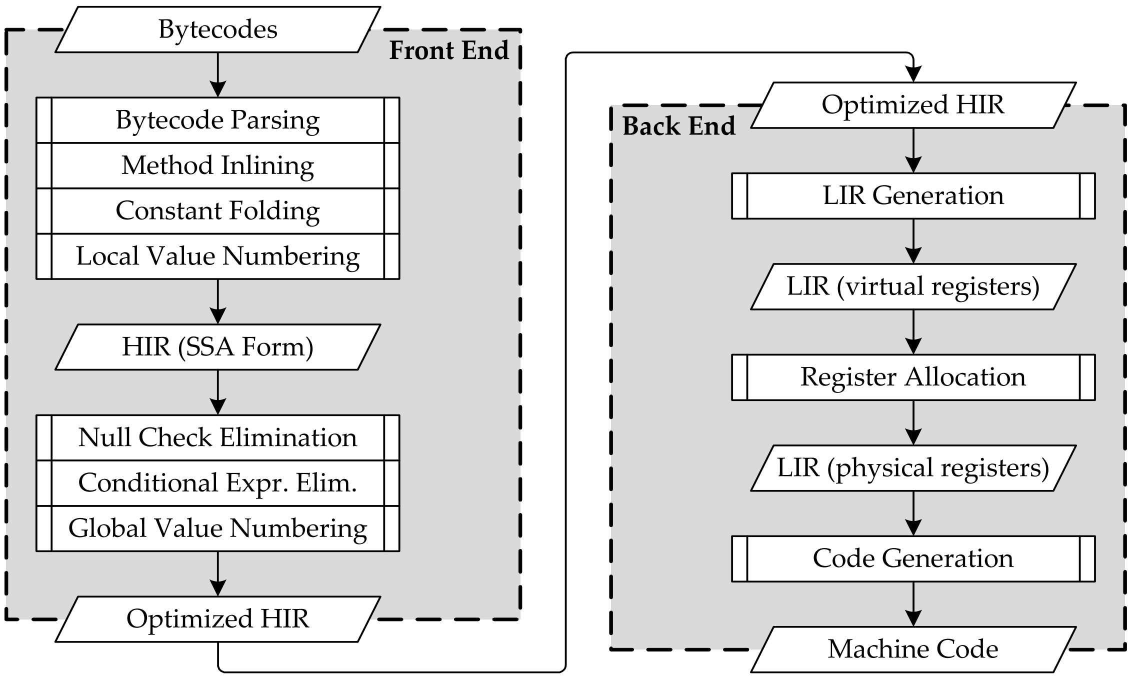


Figure 2.4: Structure of the client compiler

```
class Polyline {
  List<Point> points;

  void addPoint(int x, int y) {
    Point p = new Point(x, y);
    points.add(p);
  }
}
```

```
 0: new at.ssw.Point
 3: dup
 4: iload_1
 5: iload_2
 6: invokespecial at.ssw.Point.<init>
 9: astore_3
10: aload_0
11: getfield at.ssw.Polyline.points
14: aload_3
15: invokeinterface java.util.List.add
20: pop
21: return
```

Figure 2.5: Compilation example—Java source code and Java bytecodes

The operation to be performed is encoded in the first byte of an instruction. Symbolic references to names of classes and fields are stored in the constant pool of the class. The operands of the allocation and method invocation bytecodes in the example are two-byte indices into the constant pool. Bytecodes are executed using an operand stack, therefore most operands are implicitly passed on this stack. The bytecodes can be divided into the following categories:

- Loads and stores for local variables, fields, array elements, and constants.
- Arithmetic and logical instructions.
- Instructions for type conversions of scalar and reference types.
- Conditional and unconditional jumps.

- Call and return instructions.
- Instructions that directly manipulate the operand stack.
- Instructions for synchronization of threads, exception handling, and allocation.

2.4.2 Front End

The high-level intermediate representation (HIR) is a graph-based representation of the method. Instructions refer to their operands using pointers to the instructions that compute these operands. Instructions are grouped into basic blocks, i.e. longest possible sequences without jumps or jump targets in between. An explicit control flow graph connects the basic blocks. The HIR is in *static single assignment* (SSA) form [Cytron91, Bilardi03], which means that the value of a variable is not changed after its first assignment, i.e. every assignment creates a new variable. When control flow joins, *phi functions* merge values coming from different predecessor blocks.

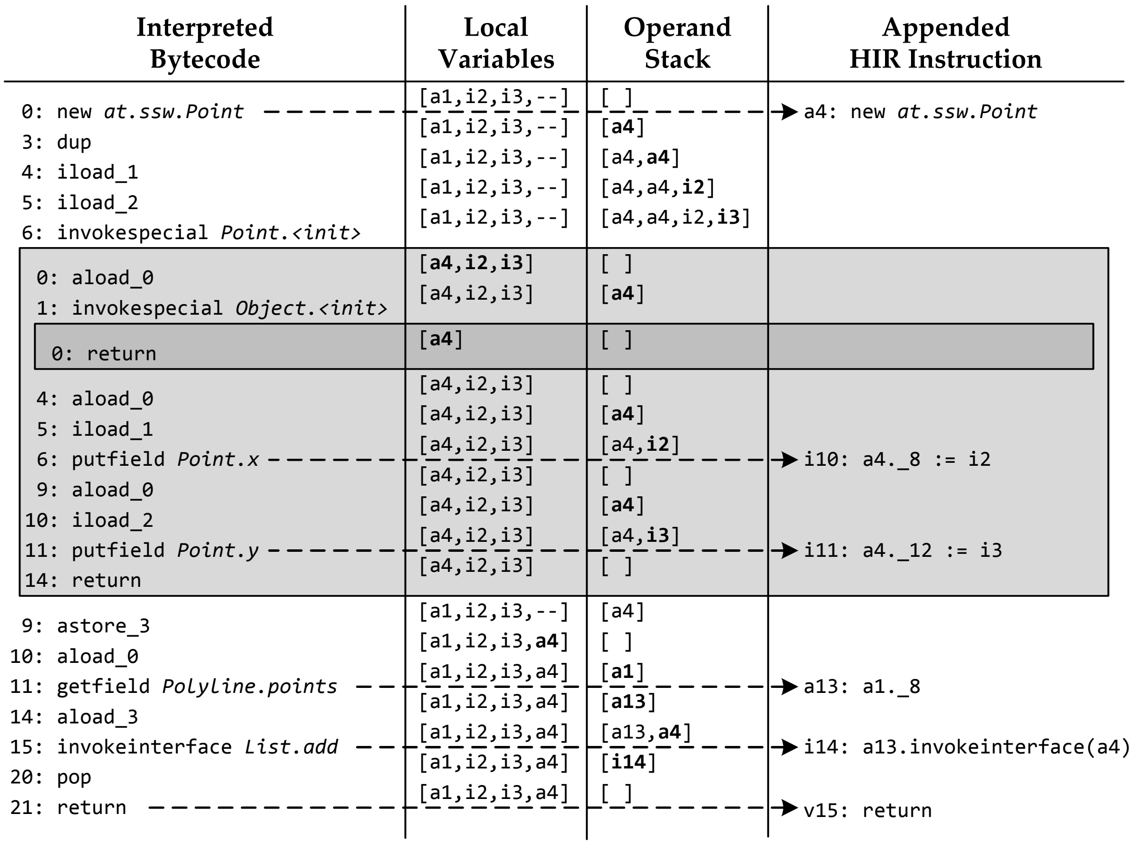
The HIR is constructed by abstract interpretation of the bytecodes. Several local optimizations are applied during parsing: *method inlining* replaces the call to a short method by a copy of the method's instructions, *constant folding* simplifies arithmetic instructions with constant operands, and *local value numbering* eliminates common subexpressions within a block.

Figure 2.6 shows the HIR for the method `addPoint()` in Figure 2.5. The method has no jumps, so all instructions are contained in a single block. Each instruction has a unique *id* number, preceded by the type *t* of the instruction (`i` for integer, `a` for object, and `v` for void). The instructions `a1`, `i2`, and `i3` represent the method parameters `this`, `x`, and `y`. The instructions `a4`, `a13`, `i14`, and `v15` directly originate from the bytecodes. The instructions `i10` and `i11` are the result of method inlining of the constructor `Point.<init>()`. The constructor assigns the two parameters to the fields `x` and `y` of the newly allocated `Point` object. The symbolic field names of the instructions `i10`, `i11`, and `a13` are already resolved to byte-offsets of the fields: 8, 12, and 8.

```
bci___tid__instruction_________________________________________
0     a4   new at.ssw.Point
6.6   i10  a4._8 := i2 at.ssw.Point.x
6.11  i11  a4._12 := i3 at.ssw.Point.y
11    a13  a1._8 at.ssw.PolyLine.points
15    i14  a13.invokeinterface(a4) java.util.List.add
21    v15  return
```

Figure 2.6: Compilation example—high-level intermediate representation (HIR)

The HIR does not need instructions for loading and storing local variables or for the manipulation of the operand stack. When a local variable is stored, the pointer to the instruction creating the value is put into the state array for the variables. A later load of the variable pushes this instruction onto the operand stack, so that the instruction that pops the stack can use the pointer. Figure 2.7 shows the generation of the HIR. The left hand side shows the bytecodes, the right hand side the HIR instructions. In between, the simulated state of the local variables and the operand stack is illustrated. The gray boxes refer to inlined methods, which have their own state.



Interpreted Bytecode	Local Variables	Operand Stack	Appended HIR Instruction
0: new *at.ssw.Point*	[a1,i2,i3,--]	[ ]	a4: new *at.ssw.Point*
3: dup	[a1,i2,i3,--]	[**a4**]	
4: iload_1	[a1,i2,i3,--]	[a4,**a4**]	
5: iload_2	[a1,i2,i3,--]	[a4,a4,**i2**]	
6: invokespecial *Point.<init>*	[a1,i2,i3,--]	[a4,a4,i2,**i3**]	
0: aload_0	[**a4,i2,i3**]	[ ]	
1: invokespecial *Object.<init>*	[a4,i2,i3]	[**a4**]	
0: return	[**a4**]	[ ]	
4: aload_0	[a4,i2,i3]	[ ]	
5: iload_1	[a4,i2,i3]	[**a4**]	
6: putfield *Point.x*	[a4,i2,i3]	[a4,**i2**]	i10: a4._8 := i2
9: aload_0	[a4,i2,i3]	[ ]	
10: iload_2	[a4,i2,i3]	[**a4**]	
11: putfield *Point.y*	[a4,i2,i3]	[a4,**i3**]	i11: a4._12 := i3
14: return	[a4,i2,i3]	[ ]	
9: astore_3	[a1,i2,i3,--]	[a4]	
10: aload_0	[a1,i2,i3,**a4**]	[ ]	
11: getfield *PolyLine.points*	[a1,i2,i3,a4]	[**a1**]	a13: a1._8
14: aload_3	[a1,i2,i3,a4]	[**a13**]	
15: invokeinterface *List.add*	[a1,i2,i3,a4]	[a13,**a4**]	i14: a13.invokeinterface(a4)
20: pop	[a1,i2,i3,a4]	[**i14**]	
21: return	[a1,i2,i3,a4]	[ ]	v15: return

Figure 2.7: Compilation example—construction of the HIR

After the generation of the HIR, global optimizations are performed. *Null check elimination* (see e.g. [Kawahito00]) removes `null` checks if the compiler can prove that an accessed object is non-`null`. For example, all field accesses in the example method are guaranteed to throw no exception. *Conditional expression elimination* replaces the common code pattern of a branch that loads one of two values by a conditional move instruction. *Global value numbering* (see e.g. [Briggs97]) eliminates common subexpressions across basic block boundaries.

2.4.3 Back End

The back end transforms the optimized HIR to the low-level intermediate representation (LIR). It allows platform-independent optimizations that would be difficult to implement directly on machine code. The LIR operations are shared between all target platforms, but the LIR generation already contains platform-dependent code. Each basic block stores a list of LIR operations.

Figure 2.8 shows the LIR for the HIR instructions of Figure 2.6. LIR operations use explicit operands that can be virtual registers, physical registers, memory addresses, stack slots, or constants. The LIR is conceptually similar to machine code, augmented with some higher level operations, e.g. the `alloc_obj` operation for object allocation. This operation requires several temporary registers that are also specified as operands. When a machine instruction requires a dedicated register such as `eax`, the generated LIR operation already references this physical register. The other operands are virtual registers, printed as `R` followed by the virtual register number. In the example, the virtual registers `R41`, `R42`, and `R43` represent the three method parameters `this`, `x`, and `y`.

```
nr__operation____________________________________________
14  move obj:at.ssw.Point -> edx
16  alloc_obj edx, ecx, esi, size:16 -> eax
18  move eax -> R44
20  move R42 -> [R44 + 8]
22  move R43 -> [R44 + 12]
24  move [R41 + 8] -> R45
26  move R44 -> edx
28  move R45 -> ecx
30  virtual_call ecx, edx -> eax
32  return
```

Figure 2.8: Compilation example—low-level intermediate representation (LIR)

Register allocation replaces the virtual registers of the LIR with physical ones. The most commonly used approach, which is based on graph-coloring (see e.g. [Chaitin81, Briggs94, Muchnick97]), would be too slow for the client compiler, therefore the linear scan algorithm [Poletto99, Traub98] is used. First, all blocks are sorted topologically. For each virtual register of the LIR, a *lifetime interval* is constructed. *Fixed intervals* are built for the physical register operands to model register constraints of the target architecture and method calls. *Use positions* of an interval refer to the operations that read or write a certain register.

The allocation algorithm processes the lifetime intervals in the order of increasing start positions. Each interval is assigned a register that is not used by another simultaneously live interval. When more intervals are live than physical registers are available, intervals are split and spilled, i.e. get a stack slot assigned. Heuristics for the split positions, register hints, and spill store elimination reduce the number of necessary move instructions for spilling [Wimmer05].

Figure 2.9 shows the lifetime intervals for the example. Gray rectangles represent live ranges and black bars indicate use positions. The intervals for the seven physical registers are sketched in the first line. The lines for the five virtual registers contain the assigned physical registers. No spilling is necessary because three registers are free at the allocation operation 16, and no intervals are live at the method call 30.

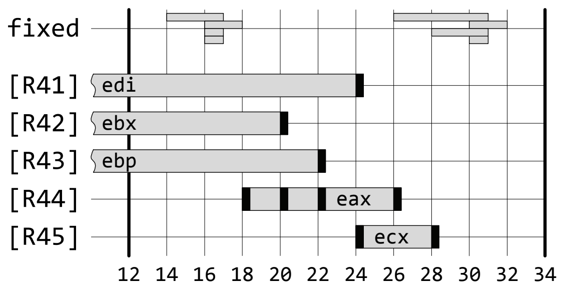


Figure 2.9: Compilation example—lifetime intervals

After register allocation, each LIR operation is mapped to one or more machine instructions by the code generator. The register constraints of the target architecture are already satisfied. Many LIR operations can be divided into a common and an uncommon case. Examples for uncommon cases are throwing bounds check exceptions for array accesses or invoking the garbage collector for memory allocations. The machine instructions for the common case are emitted in-line, while the instructions for the uncommon case (called *slow path*) are emitted out-of-line at the end of the method.

The generated machine code is augmented with metadata for the garbage collector and the deoptimization framework. The exact garbage collection algorithms require information about the locations of all object pointers. The so-called *object maps* specify the registers and spill slots that contain object pointers. Object maps are stored for certain code locations (called *safepoints*) where garbage collection can happen. Examples for safepoints are backward branches, method calls, return instructions, and allocation instructions. The machine code and the metadata are stored together in a *native method object*, often called *nmethod*.

Figure 2.10 shows the machine code for object allocation when the example method is compiled for the Intel IA-32 platform [Intel07]. Each thread has a *thread-local allocation buffer* (TLAB), a fragment of the *eden space* committed to the thread. This allows a thread-safe allocation without atomic operations in the fast path. The address of the current thread is loaded from a data structure also used for exception handling. This requires only two `mov` instructions.

```
00B919A0  mov  ecx, ptr fs:[0]     // load exception handler
00B919A7  mov  ecx, ptr [ecx-0Ch]  // load current thread
00B919AA  mov  eax, ptr [ecx+44h]  // load TLAB top (=object address)
00B919AD  lea  esi, [eax+10h]      // add size of object (16 bytes)
00B919B0  cmp  esi, ptr [ecx+4Ch]  // check TLAB overflow
00B919B3  ja   00B919FF            // slow path: possible GC
00B919B9  mov  ptr [ecx+44h], esi  // save new TLAB top
...       // initialization of the object (class and mark word)

00B919FF  // slow path: call run-time stub to refill TLAB or invoke GC
```

Figure 2.10: Compilation example—fragment of the machine code

The current top pointer of the TLAB is loaded, incremented by the size of the object, and compared with the end pointer of the TLAB. If the new top is below the end, object allocation was successful. Otherwise, the out-of-line slow path is executed. It tries to refill the TLAB, i.e. to allocate one large chunk of memory from the eden space that is then used for multiple subsequent allocations of this thread. This allocation requires an atomic operation. If no more memory is available in the eden space, the slow path invokes the garbage collector. Figure 2.11 shows the TLABs for two threads.

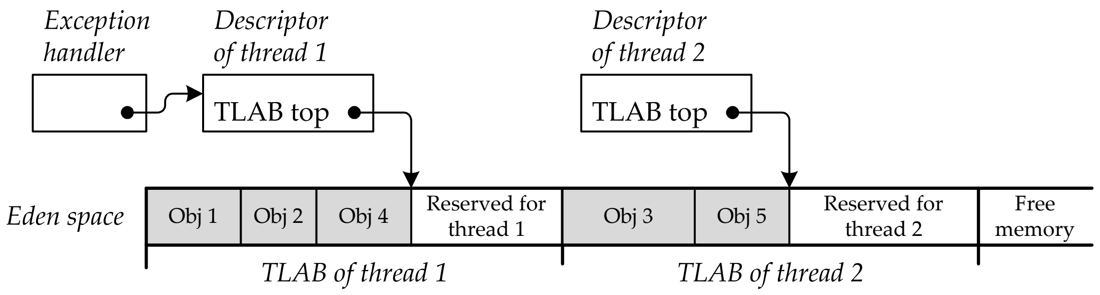


Figure 2.11: Object allocation using a thread-local allocation buffer (TLAB)

2.5 Deoptimization

Optimizations in the just-in-time compiler of a Java VM are complicated by the dynamic class loading of Java. Because additional classes can be loaded at any time, global information about the class hierarchy can change. For example, the just-in-time compiler can inline a dynamically bound method if class hierarchy analysis [Dean95] finds out that only one suitable method exists. This optimization can be invalidated by loading a class that provides another suitable method. To avoid constraints for method inlining such as *preexistence* [Detlefs99], compiler optimizations can be reverted using *deoptimization* [Hölzle92]. The optimized machine code is discarded and the execution of the method is continued in the interpreter.

Figure 2.12 shows an example for method inlining that requires deoptimization. Assume that the method `create()` normally returns instances of the class `A`. The subclass `B` is still unloaded when the method `foo()` is compiled. The method `A.bar()` is inlined by the compiler although it is a virtual method because it is the only possible method that can be called at this time. If the class `B` is loaded later and the method `create()` returns an instance of `B`, the inlining decision turns out to be wrong.

```
void foo() {            class A {               class B extends A {
  A a = create();         void bar() { ... }      void bar() { ... }
  a.bar();              }                       }
}
```

Figure 2.12: Example that requires deoptimization

The machine code of the method `foo()` is discarded. Later, the method is recompiled without inlining `A.bar()` so that new invocations of the method are correctly executed. However, it is also necessary to stop all activations of `foo()` that are further up the method call stack. This is done by inserting a call to a runtime function at the point where `foo()` will be continued in every activation. The runtime function removes the stack frame of the compiled method and builds the correct interpreter stack frames that continue the execution.

To fill the local variables and the operand stack of the interpreter, the location of the variables must be known. A variable can be either in a register or spilled in the stack frame of the compiled method. The exact location is tracked by the just-in-time compiler and stored in the metadata together with the machine code of the method. This information is called *debugging information*.

The locations of all variables are recorded at all possible deoptimization points, e.g. all method calls. The front end of the client compiler saves a copy of the state array containing the HIR instructions of the local variables and the operand stack. The back end replaces this information with the results of register allocation, i.e. the assigned physical registers and stack slots. If the deoptimization point is inside an inlined method, the state of both the enclosing method and the inlined method is maintained so that two interpreter stack frames can be reconstructed.

Converting a compiled frame to interpreted frames is an expensive operation. However, it occurs rarely because most classes are already loaded before a method is compiled, so the run-time costs of deoptimization are irrelevant in practice. Deoptimization is generally useful for all kinds of optimistic optimizations. It simplifies and accelerates the generated machine code because no code for uncommon situations must be emitted. For example, an optimistic array bounds check elimination based on deoptimization can assume that no bounds check fails and deoptimize if this assumption does not hold [Würthinger07b].

Chapter 3

Architecture of Object and Array Inlining

This chapter presents the overall system structure of our feedback-directed object inlining system. It starts with the definition of some important terms. Then, it presents the design principles that influence the architecture and explains the reasons for these principles. The chapter closes with an overview of all subsystems necessary for object inlining. The subsequent chapters present the details and algorithms of these subsystems.

The overall system architecture for a feedback-directed optimization must balance the costs for collecting profiling data and applying the algorithms at run time with the speedup of the improved machine code. Instead of optimizing everything that is possible, it is often necessary to be conservative in order to reduce the costs.

The algorithms of our object inlining system are mostly integrated into existing subsystems of the Java HotSpot VM, coordinated by a small optimization core. The phases are performed asynchronously by the just-in-time compiler and the garbage collector. There is no central component that performs a global data flow analysis. This results in a system with a low run-time overhead.

Our system can be divided into four parts: We use *read barriers* to detect frequently accessed fields at run time. The modified garbage collector uses this information for *object colocation*, i.e. to place related objects next to each other in memory. *Object inlining* then removes the field loads by address arithmetic. *Array inlining* considers the differences between objects and arrays, e.g. the variable size of arrays.

3.1 Source Code of Example

The example used throughout this thesis is a fragment of an object-oriented drawing application. The class `Polyline` uses several helper classes to manage the line color and the list of points. Figure 3.1 shows the relevant fragments of the Java source code. The class `Color` has a field `rgb` that stores the actual red, green, and blue color components encoded as an integer value. Encapsulating the single value in its own class is reasonable, because this way several reusable methods operating on the color value can be defined. The class `Point` contains two fields that store the `x` and `y` coordinates.

```java
class Color {
  int rgb;
}

class Point {
  int x, y;
}

class Polyline {
  List<Point> points;
  Color lineColor;

  Polyline() {
    points = new ArrayList<Point>();
    lineColor = new Color();
  }

  int getLineColor() {
    return lineColor.rgb;
  }

  Point getPoint(int index) {
    return points.get(index);
  }

  void addPoint(Point newPoint) {
    points.add(newPoint);
  }
}

class Test {
  void allocate() {
    Polyline poly = new Polyline();
    // Do something with poly
  }
}
```

```java
interface List<E> {
  E get(int index);
  boolean add(E e);
}

class ArrayList<E> implements List<E> {
  int modCount;
  int size;
  Object[] elementData;

  ArrayList() {
    elementData = new Object[10];
  }

  E get(int index) {
    if (index >= size) throw new ...
    return (E) elementData[index];
  }

  boolean add(E e) {
    modCount++;
    if (size+1 > elementData.length) {
      int newCapacity = ...
      elementData = Arrays.copyOf(
            elementData, newCapacity);
    }
    elementData[size++] = e;
    return true;
  }
}
```

Figure 3.1: Java source code of the example classes

The dynamic list of points is managed by a collection of the Java class library. The source code of the interface `List` and the class `ArrayList` is simplified and reduced to the methods relevant for the example. The class `ArrayList` uses an `Object[]` array for the data, referenced by the field `elementData`. The field `size` contains the number of array elements currently in use. When a new element is added using the method `add()` and the capacity of the array does not suffice, the array is replaced by a larger copy, i.e. the field `elementData` is changed. The field `modCount` is used to detect concurrent modifications while the list is iterated. It is not relevant for this thesis, but the size of `ArrayList` objects and the field offsets reported in the thesis would differ with the actual values if the field were omitted.

Figure 3.2 shows an object graph for these classes, consisting of one `Polyline` object with its `Color` and its `ArrayList`, which references the `Point` objects via the `Object[]` array. The numbers on the left hand side of each field show the field offsets in bytes.

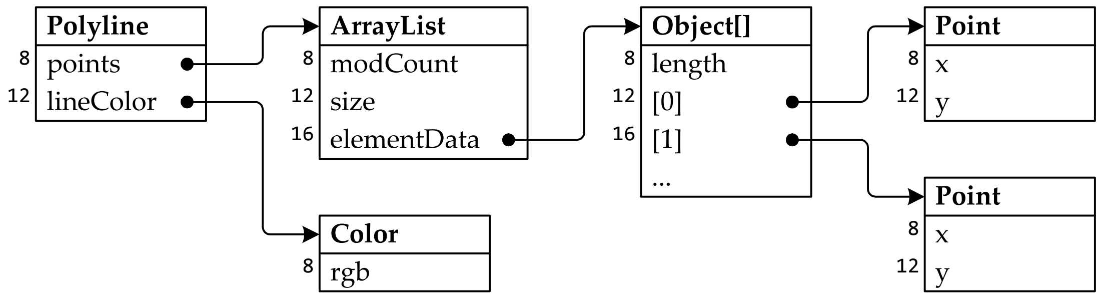


Figure 3.2: Object graph for example classes

The field `points` of the class `Polyline` is declared using the interface type `List` instead of the implementation class `ArrayList`. This allows the list implementation to be exchanged by modifying only one line of the source code. For our object inlining algorithm, the declared type is irrelevant and only the implementation class is considered. The generic parameter `<Point>` improves the type safety of the Java code, but does not have any impact on the bytecodes, i.e. the `ArrayList` still uses an `Object[]` array and not a `Point[]` array.

The class `Polyline` has three methods to access and modify the color and the points. Additionally, constructors are treated like methods with the special name `<init>` in the bytecodes and the virtual machine. They are invoked using the bytecode `invokespecial`, which binds the constructors statically.

3.2 Definition of Terms

Object inlining operates on groups of heap objects that are in a parent-child relationship. An *inlining parent* contains a reference to the *inlining child*. A child has exactly one parent, but a parent can have references to multiple children. Additionally, inlining hierarchies can exist, i.e. an inlining child can in turn be another inlining parent. Consequently, an inlining parent, its direct children, and all its indirect children form a single group of objects.

The inlining parent always stores a reference that points to the inlining child. If the inlining parent is an object, the reference to the inlining child is a field declared in the class of the inlining parent. Such a field is called an *inlined field*. The declared type of the field can be either an object type or an array type. We therefore distinguish between *object fields* and *array fields*. From the point of view of the bytecodes, there is no difference between object and array fields because the superclass of all arrays is `Object`. However, objects and arrays have different characteristics that are relevant for inlining. For example, the size of an object is constant, while the size of an array is unknown until the allocation.

If the inlining parent is an array, the inlining child is referenced by an *array element*. In our approach, array elements cannot be inlined because the Java bytecodes for array accesses do not have enough static type information. A global data flow analysis would be necessary for the efficient handling of arrays as inlining parents. Section 7.2 on page 83 explains the details of this constraint.

3.2.1 Example

Figure 3.3 illustrates the defined terms on a graph of objects and arrays that form an inlining hierarchy. The `Polyline` object is the inlining parent. Its inlined object field `points` references the `ArrayList` object, which is the inlining child of the `Polyline`. It is also an inlining parent itself because of its inlined array field `elementData`. The `Object[]` array referenced by this field is an inlining child. The references to the `Point` objects are not inlined because inlining of array elements is not possible with our approach.

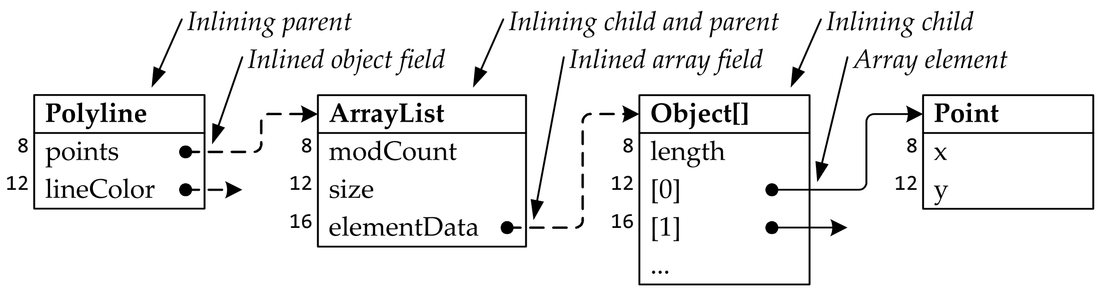


Figure 3.3: Definition of terms

A parent object can have multiple inlined fields. In our example, the field `lineColor` of the `Polyline` object is also inlined, so the `Polyline` object has two inlining children. In the graphical representation of object graphs, we highlight inlined fields by dashed lines, while non-inlined fields and array elements are drawn as solid lines.

3.2.2 Method Inlining

Object inlining must not be confused with *method inlining*. Method inlining is a compiler optimization that replaces a call to a method by a copy of the method body. This eliminates the overhead of method dispatching. It is beneficial if a small method like an accessor method for a field is called. The high overhead of the call compared to the execution time of the method is eliminated, and the total size of the machine code increases only slightly.

Additionally, method inlining supports other compiler optimizations. When processing a call instruction, the compiler must either conservatively assume that the called method has side effects, e.g. that it modifies a certain field, or use interprocedural information to ensure that the method has no side effects. As a result, method inlining can be used to avoid the expensive computation of interprocedural information. Aggressive inlining of some larger methods can simplify optimization algorithms because it avoids the complicated handling of possible side effects. We use this e.g. for the inlining of constructors.

3.3 Design Principles

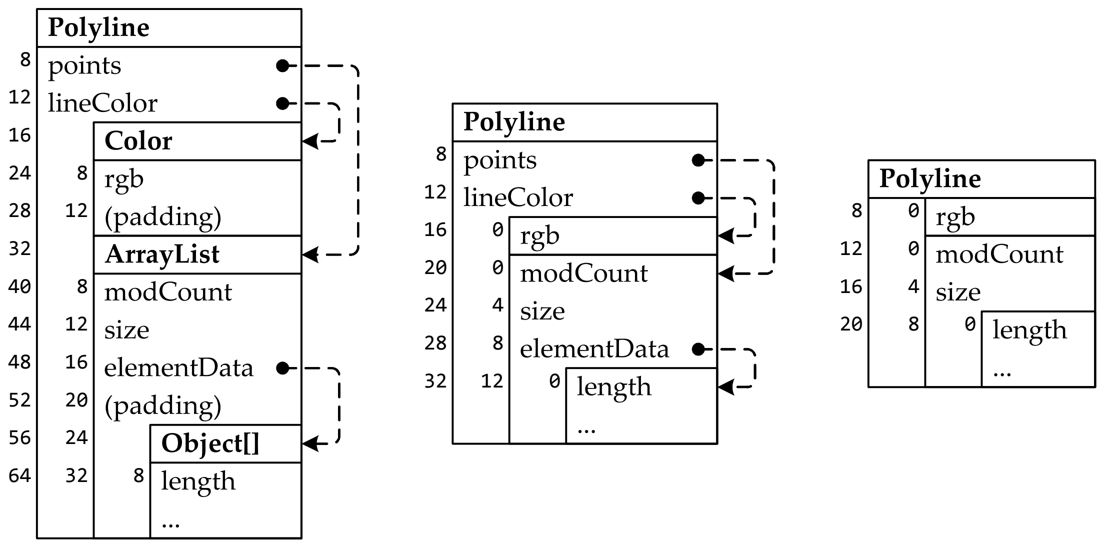
Our object inlining system is based on the design principles listed in the following enumeration. We believe that many of these principles are important not only for object inlining, but generally applicable for optimizations inside a Java virtual machine.

- *Automatic*: The optimization does neither require any actions on the part of the programmer nor any special tools at compile time or deployment time. All analysis and optimization steps are performed automatically at run time. Without this principle, it would be difficult to execute existing applications whose source code is not available.
- *Dynamic*: We fully support dynamic class loading because it is a key feature of modular Java applications. State-of-the-art application frameworks such as the Eclipse Rich Client Platform [Eclipse08] or the NetBeans Platform [NetBeans08] have a small application kernel that performs lazy loading of most application classes. This improves the startup speed of large applications. In contrast, we handle other dynamic features of Java conservatively, e.g. fields that are modified using reflection are not considered for object inlining because the handling would be too expensive.
- *Feedback-directed*: We use profiling data collected at run time to decide which fields should be optimized. This saves the programmer from annotating fields that he considers to be important and allows optimizations across modules and libraries. Field access statistics are collected using lightweight read barriers that increment access counters.
- *No global data flow analysis*: A global analysis, e.g. building a global call graph, is complicated in Java because most methods are dynamically bound and new classes can be loaded at a later time. Instead of a complex and expensive algorithm that can handle the dynamic features, we use only local information collected e.g. by the class loader and the just-in-time compiler.
- *Optimization on per-class basis*: All analysis and optimization steps operate on a per-class basis, i.e. either all or no objects of a certain class are optimized. This reduces the overhead as it is not necessary to distinguish between optimized and unoptimized objects of the same class.

- *Limit changes to few subsystems of the VM*: Most of our modifications concentrate on two parts of the Java HotSpot VM: the just-in-time compiler and the garbage collector. Some subsystems, for example the class loader, the interpreter, and the internal representation of classes, contain small changes such as notifications when new classes are loaded. Many other subsystems remain unchanged, including the locking scheme for synchronization on objects as well as the handling of threads and safepoints.

3.3.1 Memory Layout

In the Java HotSpot VM, every object has an 8-byte header for internal status information and for the reference to the class of the object (see Section 2.2 on page 12). When objects are grouped together by object inlining, the header for child objects can be either eliminated or preserved. Similarly, the 8-byte alignment of child objects is optional. Figure 3.4 a) and b) illustrate the differences between these two cases. Eliminating the object headers of the three child objects saves 24 bytes. Eliminating the padding of the `Color` and the `ArrayList` objects saves another 8 bytes, so the optimized group of objects is 32 bytes smaller than the original objects.



a) Unmodified layout b) Inlining without headers c) Inlining without fields

Figure 3.4: Possible memory layouts for inlined objects

When the children do not conform to the standard object layout, there is no reason to keep the internal pointers to them. Removing the fields `points`, `lineColor`, and `elementData` reduces the size of the object group by additional 12 bytes. In total, object inlining can save 44 bytes of memory for each `Polyline` object group. Figure 3.4 c) shows the resulting memory layout.

Reducing the object size improves the cache behavior and reduces the pressure on the garbage collector. Nevertheless, we use the unmodified layout for object inlining as shown in Figure 3.4 a) because this has several advantages:

- *Object locking*: The mark word of the object header is used for synchronization. It must be guaranteed that the child object is never locked before the header can be removed.
- *Side pointers to children*: External references to the child object, e.g. from another object, expect the header to be present. Removing the object header would change the field offsets and thus disallow such references. Additionally, it would not be possible to pass a child object as a method parameter.
- *No change of class metadata*: With the unmodified layout, the parent and its children are distinct objects from the garbage collector's point of view. Therefore, the class metadata remains unchanged. The elimination of object headers and fields would lead to a new kind of heap elements that are a mixture of objects and arrays. For example, the `Polyline` object would consist of an object part with three fields and an array part whose array length is stored at the unusual offset 20.
- *No change of the interpreter*: When the object layout remains unchanged, the interpreter can access child objects by following the field pointers and using the normal field offsets. Therefore, it is not necessary to compile all methods that load inlined fields.
- *Smooth transition to optimized code*: Our system performs the optimization steps asynchronously. The garbage collector builds the optimized object order before all preconditions are satisfied, and methods with optimized field loads are compiled gradually after the preconditions are satisfied. Changing the object layout would require a distinct point where all affected objects are rewritten and the accessing methods are compiled.
- *Smooth deoptimization*: A newly loaded class can invalidate preconditions of already optimized fields. In addition to the deoptimization of the machine code with optimized field loads, it would be necessary to restore the removed object headers and field pointers. When the memory layout is not changed, it is not necessary to modify the heap.
- *Support for reverse object inlining*: When a parent object's class has subclasses, we place inlining children in front of the parent (see Section 6.8 on page 76). Removing the header of child objects would lead to objects that do not start with a header, or would require a complicated change of the class metadata.
- *Support for dynamic array inlining*: Our algorithm for inlining array fields that are changed at run time requires a pointer to the newly allocated (resized) array. The array is accessed via this pointer until the next run of the garbage collector inlines it again (see Section 7.2 on page 83).

In conclusion, changing the object layout does not fit in the structure of the Java HotSpot VM and would require changes in nearly all subsystems. We claim that such radical changes of the object layout cannot be added to the VM, but would require a different VM design.

3.3.2 Preconditions for a Field

Before loads of a certain field can be replaced by address arithmetic, it must be guaranteed that all inlining parents are correctly colocated with their children, i.e. the inlined field must always point to the location that is also computed by the address arithmetic. We define the following preconditions that a field must satisfy for a safe application of object inlining:

1. The parent and all of its inlining children must be allocated together, and the field store that installs a reference to the children into the parent must occur immediately after the allocation.
2. The field referencing the child must not be modified after the allocation. If the field were overwritten later with a new value, the new object would not be colocated to the parent and an optimized field load would access the old child.

In the Java HotSpot VM, it is not possible to influence the heap address of newly allocated objects. They are placed next to each other in the eden space. It is not possible to allocate an inlining child after or before its parent. Instead, the memory for both objects must be allocated at once. If a combined allocation is not possible, the field is not inlinable. Likewise, inlining is not possible if the child object is not always allocated together with its parent, i.e. if the field stays at its initial `null` value in some code paths.

Figure 3.5 illustrates the consequences when the inlining preconditions are not satisfied. Assume that the machine code for the optimized field load accesses the field `rgb` of the class `Color` with the field offset 24 relative to the `Polyline` object without further checks. If the field `lineColor` is not initialized as shown in Figure 3.5 a), the `null` Pointer is not detected. No `NullPointerException` is thrown, and an undefined value is loaded from memory.

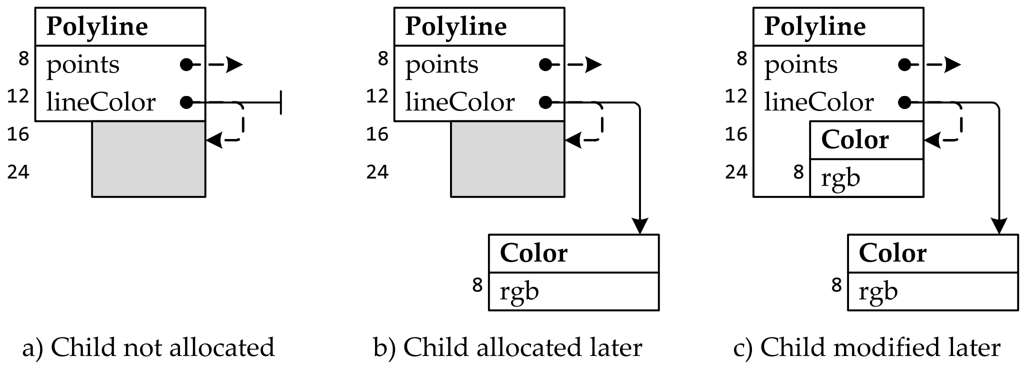


Figure 3.5: Consequences when inlining preconditions are not satisfied

In Figure 3.5 b), the field `rgb` was initialized later with a `Color` object. The new object is not placed correctly next to the `Polyline` object. Again, an undefined value is loaded by an optimized field load. Figure 3.5 c) shows an example where the second precondition is violated and the field `lineColor` is changed later. The optimized field load still accesses the old `rgb` value because it is not possible to allocate the new `Color` object at the same address as the old `Color` object.

The second precondition is necessary because detecting at run time that an object field has been changed is equally or even more expensive than the normal unoptimized field access. Therefore, object inlining is only beneficial in our example if the field with the offset 24 relative to the `Polyline` object can be accessed without further checks. For array fields, this constraint can be relaxed. It is possible to integrate the check for a changed array field into the array bounds check, which is required by the Java language specification (see Section 7.2 on page 83). The modified bounds check does not need an additional machine instruction in the common case.

3.4 Components for Object Inlining

Figure 3.6 shows the components required for object inlining and their interactions. While this section contains a short description of the components, the subsequent chapters present the detailed algorithms.

- *Method tracking*: The class loader builds the *method table*. It contains information about all methods that allocate objects as well as all methods that modify reference fields.
- *Hot-field detection*: When a method is compiled, *read barriers*, i.e. increments of per-class counters, are inserted for all field loads. If a counter for a field `f` exceeds a certain threshold, `f` is considered to be hot and is entered into a *hot-field table*.
- *Object colocation*: When the garbage collector moves objects, it processes groups of objects that are linked by hot fields so that the parent object and its children are consecutive.
- *Co-allocation*: For every hot field `f` that links a parent object of class `P` to a child object of class `C`, the methods that allocate `P` objects are compiled. If possible, the compiler combines the allocations of `P` and `C` objects to a co-allocation. This ensures that the newly allocated objects are placed next to each other in the correct order. If co-allocation is not possible, then object inlining of `f` fails. This includes situations where `C` is not allocated in all cases, i.e. where `f` can be `null` for some code paths. Co-allocation guarantees the first precondition for object inlining.

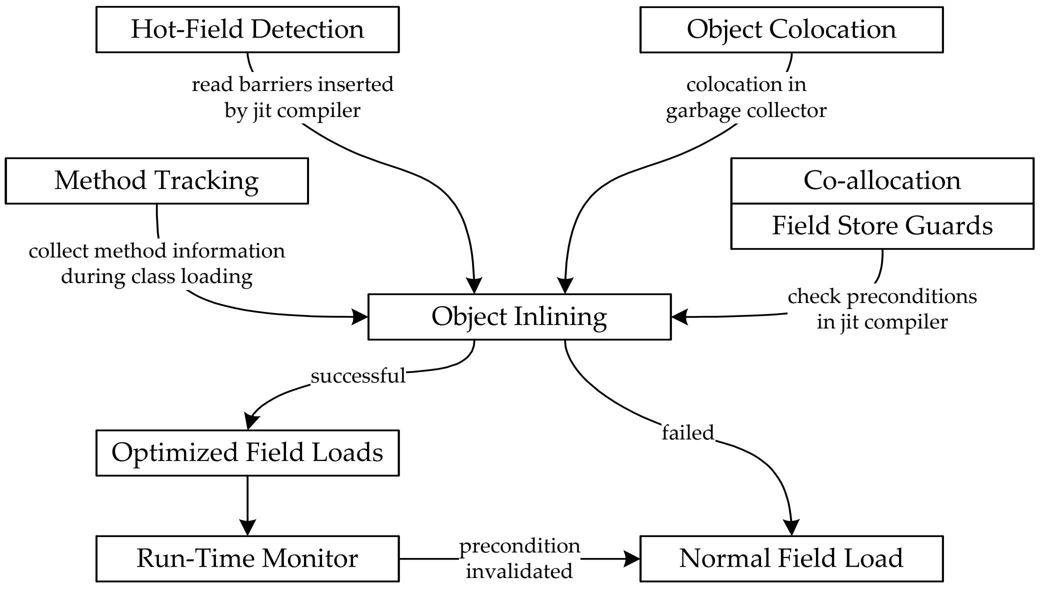


Figure 3.6: Components for object inlining

- *Guards for field stores*: For every hot field `f`, the methods that modify `f` are compiled. The compiler inserts guards in front of the field stores. When a guard is executed later, object inlining of `f` fails and methods with optimized field loads are deoptimized. This guarantees the second precondition for object inlining. For array fields that are allowed to change, the field store guard marks the old array as invalid so that it is no longer accessed.
- *Optimized field loads*: When the two preconditions for a field are satisfied, loads of the field are optimized, i.e. the memory access is removed. In some cases, also array bounds checks and dynamic type checks can be optimized.
- *Run-time monitor*: It is possible that object inlining fails after optimized field loads were emitted, e.g. when a class is loaded later that invalidates a precondition. The run-time monitor detects these cases. All methods with optimized field loads are deoptimized, and execution continues with normal field loads.

If a precondition for a hot field cannot be guaranteed, object inlining is not possible and the field loads must be preserved. However, it is still possible to colocate the objects during garbage collection to improve the cache behavior. Therefore, object colocation in the garbage collector processes all hot fields regardless of their object inlining state.

The subsequent sections explain these components in more detail. We use the example classes defined in Section 3.1. The object fields `lineColor` and `points` of the class `Polyline` and the array field `elementData` of the class `ArrayList` are first detected as hot and then inlined. Each field runs through the optimization phases shown in Figure 3.7.

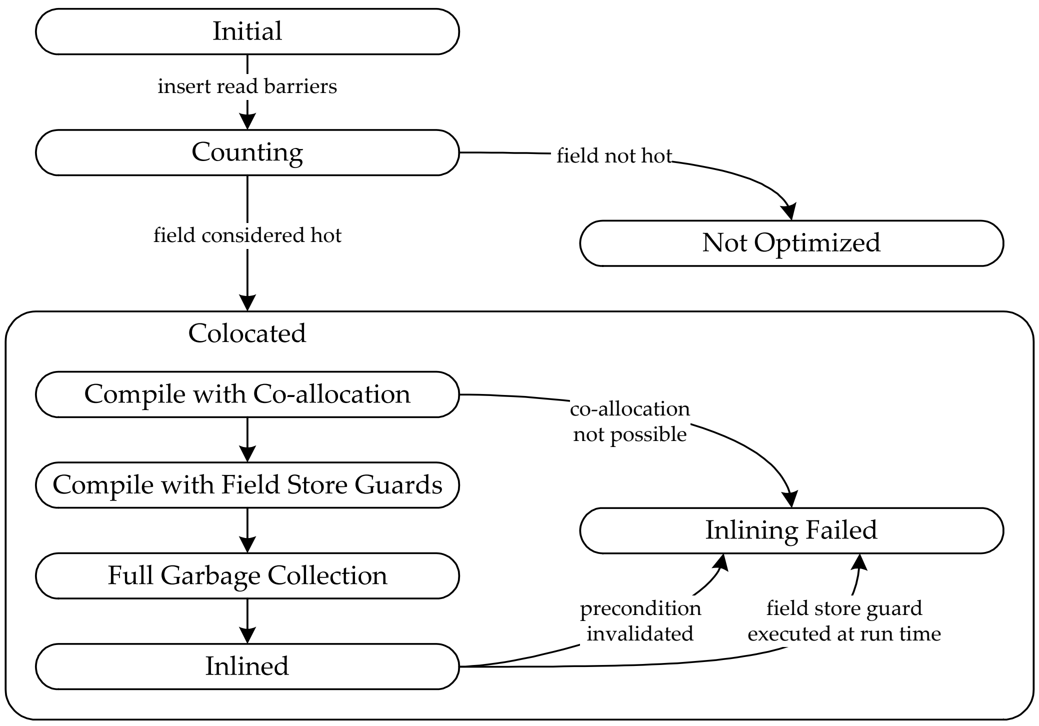


Figure 3.7: Optimization phases for a field

3.4.1 Method Tracking

The class loader maintains a so-called method table. It maps class names to methods that allocate objects of this class, as well as field names to methods that modify this field. Table 3.1 shows the method table for the classes of our example. Classes that do not have reference fields, such as `Color` and `Point`, are not inserted into the table because information about them is not needed for our algorithms.

Key (class name or field name)	**Value** (list of methods)
class `at.ssw.Polyline`	`at.ssw.Test.allocate()`
field `at.ssw.Polyline lineColor`	`at.ssw.Polyline.<init>()`
field `at.ssw.Polyline points`	`at.ssw.Polyline.<init>()`
class `java.util.ArrayList`	`at.ssw.Polyline.<init>()`, `java.io.DeleteOnExitHook.run()`, …
field `java.util.ArrayList elementData`	`java.util.ArrayList.<init>()`, `java.util.ArrayList.add()`

Table 3.1: Mapping of classes and fields to method lists

Information from the table is used in subsequent steps to identify affected methods. Methods that allocate objects must be compiled with co-allocation, and methods that store a field must be compiled with field store guards to guarantee the preconditions for object inlining. Methods contained in both entry kinds like the constructor

`Polyline.<init>()` contain both co-allocations and field store guards. The collection class `ArrayList` is instantiated by several methods of the Java class library during system startup. When the field `elementData` of the `ArrayList` is inlined, all of these methods must be compiled with co-allocation.

3.4.2 Hot-Field Detection

At first, methods are executed by the interpreter. If the invocation counter of a method reaches a certain threshold, the method is scheduled for compilation. Upon compilation of the method `Polyline.getLineColor()`, the compiler emits a read barrier that increments a counter for the field `lineColor`, and the state of this field goes from *initial* to *counting*.

Figure 3.8 shows the HIR for the method. When code is generated for the field load instruction `a2`, the counter increment is emitted. No read barrier is needed for the field load `i3` because it loads a field of a scalar type, which is not of interest for object inlining.

```
bci__tid__instruction_____________________________________
1    a2   a1._12 at.ssw.Polyline.lineColor
4    i3   a2._8 at.ssw.Color.rgb
7    i4   ireturn i3
```

Figure 3.8: HIR of method `Polyline.getLineColor()`

When a field counter reaches a certain threshold, the field is recorded in the hot-field tables, and the state of the field changes from *counting* to *colocated*. If the threshold is not reached in a certain time frame, the field is considered unimportant and the state transitions to *not optimized*. In both cases, the read barriers are removed by recompiling all methods that contain read barriers for the field.

3.4.3 Object Colocation

The garbage collector uses the hot-field tables for object colocation. If a field is accessed frequently, the parent and child objects that are connected by the field are likely to be accessed in quick succession. To improve the cache behavior, it is beneficial to colocate these objects even if object inlining is not possible. When the objects are colocated, they are probably in the same cache line so that accessing the parent also brings the child into the cache.

We modified the stop-and-copy algorithm for the young generation to copy groups of objects instead of individual objects. This ensures that a parent object is copied together with all its child objects. If the parent and the children are not consecutive, they are moved together and the grouping is established. The object groups are also promoted as a whole to the old generation if they survive a certain number of copying cycles.

The mark-and-compact algorithm for the old generation preserves the object order during collection. During compaction, objects are moved towards the beginning of the heap, but their order remains unchanged. Therefore, the optimized order of the promoted objects is retained.

3.4.4 Co-allocation of Objects

Object colocation during garbage collection ensures that a parent object and its child objects are consecutive after the first garbage collection run following their allocation. For object inlining, however, it is a precondition that the objects are already consecutive immediately after their allocation. Therefore, *co-allocation* combines the allocations for a group of objects, and object colocation ensures that the groups are not separated during garbage collection. Our co-allocation is integrated into the client compiler. Therefore, all methods that allocate parent objects must be compiled. The list of methods is retrieved from the method table. In our example, the method `Test.allocate()` must be compiled for the inlining of the fields `lineColor` and `points`, and the method `Polyline.<init>()` must be compiled for the inlining of the field `elementData`.

Figure 3.9 shows the HIR for the method `Test.allocate()`. The instructions refer to the state after method inlining. All constructors were inlined, so there are no more method calls. All instructions except `a2` represent bytecodes of inlined methods.

```
bci_____tid__instruction________________________________________
0       a2   new at.ssw.Polyline
4.5     a8   new java.util.ArrayList
4.9.5   i11  10
4.9.7   a30  new java.lang.Object[i11]
4.9.10  a31  a8._16 := a30 java.util.ArrayList.elementData
4.12    a34  a2._8 := a8 at.ssw.Polyline.points
4.16    a35  new at.ssw.Color
4.23    a42  a2._12 := a35 at.ssw.Polyline.lineColor
// Do something with a2
```

Figure 3.9: HIR fragment of method `Test.allocate()`

The allocations of `Polyline`, `Color`, `ArrayList`, and `Object[]` end up in the same method, as well as the field stores that install the inlining children into the parent objects. A single co-allocation instruction replaces all these instructions. It allocates only one chunk of memory large enough for all objects and then installs the object headers and field pointers appropriately.

Object inlining for a field requires that all methods that allocate objects of the parent class are compiled with co-allocation. If the co-allocation in one method fails, the field is not optimized and the analysis stops. The compiler reports this information as feedback data to the object inlining system. This avoids a data flow analysis during object inlining.

3.4.5 Guards for Field Stores

The second precondition for object inlining specifies that an object field must not be modified after co-allocation. Therefore, we compile all methods that modify the inlined field and instrument the field store to revoke object inlining before the field value is changed at run time. Field stores that are already part of a co-allocation are ignored.

A static check at compile time would not be sufficient because field stores for inlined fields are allowed as long as they are not executed. The static check would inhibit object inlining e.g. for all fields that are assigned inside a constructor. Even though constructors are mostly inlined into the allocating method, they also remain as distinct methods and are separately compiled. Because initializing fields in the constructor is a recommended and frequently used code pattern in Java, the static check would lead to nearly no inlinable fields.

In our example, field store guards are necessary for the fields `lineColor` and `points` when the constructor `Polyline.<init>()` is compiled. Figure 3.10 shows the HIR of the constructor. In contrast to Figure 3.9 of the previous section, the `ArrayList` and the `Color` object cannot be co-allocated with the `Polyline` object because the `Polyline` is already passed in as the first method parameter `a1`. Co-allocation is only performed for the `ArrayList` object and its child `Object[]` array, connected by the field store `a29`.

```
bci___tid__instruction_____________________________________
5     a6   new java.util.ArrayList
9.5   i9   10
9.7   a28  new java.lang.Object[i9]
9.10  a29  a6._16 := a28 java.util.ArrayList.elementData
12    a32  a1._8 := a6 at.ssw.PolyLine.points
16    a33  new at.ssw.Color
23    a40  a1._12 := a33 at.ssw.PolyLine.lineColor
26    v41  return
```

Figure 3.10: HIR of constructor `Polyline.<init>()`

Therefore, the two field store instructions `a32` and `a40` have to be guarded. A call into the VM is emitted in front of the machine code that performs the field stores. The VM method revokes object inlining for the fields `points` and `lineColor`. It is, however, unlikely that this ever happens. The constructor `Polyline.<init>()` was inlined in the method `Test.allocate()`, which is the only method that allocates `Polyline` objects in the bytecodes. Therefore, the constructor itself is only executed if the application allocates a `Polyline` object using reflection or the Java Native Interface.

The field `elementData` of the class `ArrayList` is modified by two methods: `ArrayList.<init>()` and `ArrayList.add()`. Both methods are compiled with field store guards. Because the field `elementData` is an array field and thus allowed to change, the guards have different semantics. They do not revoke object inlining, but mark the old array as inaccessible before the field is overwritten with the pointer to the new array. The optimized array load checks this mark so that the old array is no longer accessed. The guard inserted into `ArrayList.add()` is likely to be executed several times because this method increases the capacity of the `ArrayList`.

3.4.6 Transition to Object Inlining

After all methods that allocate parent objects or store the inlined field are successfully compiled, the two preconditions are satisfied for all objects that will be allocated in the future. However, the heap can still contain objects that were allocated before and that are not colocated yet. Such objects are colocated by the garbage collector.

Therefore, it is necessary to wait for a full garbage collection that processes all generations. In this run of the mark-and-compact algorithm, parent objects are treated specially: when it is necessary to colocate objects, the order of objects is changed in the old generation. Before this, optimized field loads are not possible. The optimizations described in the next section are delayed until the full collection has completed.

3.4.7 Optimized Field Loads

The optimization of field loads is again performed by the just-in-time compiler. Previously compiled methods that load an inlined field are recompiled to apply the optimization. There are two possibilities to optimize field loads:

- *Load folding*: If a field of the child object is accessed, i.e. if the loaded object is used only for another field access, the two field accesses can be merged into a single access with a larger offset. This eliminates one field load.
- *Address computation*: If the address of the child object is required, the inline offset is added to the address of the parent object. This replaces a field load with an arithmetic instruction.

Figure 3.11 b) shows the optimized HIR of the method `Polyline.getLineColor()` where load folding is applied. In contrast to unoptimized HIR, the field load `i3` accesses the field `rgb` relative to the `Polyline` object, which is passed as the method parameter `a1`. The size of the `Polyline` object (16 bytes) is added to the field offset 8 of the field `rgb`, so the overall offset is 24. This offset is visualized in Figure 3.4 a). The field load `a2` of the field `lineColor` is no longer necessary and thus eliminated.

```
tid  instruction
a2   a1._12 at.ssw.Polyline.lineColor
i3   a2._8 at.ssw.Color.rgb
i4   ireturn i3
```

a) Unoptimized HIR

```
tid  instruction
i3   (a1+16)._8 at.ssw.Color.rgb
i4   ireturn i3
```

b) Optimized HIR

Figure 3.11: HIR of method `Polyline.getLineColor()`

3.4.8 Run-Time Monitoring

The preconditions for object inlining can only be guaranteed for the currently loaded classes. Dynamic class loading can introduce new methods that allocate parent objects in such a way that co-allocation is not possible. In this case, object inlining was too optimistic and must be undone by deoptimizing all methods that contain an optimized load of the affected field. Fortunately, this happens rarely. Because the hot-field detection and the necessary compilations that guarantee the preconditions take some time, most applications have already reached a stable execution state when a field is inlined and it is not likely that new classes are loaded afterwards.

When examining the preconditions, we only consider methods that allocate objects or store fields using normal bytecodes. However, Java offers several other and more dynamic ways for this purpose. Objects can be allocated and fields can be modified using *reflection* or the *Java Native Interface* (JNI). New objects are also allocated when an object is cloned using `Object.clone()`.

Because these cases are difficult to handle and rather rare, we are conservative. We disable object inlining for fields that are stored using reflection or the JNI. Similarly, we disable it for all fields of classes that are instantiated dynamically. The subsystems for reflection, JNI, and object cloning are instrumented so that additional code is invoked. This code checks our preconditions and triggers deoptimization if an access affects an inlined field.

Chapter 4

Hot-Field Detection

This chapter presents the data structures and code patterns used to detect hot fields at run time. Read barriers increment per-field and per-class counters, which are checked regularly. Fields whose counters exceed a certain threshold are considered hot and are added to the hot-field tables. Read barriers that are no longer necessary are removed by recompiling methods so that they do not contain counter increments anymore.

Feedback-directed optimizations in virtual machines require live profiling data collected at run time. For our object inlining, we need information about frequently accessed fields. Only these *hot fields* are optimized. We use read barriers inserted by the just-in-time compiler to increment field access counters on a per-field and per-class basis.

The just-in-time compiler has full information about fields. The instruction for a field access in the HIR does not only contain the offset of the accessed field, but also the class that declares the field (the class of the parent object) and the declared type of the field (the class of the child object). Using this information, a unique counter can be created for each field.

Adding read barriers to the interpreter would be far more expensive. The interpreter cannot attach additional information to a specific bytecode, so the address of the field counter would have to be computed anew for each field access. Therefore, we do not count the field accesses executed by the interpreter. This does not have a significant impact on the precision of the measurements as the number of such accesses is comparatively low. The most active methods are compiled so that fields that are only accessed in interpreted code are unimportant for the overall performance. It is a positive side effect of our approach that no counters are allocated for such fields.

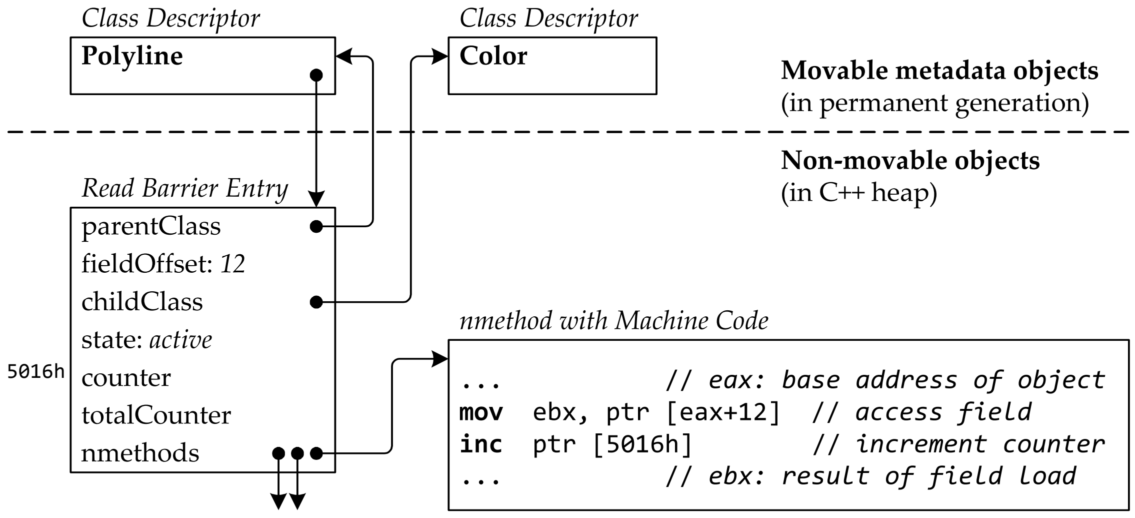
The compiler eliminates a field load if the value of the field is a compile-time constant or if the load is redundant. Our analysis takes such compiler optimizations into account and does not emit read barriers for these loads. Therefore, the resulting counter values can be lower than a naive counting using an instrumented interpreter, but they better reflect the actual behavior of the application.

We count only field loads but no field stores for several reasons. At first, object colocation and object inlining optimize only field loads, so there is no benefit for a field that is frequently stored but rarely loaded. Secondly, a large number of stores can be considered as an argument both in favor and against optimizing a field: If the large number of field stores originates from a large number of objects in which this field is stored once (but then never changed), the field should be optimized. In contrast, if there is a small number of objects in which the field is stored again and again, this indicates a frequently changing data structure for which object inlining is difficult or even impossible.

4.1 Read Barriers

Read barriers allow dynamic measurements of an application's memory access behavior. A read barrier is a piece of machine code that is emitted together with the code that performs the actual load of a field. In our case, the read barrier is a single machine instruction that increments a counter after the field load. The counter is located in a *read barrier entry* that also stores additional information such as the parent class, the child class, and the field offset. It is registered in the class descriptor (see Section 2.2.2 on page 13) of the class that declares the field. When the same field is accessed multiple times, the same entry and therefore the same counter is used.

Figure 4.1 shows the read barrier entry for the field `lineColor` of the example, which is defined in the class `Polyline` and has the declared type `Color`. While the class descriptors are located in the garbage collected heap and can be moved by the garbage collector, the read barrier entries are normal C++ objects with a fixed address. Therefore, the address of the field `counter` is statically known and can be used directly in the machine code. Assume that the field `counter` in our example is located at the address `5016h`.



```
...            // eax: base address of object
mov  ebx, ptr [eax+12]  // access field
inc  ptr [5016h]        // increment counter
...            // ebx: result of field load
```

Figure 4.1: Data structures and machine code for a read barrier

4.1.1 Machine Code Pattern

The right hand side of Figure 4.1 shows the machine code pattern for a read barrier. This machine code is created e.g. for the HIR instruction `a2` of the method `Polyline.getLineColor()` that was presented in Figure 3.8 on page 36. The first machine instruction loads the field of the object whose address is already in the register `eax` and stores the result into the register `ebx`. Then, the increment instruction modifies the counter located at the fixed address `5016h`. The IA-32 instruction set allows instructions to operate on memory operands [Intel07], so it is not necessary to load the counter value into a register. Only a single instruction is necessary for the increment.

The counter increment is not executed atomically by the processor. If two threads load, increment, and store the same counter at the same time, one increment is lost. An atomic machine instruction would ensure that threads do not access the memory concurrently. However, the execution of atomic instructions is an order of magnitude slower and would increase the overhead disproportionally. The small imprecision of the non-atomic counters can be neglected in practice.

Accesses of array elements are counted similarly to fields. The increment instruction is emitted after the array bounds check and the move instruction for the array load. The only difference between object fields and array elements is that array elements lack a field offset. Therefore, we use the marker value -1 in the read barrier entries. A single counter is used for all elements of an array. The parent class of such an entry is the array class (e.g. `Object[]`), the child class is the element type of the array (e.g. `Object`). Information about frequently accessed array elements is used only for object colocation because array elements cannot be inlined by our approach. For both object fields and array elements, we only count loads of references and ignore loads of scalar values such as `int` fields.

4.1.2 Processing of Field Counters

The field counters of all read barrier entries are checked in regular intervals. If a counter exceeds a certain threshold, the field is considered hot and recorded in the hot-field table of the parent class. If the counter does not cross the threshold in several successive measurement intervals, the field is considered unimportant and ignored in all further optimization steps. In most cases, we use the time between two garbage collections as the measurement interval, i.e. the counters are checked at a safepoint before garbage collection. This eliminates the need for explicit locking because all threads are stopped at this time. Newly detected hot fields are immediately optimized by the subsequent garbage collection. Only if the timeframe between two garbage collections is too long, we check the counters using timer interrupts.

Algorithm 4.1 outlines the processing of the field counters. We want to detect fields that are accessed frequently during the current measurement interval, and to filter out the large number of fields that are accessed infrequently. As a heuristic, a field is considered hot if it accounts for more than 5% of all field loads within one interval, i.e. if the counter field of a read barrier entry divided by the sum of all counter fields is above 0.05. This value was determined empirically.

```
ProcessCounters
    sumActive = sum of rb.counter for all active read barrier entries rb

    for each read barrier entry rb do
        if rb.state == active and rb.counter / sumActive > 0.05 then
            add entry(rb.fieldOffset, rb.childClass) to hot-field table of rb.parentClass
            rb.state = added
        end if

        if rb.state != deoptimized and rb.totalCounter / rb.counter > 8 then
            for each nmethod nm in rb.nmethods do
                deoptimize nm        // discard machine code with counter increments
            end for
            rb.state = deoptimized
        end if

        rb.totalCounter += rb.counter
        rb.counter = 0
    end for
```

Algorithm 4.1: Detection of hot fields based on access counters

The heuristic fills the tables iteratively. When processing the counters for the first time, fields with an exceptionally high access frequency are added to the hot-field tables. Their read barrier counters are then deactivated and ignored when computing the percentages in succeeding runs of the algorithm, so the next fields with still a high access frequency are added. This is repeated until a stable state is reached where most fields have similar access frequencies, i.e. no single one is above 5%.

Reaching the stable state usually requires only a few measurement intervals. We use a heuristic to detect unimportant fields by estimating how many intervals passed since the counting has started. We divide the total access count of the field, i.e. the sum of all previous intervals, by the access count of the current interval. If this value is above 8, the field is unimportant. Using this quotient is better than counting intervals. If the access count is decreasing over time, e.g. because the field is accessed only in initialization code, the count of the current interval is low and the field is unimportant. If the access count is increasing over time, the count of the current interval is high and the quotient low, so the field remains counted and can cross the hot-threshold in the next interval. If the access count remains approximately the same over a sequence of intervals without the field becoming hot, it is considered unimportant after 8 intervals.

Incrementing a counter for each field load involves run-time overhead. Therefore, it is necessary to remove read barriers as soon as they are no longer needed, i.e. when it is known that a field is either hot or unimportant. This is done by recompiling all methods that increment the counter of the read barrier entry. The list of these methods is maintained in the read barrier entry. The state of a read barrier entry is used to decide whether a counter increment is necessary for a field. We distinguish three states:

1. *Active*: Field accesses are actively counted because there is no decision for the field yet. When new methods are compiled, read barriers are emitted.
2. *Added*: The field was identified as hot and added to the hot-field table. It is no longer necessary to count field accesses, so no read barriers are emitted for newly compiled methods. However, old methods still increment the counter.
3. *Deoptimized*: Methods that increment the field counter have been scheduled for recompilation and no new read barriers are emitted. The counter will stop being incremented soon. The read barrier entry only serves as a marker that a final decision for this field is available.

Distinguishing between the states *added* and *deoptimized* delays the recompilation of methods that access hot fields. Even if a field is detected as hot in the first measurement interval, the methods with the read barriers are deoptimized approximately after 8 intervals. This reduces the number of methods that need to be recompiled. When object inlining is possible for a hot field, the methods are recompiled anyway to perform the optimized field load. Because it takes some time until inlining succeeds, we delay the recompilation for the read barriers so that both transformations can be applied in one recompilation.

4.2 Hot-Field Tables

The hot-field tables are a VM-global data structure with a separate table for every class with hot fields. This table is registered in the class descriptor and contains a pointer back to the descriptor. Figure 4.2 shows the tables for the example classes. The table of a class stores a list of entries for its hot fields. Each entry holds the offset (*off*) of the field as well as the field's declared type (*class*), which points to another class descriptor.

The table entries are processed by the garbage collector and the object inlining system according to their order in the hot-field tables, so the entries should be sorted by decreasing importance. This is automatically accomplished by the iterative processing algorithm for the field counters that detects hot fields in the order of their importance. In Figure 4.2, it is assumed that the field `lineColor` of the class `Polyline` with the offset 12 is accessed more often than the field `points` with the offset 8.

Classes without frequently accessed reference fields do not have a hot-field table. This includes all classes that have only scalar fields, e.g. the class `Color` in our example. For array classes such as `Object[]`, the marker value -1 is stored as the field offset. A hot-field table does not contain entries for fields declared in a superclass or a subclass. Instead, these classes have their own hot-field tables. In our example, the table entry of the `Object[]` array class points to the `Object` class. `Object` does not have a hot-field table because only some subclasses like `Polyline` have children, not `Object` itself.

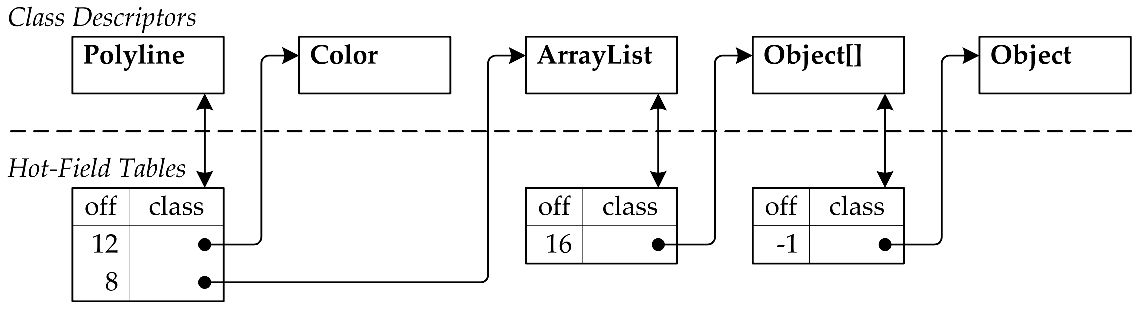


Figure 4.2: Hot-field tables for example classes

Indirect children are only implicitly visible: the class of a child entry also has its own hot-field table. In our example, the `ArrayList` is a direct child of a `Polyline`, while the `Object[]` array is an indirect child. The information about indirect children and children of sub- and superclasses is important for object colocation and object inlining, but including this information in the hot-field tables would complicate building and maintaining the tables.

4.2.1 Graph Representation

The hot-field tables can be visualized as a directed graph. The nodes represent the class descriptors, and the edges represent the hot fields. Figure 4.3 shows the hot-field graph for our example. In addition to the hot-field edges, the class inheritance is modeled. Even this simple example contains a cycle in the graph: the `Object[]` array, which is an indirect child of a `Polyline` object, could contain a `Polyline` object as an array element. For the detection of cycles, an edge that reaches a superclass like `Object` must be treated as multiple edges to each subclass. Cycles are not allowed for object inlining because they would lead to object groups with an unbound number of elements.

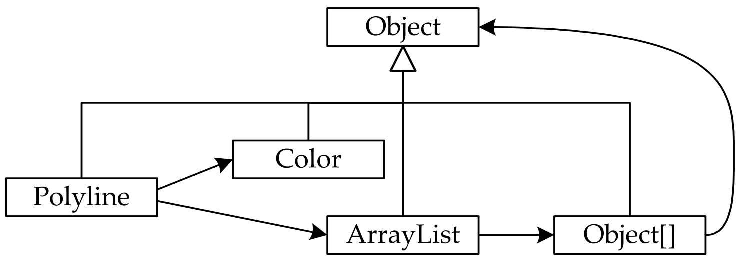


Figure 4.3: Hot-field graph for example classes

For the example, it is guaranteed that concrete object graphs are acyclic because the `ArrayList` of a `Polyline` contains just `Point` objects. However, this information is not available in our implementation because it would require a global data flow analysis. Instead, our algorithms correctly handle and break such cycles. The class `Point` does not show up in the graph. It is neither a parent because it does not have reference fields, nor a child because no field has the declared type `Point`.

Chapter 5

Object Colocation

This chapter presents the algorithms for object colocation, which are integrated into the garbage collection algorithms of the young and the old generation. Before garbage collection, the hot-field graph is converted to colocation tables in order to allow a fast access of colocation information. After assigning a new location to a parent object, all children immediately get consecutive addresses. If a child would be processed before its parent, the processing is delayed.

Object colocation is an optimization that groups heap objects together and sorts them so that their order in memory matches their access order in the program. Our implementation uses the information in the hot-field tables to adapt the copying order of objects in the garbage collector. Object colocation has two goals:

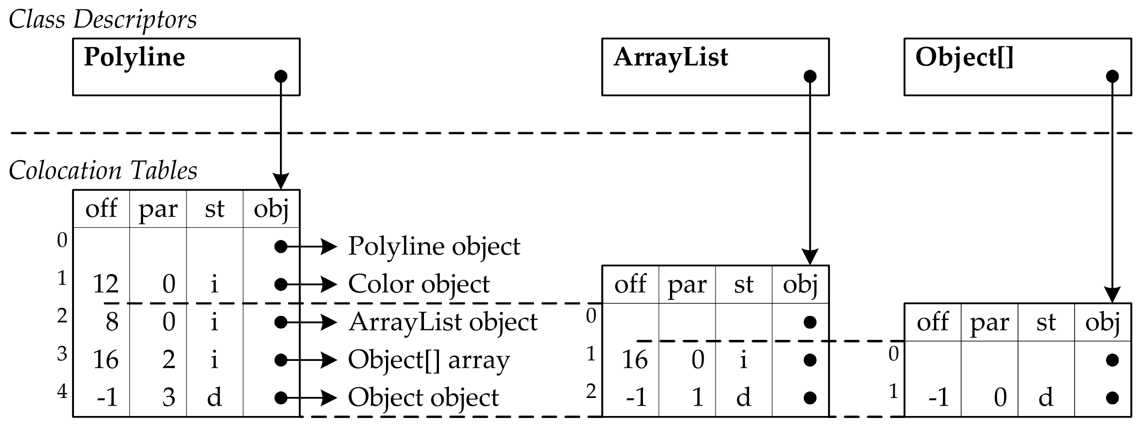
1. *Improve the cache behavior*: If objects that are accessed together are placed next to each other on the heap, the spatial locality is improved. It is more likely that objects either end up in the same cache line or can be optimized by automatic hardware memory prefetching [Hegde07]. This is a statistical optimization. A small ratio of unoptimized objects for a certain class does not impact the performance significantly.
2. *Guarantee preconditions for object inlining*: Memory loads can be replaced by address arithmetic only if the colocation of a parent with its children is guaranteed for all instances of the parent class. If a single parent object cannot be colocated with its children, the optimized access for all objects of this class must be reverted.

To achieve both goals, we distinguish between fields that *should* be colocated to improve the cache behavior and fields that *must* be colocated for object inlining. Such inlined fields are preferred if not all hot fields can be optimized.

The time spent in the garbage collector is critical for the overall performance of a Java application. Object colocation must check for each live object whether another object should be colocated to it. This additional time during garbage collection must be outweighed by the improved cache performance. Therefore, our object colocation algorithm uses an efficient data structure with low per-object costs. Before garbage collection, the hot-field tables are converted to *colocation tables*.

5.1 Colocation Tables

The hot-field tables introduced in Section 4.2 on page 45 are easy to maintain because they store only direct children of a class. However, it is expensive to detect all direct and indirect children that should be colocated to a particular parent object. To limit the overhead during garbage collection, an additional *colocation table* is created from the hot-field table for each class. Figure 5.1 shows the colocation tables for our example classes. They are created from the hot-field tables presented in Figure 4.2 on page 46.



	off	par	st	obj
0				●
1	12	0	i	●
2	8	0	i	●
3	16	2	i	●
4	-1	3	d	●

	off	par	st	obj
0				●
1	16	0	i	●
2	-1	1	d	●

	off	par	st	obj
0				●
1	-1	0	d	●

Figure 5.1: Example of colocation tables used during garbage collection

Each table contains a flat list of all fields that should be colocated for a given class. It is created once before garbage collection, and the *obj* column is filled multiple times during garbage collection. The first entry stores the parent object. All other entries are direct or indirect children of this object. The columns contain the following information:

- Field offset (*off*): The offset of the field whose value is stored in this entry, or -1 as a marker for array elements.
- Parent entry (*par*): The index of this object's immediate parent in the same table. It is 0 for direct children and greater than 0 for indirect children of the parent object for which the table is filled.
- State (*st*): The inlining state of this field, i.e. the information whether colocation should or must be performed. The possible states are listed in Section 5.1.1.
- Object (*obj*): The actual object that is referenced by this field. It is filled anew for each parent object processed by the garbage collector.

In our example, all information required for the colocation of direct and indirect children of a `Polyline` object is contained in the colocation table for `Polyline`. The entries with the indices 1 and 2 denote fields that reference direct children of the `Polyline` object with the index 0. The entries 3 and 4, i.e. the `Object[]` array and another `Object`, denote indirect children. They are direct children of the entries 2 and 3, respectively.

When an `ArrayList` object is not colocated to a `Polyline` object, the colocation table of `ArrayList` is needed. This table is smaller because it contains only the objects that are colocatable to an `ArrayList` object. Similarly, there is a colocation table for the array class `Object[]`. The smaller tables contain parts of the information of the bigger tables, indicated by the dashed lines in Figure 5.1.

Arrays can reference an arbitrary number of child objects. All array elements are usually accessed with similar frequencies. As a pragmatic solution, we colocate only the object referenced by the first non-`null` array element. This removes the necessity for a special support of array elements in the colocation tables. The single colocated array element can be stored in the field *obj* of the table entry.

5.1.1 Creation of Colocation Tables before Garbage Collection

The colocation tables are created before garbage collection using information from the hot-field tables and are discarded afterwards. This simplifies the maintenance because the colocation tables need not be updated when the hot-field tables are changed. Conceptually, the algorithm processes the class nodes of the hot-field graph. For each class, all reachable nodes are added to its colocation table. The graph is modified in a preprocessing step before the tables are created. One of the following edge states is assigned to each edge of the graph:

- *Inlining* (marked with "i"): The colocation must be guaranteed because the field is inlined and field loads are replaced by address arithmetic. This is the initial state of all hot fields that are inlinable according to the analyses presented in Chapter 6 on page 57.
- *Colocation* (marked with "c"): Colocation should be performed to improve the cache behavior. This is the initial state of all non-inlinable fields listed in the hot-field table.
- *Dynamic* (marked with "d"): Colocation should be performed. However, the class of the object referenced by this field can have its own inlining children. In such a case, these children have a higher priority for colocation, so the edge is dynamically disabled for such an object. If the class of the referenced object has no children, colocation is performed. This state is used e.g. for fields of the declared type `Object`.
- *Disabled* (marked with "x"): The edge was removed from the graph e.g. to break a cycle. The field is not added to the colocation table, so this state is used only while processing the graph and never occurs in an entry of a colocation table.

Figure 5.2 shows some situations where the original hot-field graph must be modified before the colocation tables can be created. Edges are disabled if the graph is cyclic like in Figure 5.2 a) because cycles in the flat colocation tables would lead to infinitely large tables. Additionally, edges are disabled if the nesting level of indirect children is too high like in Figure 5.2 b), i.e. when a sequence of colocated objects would involve more than three levels. This limits the maximum size of a colocation table.

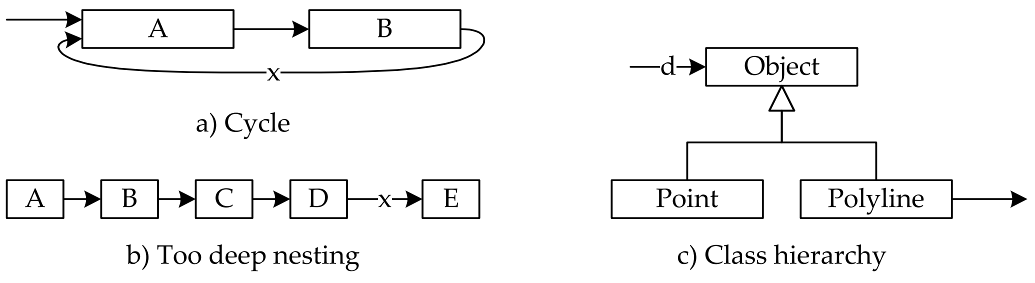


Figure 5.2: Assigning edge states to the hot-field graph

Dynamic edges are required for the optimization of generic data structures that use for example fields of the declared type `Object` or `Object[]` arrays. It is beneficial to colocate an object referenced by such a field or array if it is small and stores only few scalar values. However, it would not be wise to colocate the referenced object if it has colocated reference fields itself, because in this case it would not be possible to optimize these fields as well.

Figure 5.2 c) shows an extract of the graph for the example classes. Initially, the `Object` class is reached by a colocation edge from the `Object[]` class (see Figure 4.3 on page 46). In our example, the `Object[]` array references a `Point` object that should be colocated. This information is however not statically known. Some other `Object[]` arrays on the heap might reference a `Polyline` object instead. If a `Polyline` object were colocated to an `Object[]` array, it would not be possible to colocate also the fields of the `Polyline` object at the same time because the colocation table of `Object[]` contains only an entry for `Object` but no entries for the children of `Polyline`.

Marking the edge to `Object` as dynamic solves the problem. The child object is only colocated if its class does not have its own colocation table. For example, `Point` objects are colocated to `Object[]` arrays, while `Polyline` objects are not. This makes it possible to colocate children of a `Polyline` when such objects are processed by the garbage collector.

5.1.2 Filling the Colocation Tables with Objects

The colocation tables of all classes are created once before garbage collection. All columns but obj are initialized during construction. Algorithm 5.1 shows how the obj column of a table is filled with the children of a parent object during garbage collection. If no colocation table is registered for the class of parent, an empty table is returned. Otherwise, parent is stored in the first entry and then the algorithm iterates over its children. Because the entry c for a child object is always located after its immediate parent, c.par has already been processed before c and its obj column is initialized.

```
GETCHILDREN(parent)
    tab = colocation table for class of parent
    if tab not found then
        return empty table
    end if

    tab[0].obj = parent                   // initialize first entry, which holds the parent
    for i = 1 to tab.length - 1 do        // iterate over all entries except the first
        c = tab[i]                        // get the entry of the current child
        if c.par.obj == null then
            c.obj = null                  // correct handling of fields that are null
        else
            if c.off == -1 then           // -1 is marker value for array elements
                childObj = c.par.obj.firstElement      // access first non-null array element
            else
                childObj = c.par.obj.fieldAt(c.off)    // access field at the specified offset
            end if

            if c.state == dynamic and class of childObj has colocation table then
                c.obj = null              // support for dynamic edges
            else if childObj occurs in tab[0..i-1].obj then
                c.obj = null              // same object is not allowed in table twice
            else
                c.obj = childObj          // common case: store childObj in table
            end if
        end if
    end for

    return tab
```

Algorithm 5.1: Filling a colocation table during garbage collection

If the parent object c.par.obj is `null`, then the object of the current entry c.obj is also set to `null`. This is necessary for the correct handling of fields that are `null`. Such entries are ignored later when the table is used. However, a large number of `null` values slows down the garbage collector. To avoid this, we count `null` values for every hot field and disable a field if it has too many `null` values.

The value of an object field is loaded by accessing the memory at offset c.off relative to the parent object c.par.obj. For array elements, which are identified by the marker value -1 in the colocation table, the first non-`null` array element is searched and used as the child object. This is better than using always the first element because data structures like hash tables populate arrays randomly and not starting with the first element.

Two additional checks are performed before the child object is stored in the colocation entry. If a check fails, the entry is invalidated by setting the object to `null`:

1. Dynamic edges are handled as described in the previous section. If the class of the child object has a colocation table itself, this entry is invalidated. The child's colocation table is filled later when GETCHILDREN is called for this child.
2. One colocation table must not contain the same object twice. The later steps of the garbage collection algorithm would also copy the object twice. Therefore, the entry is invalidated if the field obj of a previous entry already contains the current child object. The duplication check is implemented efficiently using a lightweight hash table.

The colocation tables are used by the stop-and-copy and the mark-and-compact collection algorithm. All objects listed in the table after a call to GETCHILDREN are placed next to each other on the heap. The following sections describe how this is done.

5.2 Stop-and-Copy Algorithm

The young generation of the heap is collected using a stop-and-copy algorithm (see Section 2.3.1 and Algorithm 2.1 on page 15). We extended this algorithm to process groups of objects instead of individual objects. Algorithm 5.2 shows the modified algorithm for COPYOBJECT. The parts added for object colocation are marked gray. Before a parent object is copied, the colocation table is filled using the algorithm GETCHILDREN. All child objects in the table are then copied together with their parent object. The memory for the object group is allocated at once, so the size of the whole object group must be computed before the actual memory allocation. The handling of child objects that are `null` or that are already in the old generation is omitted from the algorithm; such children are ignored.

The root pointers are processed in arbitrary order. If both the parent object and a child object are referenced by a root pointer, it can happen that the child object is copied before the parent. An object must not be copied twice, so the two objects cannot be colocated in this garbage collection run.

To avoid that children are copied before their parents, objects that were once detected to be colocation children are tagged with a dedicated bit in the mark word of the object header (see Section 2.2.1 on page 12), referred to as isColocationChild in the algorithm. The copying of tagged objects is delayed until the parent object is processed to ensure that colocation succeeds. All children keep the tag for their entire lifetime, i.e. the tag is persistent between two garbage collections because it is stored in the object header. Therefore, objects are guaranteed to stay colocated even when new root pointers to children are introduced.

```
COPYOBJECT(obj)
    if obj.forwardPtr is set then
        return obj.forwardPtr                   // prevent copying an object twice
    else if obj.isColocationChild then
        return fixupMarker                      // delay copying; a fixup is done later
    end if

    tab = GETCHILDREN(obj)                      // get children of obj (or empty table)
    allocSize = obj.size                        // computation of total allocation size
    for i = 1 to tab.length - 1 do
        tab[i].obj.isColocationChild = true     // tag object as colocation child
        if tab[i].obj.forwardPtr not set then   // must not have forward pointer to colocate
            allocSize += tab[i].obj.size
        end if
    end for

    newObj = ALLOCATE(obj, allocSize)           // allocate memory for parent and children

    memmove(obj, newObj, obj.size)              // copy and forward parent object
    obj.forwardPtr = newObj
    offset = obj.size
    for i = 1 to tab.length - 1 do              // copy and forward children
        if tab[i].obj.forwardPtr not set then
            memmove(tab[i].obj, newObj + offset, tab[i].obj.size)
            tab[i].obj.forwardPtr = newObj + offset
            offset += tab[i].obj.size
        end if
    end for

    return newObj
```

Algorithm 5.2: Modified stop-and-copy algorithm for object colocation

If the copying of a child object is delayed, the references to the child require a later fixup. COPYOBJECT returns a fixup marker so that the reference is added to a list. When the scan of the to-space is completed, these references are updated to the forward pointer of the child object that was set during the colocated copying. In rare cases, it can happen that the parent object has died, but the child object is still alive because another object holds a reference to it. Similarly, a field update of the parent object can install a new child object and leave the old one without a parent. Such objects are still uncopied at the start of the fixup phase, so they are copied before the fixup and their colocation bit is cleared.

Figure 5.3 shows an example for object colocation in the stop-and-copy algorithm. An object group consists of a parent object `P` and a child object `C`. The arrows above the objects are root pointers, the arrows below the objects are field references. Assume that the root pointers are processed from left to right. Objects that are tagged as colocation children in the mark word are shown with a `c` in the upper left corner. After garbage collection, all child objects are tagged.

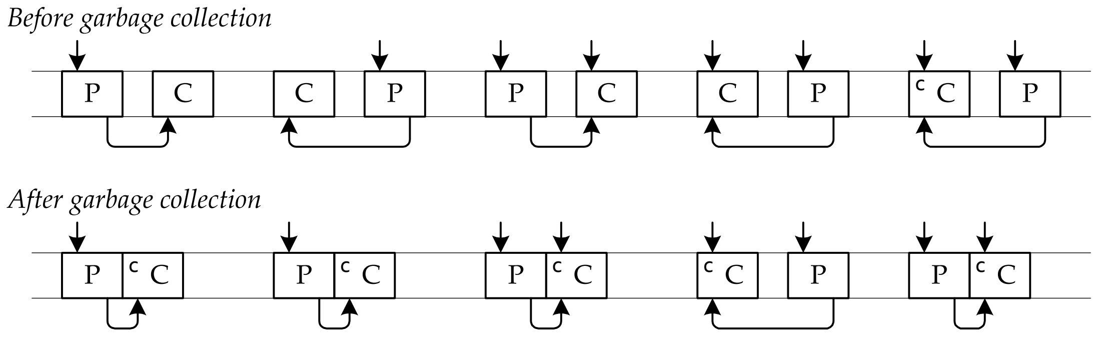


Figure 5.3: Example for object colocation in the stop-and-copy algorithm

As long as the child object is not accessible via a root pointer, the optimized order can be established regardless of the object order before garbage collection. If a root pointer to the child object is processed before the root pointer to the parent object and the child object is not yet tagged, the objects are still unoptimized after garbage collection, but now the child is tagged. The next garbage collection cycle establishes the optimized order. The copying of the tagged child is delayed until the parent object is processed.

5.3 Mark-and-Compact Algorithm

The mark-and-compact algorithm is used to collect the entire heap. This is necessary when the old generation is full and therefore no space would be available for the promotion of objects during a collection of the young generation. Object colocation in the stop-and-copy algorithm also affects the old generation because colocated objects are promoted together, i.e. they end up colocated in the old generation. The basic mark-and-compact algorithm does not change the order of objects and therefore preserves this optimized order.

A collection of the young generation can only colocate a group of objects if all members are still in the young generation. If a child has already been promoted, it cannot be colocated. However, measurements in an early phase of the project showed that this is a rare case. Object colocation for the mark-and-compact algorithm did not improve the cache behavior [Wimmer06].

Therefore, modifications of the mark-and-compact algorithm are only necessary for the second goal of our object colocation algorithm: guaranteeing the preconditions for object inlining. Only inlining edges of the hot-field graph are processed, all other edges are disabled to improve performance.

Algorithm 5.3 shows the modifications for the basic mark-and-compact algorithm, which was presented in Section 2.3.2 and Algorithm 2.2 on page 16. The parts added for object colocation are marked gray. Because the entire heap is traversed several times, all children can be detected and colocated without consulting the isColocationChild bit in the object header.

```
MarkAndPush(obj)                          // modifications for phase 1
    if not obj.marked then
        obj.marked = true
        markStack.push(obj)
        tab = GetChildren(obj)            // get children of obj (or empty table)
        for i = 1 to tab.length - 1 do
            tab[i].obj.forwardPtr = CHILD // use forward pointer to mark children
        end for
    end if

ComputeNewAddresses                       // modifications for phase 2
    newObj = space.begin
    for each marked object obj do
        if obj.forwardPtr != CHILD then   // children are processed with their parents
            obj.forwadPtr = newObj
            newObj += obj.size
            tab = GetChildren(obj)        // get children of obj (or empty table)
            for i = 1 to tab.length - 1 do // forward all children
                tab[i].obj.forwardPtr = newObj
                newObj += tab[i].obj.size
            end for
        end if
    end for

MoveObjects                               // modifications for phase 4
    for each marked object obj do
        newObj = obj.forwardPtr
        if newObj > obj then              // object moves towards end of heap
            newObj = new scratch buffer   // copy object to temporary buffer
        end if
        memmove(obj, newObj, obj.size)
    end for
    for each scratch buffer obj do        // copy temporary buffers to their destination
        newObj = obj.forwardPtr
        memmove(obj, newObj, obj.size)
    end for
```

Algorithm 5.3: Modified mark-and-compact algorithm for object colocation

Object colocation is integrated into the four phases of the mark-and-compact algorithm:

1. *Mark live objects*: Children are detected using the colocation table for the class of the marked object. We use the forward pointer, which is normally unused in this phase of the algorithm, to tag child objects. After the marking phase, it is guaranteed that all children are tagged and that a parent object exists for each tagged child.

2. *Compute new addresses*: In this phase, the algorithm GETCHILDREN must be called again for each parent object because there is only one colocation table per class. All children of a parent get assigned consecutive addresses. This may change the order of objects on the heap. With the help of the tags that are set in the first phase, the processing of a child is delayed if it is reached before its parent. Such a child object never gets a new address before its parent. As a result, child objects can move towards the end of the heap. However, this only happens if the child was promoted to the old generation before its parent was promoted.
3. *Adjust pointers*: No changes are necessary in this phase because the forward pointers of all parent and child objects are set correctly.
4. *Move objects*: Since objects can also move towards the end of the heap now, this phase must take precautions to avoid overwriting yet uncopied objects. Objects that move towards the end are first copied into a scratch area and then copied back to their final location after all other objects have been processed. However, this is only necessary for a small number of objects. Each object is rescued at most once. At the next collection, the object order is already correct and no reordering and rescuing is necessary.

Chapter 6

Object Inlining

This chapter presents the main algorithms for performing feedback-directed object inlining. To guarantee the preconditions, object allocations must be modified such that a parent object and all its children are allocated together. Additionally, field stores of inlined fields must be guarded by a runtime call to detect field modifications. These algorithms are integrated into the just-in-time compiler, i.e. all relevant methods need to be compiled before memory loads can be replaced by address arithmetic. To handle the case where the parent object's class has subclasses, it is necessary to reverse the order of objects in the object group and place the parent object, whose size is not fixed if there are subclasses, at the end of the group.

Object inlining is an optimization that replaces memory loads by address arithmetic when a field of an object is known to point to another object that is colocated with the first one. In Section 3.3.2 on page 32, we define two preconditions for optimizing a field. We use the just-in-time compiler as well as run-time monitoring to guarantee these preconditions. Methods that are relevant for a field, i.e. methods that allocate parent objects and methods that store the field, are compiled with additional compiler phases that transform the methods. The compiler reports feedback data to the object inlining system. If the transformation fails for one method, the field is considered not inlinable.

The compiler combines the analyses whether inlining is possible with the necessary transformations of methods to guarantee the preconditions. This way, we avoid a data flow analysis. The compiler operates on a per-method basis, so only intraprocedural information can be collected and method calls must be handled conservatively. This constraint is lowered by method inlining that replaces a method call by a copy of the called method. Normally, only small methods are inlined. However, larger methods are inlined if object inlining can profit from the enlarged analysis scope.

6.1 Method Tracking

Two kinds of methods are relevant to guarantee the preconditions for inlining a field: methods that allocate objects of the class that contains the field, and methods that modify the field. To ensure that the child object that is referenced by a field is always colocated with the parent object that contains the field, methods of the first kind are compiled with co-allocation (see Section 6.3), and methods of the second kind are compiled with guards for field stores (see Section 6.4). To decide which methods must be compiled, the bytecodes of all methods are analyzed. Methods that allocate objects or modify fields are inserted into the *method table* during class loading and linking.

The method table is a mapping of class names and field names to method lists. It is organized as a hash table. The key of an entry consists of three parts: the kind of the entry, i.e. *class* or *field*, the class name, and the field name. The field name is unused for *class* entries. The value of an entry is a list of methods, i.e. references to the internal metadata objects maintained for methods. Table 3.1 on page 35 shows the method table for our example classes.

Class names are fully qualified, i.e. they contain the package names. Unfortunately, it is not possible to use the internal class metadata objects of the VM as keys instead of the names. Linking is not done during class loading but only at the first execution of the referencing bytecodes, so the metadata is not available yet when the method table is filled. This introduces a small imprecision because we cannot distinguish two classes with the same package name and the same class name. This is allowed in Java when the two classes are loaded by different class loaders. We handle this case conservatively: either both or none of the two classes are optimized. However, such name conflicts are unlikely in practice because the package name usually contains the vendor of a class.

Only classes and fields that could be relevant for object inlining are tracked. It is not necessary to track a class if this class and its superclasses do not have reference fields, i.e. if it cannot be an inlining parent. When such a class is loaded, a marker is inserted into the table. It prevents the tracking of methods that allocate objects of this class. Additionally, fields of scalar types can be ignored. The type of a field can be checked during the bytecode analysis because it is stored as static information in the constant pool of a class.

6.1.1 Bytecode Analysis

Algorithm 6.1 shows the bytecode analysis that fills the method table. The `new` and `putfield` bytecodes [Lindholm99] of a method are relevant for object inlining. These bytecodes have one static operand, which is an index into the constant pool of the method's class. For `new` bytecodes, the constant pool entry is the name of the allocated class. For `putfield` bytecodes, the entry is a symbolic reference to a field, which consists of the name and type of the field as well as the name of the class declaring the field. For both bytecodes, a new entry is added to the method table.

```
ANALYZEMETHOD(method)
    for each bytecode bc of method do
        if bc.opcode == new then
            klassName = method.klass.constantPool.klassNameAt(bc.index)
            if methodTable.isTracked(klass, klassName) then
                methodTable.add(klass, klassName, method)
            end if
        else if bc.opcode == putfield then
            fieldType = method.klass.constantPool.fieldTypeAt(bc.index)
            if fieldType == object or fieldType == array then
                klassName = method.klass.constantPool.klassNameAt(bc.index)
                fieldName = method.klass.constantPool.fieldNameAt(bc.index)
                methodTable.add(field, klassName, fieldName, method)
            end if
        end if
    end for
```

Algorithm 6.1: Bytecode analysis to fill method table

A method must be processed by ANALYZEMETHOD before it is executed for the first time, so the time span starts when the method's class is loaded and ends at the first invocation. In general, it is best to analyze a method as late as possible. Many methods, especially those from library classes, are loaded but never executed. Analyzing such methods would hinder object inlining: the transformations for object inlining can fail when they are compiled to guarantee a precondition. Additionally, compiling such methods increases the run-time overhead.

Usually, the first invocation of a method starts in the interpreter. Only if the method affects an inlined field, it is necessary to compile it before its first invocation. Therefore, we initiate the analysis at link time shortly before the method is invoked for the first time and compilation is still possible before execution starts.

6.1.2 Class Hierarchies

The class information of a `putfield` bytecode contains the static type of the variable used for the field access. This type can be a subclass of the class in which the field is declared. Therefore, the class hierarchy must be checked when accessing information from the method table. For example, Figure 6.1 introduces a subclass `Polygon` that extends the class `Polyline`. Both classes have a method that stores the field `lineColor` defined in the class `Polyline`. The information of the `putfield` bytecode in the method `Polyline.foo()` contains the class name `Polyline`, whereas the `putfield` bytecode in the method `Polygon.bar()` contains the class name `Polygon` because the static type of `this` is `Polygon`. The information that both stores access the same field is not available until linking, i.e. when the `putfield` bytecodes are executed for the first time.

```
class Polyline {
  Color lineColor;

  void foo() {
    lineColor = ...
  }
}
```

```
class Polygon extends Polyline {
  void bar() {
    lineColor = ...
  }
}
```

Figure 6.1: Field access across class hierarchy

Table 6.1 shows the resulting method table. The field `lineColor` appears twice: with the class name `Polyline` as well as with the class name `Polygon`. It would be complicated to coalesce these entries and merge the method lists. Instead, we access the entries of a class and all its subclasses when searching for methods that store a field. In our example, both entries and therefore both methods are returned when searching for methods that store the field `lineColor` of the class `Polyline`.

Key (class name or field name)	**Value** (list of methods)
field `at.ssw.Polyline lineColor`	`at.ssw.Polyline.foo()`
field `at.ssw.Polygon lineColor`	`at.ssw.Polygon.bar()`

Table 6.1: Method table for class hierarchy

This introduces another minor imprecision of the method table. The class `Polygon` could define a field with the name `lineColor` itself. The two methods would access different fields, but the method table would still return both methods. The method `Polygon.bar()` would be unnecessarily compiled when the field `Polyline.lineColor` is to be inlined. This is correct because it is safe to compile any method, but it introduces an additional overhead. In practice, such cases are rare and considered bad programming style.

Static fields are not tracked in the method table. They can be distinguished from instance fields because they are accessed using the bytecode `putstatic` instead of `putfield`. Static fields are stored at the end of the class descriptor, i.e. the metadata object of a class (see Figure 2.3 on page 14). It is not possible to colocate an object referenced by a static field with the class descriptor because the class descriptor is located in the permanent generation, while the referenced object is located in the young or old generation. Therefore, we cannot optimize static fields and do not need to track them.

6.2 Inline Requests

Guaranteeing the preconditions for a field requires the compilation of several methods and a full garbage collection. Therefore, the analysis must be asynchronous to the execution of the application. To track the state of the inlining process, an *inline request* object is maintained for each field that is a candidate for inlining. It is registered in the hot-field tables and stores the following information:

- *Unique ID*: To allow references to an inline request from compiled code and to simplify tracing, a unique number identifies each request.
- *State*: The inline request can be in the state *compiling* (methods must be compiled), *garbage collection* (only the full garbage collection is pending), and *successful* (optimized field loads are allowed). The state *failed* marks fields for which the compilation of a method could not guarantee a precondition and optimized field loads are impossible.
- *Methods to compile*: A list of methods that must be compiled to guarantee the preconditions. When such a method is compiled, it is removed from the list.
- *Methods to recompile*: Methods that were compiled due to an invocation counter overflow before the inlining process of a field was initiated must be recompiled to insert co-allocations and guarded field stores. The old machine code is discarded when the new one is available.

6.3 Co-allocation of Objects

Object colocation in the garbage collector (see Chapter 5 on page 47) ensures that a group of objects, i.e. a parent object, its children, and further indirect children, are next to each other in memory after garbage collection. However, to safely eliminate field loads of pointers connecting these objects, it is necessary that the objects are colocated before they are processed by the garbage collector the first time. This is guaranteed by co-allocation of objects, which combines the allocation of a parent and all its children.

6.3.1 Modification of the Just-in-Time Compiler

The interpreter processes one bytecode at a time. Only information about this bytecode is available. However, co-allocation combines several bytecodes: the allocation of the parent object, the allocations of the child objects, and the field stores that install pointers to the children into the parent. The interpreter is not designed for such sophisticated analyses, therefore we limit our implementation to the just-in-time compiler. All methods that allocate parent objects are compiled and checked for the above pattern. These methods are listed in the method table described in Section 6.1.

Algorithm 6.2 shows the insertion of co-allocation instructions into the HIR of a method. The algorithm is split into two phases. First, we search for field store instructions that are suitable for co-allocation and build trees of allocation instructions. Then, we create one co-allocation instruction for each tree and add all allocation and field store instructions of the tree to the co-allocation instruction.

```
COALLOCATION
    parents = { }
    children = { }

    for each field store fs do
        if fs.obj is allocation and fs.val is allocation and    // allocated in same method
                fs.obj not reachable from sf.val and            // no cycle
                fs.val not in children and                      // only one parent
                checkMemoryFlow(fs, fs.obj, fs.val) then        // field is never null
            fs.obj.addChild(fs)                                 // add to allocation tree
            parents.add(fs.obj)
            children.add(fs.val)
        end if
    end for

    for each allocation instruction alloc in parents do
        if alloc not in children then               // alloc is root of an allocation tree
            create and insert new co-allocation instruction coalloc
            FILL(alloc, null, coalloc)
        end if
    end for

FILL(alloc, fs, coalloc)
    coalloc.add(alloc, fs, coalloc.totalSize)       // add allocation, field store and offset
    coalloc.totalSize += alloc.size

    sort children of alloc according to hot-field table
    for each child c of alloc do                    // children are field store instructions
        FILL(c.val, c, coalloc)                     // recursive call
    end for
```

Algorithm 6.2: Co-allocation in just-in-time compiler

To detect all allocation instructions that are applicable for co-allocation, we iterate over the field store instructions of a method because field stores connect the allocations. A field store instruction has a reference to two other instructions as its parameters: the object that is modified (obj) and the new value of the field (val). Co-allocation is possible if all of the following criteria are satisfied:

1. The object and the value are both allocation instructions, i.e. both refer to objects created within the same compiled method. It is not possible to co-allocate a new object with a pre-existing object that is e.g. passed to the method as an argument.
2. Co-allocation of a parent and its indirect children is possible, but the resulting structure must be acyclic. Field stores that install a pointer to a parent object into one of its children are ignored.
3. A child object can only have one parent. If an allocation is the value operand of two field stores, one of the stores must be ignored. In other words, the resulting structure of parents, children, and indirect children must be a tree.

4. The allocations and the field store may be in different basic blocks, but they must be executed together in all possible code paths. For example, it is not allowed that a child object is only allocated in an if-branch and the field remains `null` or gets another value assigned in the else-branch. Additionally, it must be guaranteed that the field is not loaded before the field store because we cannot handle cases when an inlined field is still `null`.

If a field store and its connected allocations satisfy all criteria, they are added to the tree-based data structure for co-allocation. After all field stores have been processed, the allocations are clustered into one or more trees. For each tree, one co-allocation instruction is created and inserted into the HIR. The allocations of the parent and all its children are added recursively to a flat list. Each entry of the list contains the allocation instruction, the field store instruction, and the offset of the object relative to the start of the object group. Additionally, the total size of the object group is computed.

The back end of the compiler uses this information to generate LIR operations for the co-allocation. The operation that allocates a single chunk of memory for all objects at once is followed by several operations that install the appropriate object headers. The field stores are also performed at this time because the child objects must not remain unreferenced. Otherwise, they could be reclaimed by the garbage collector because there is no reference to them before the field store is executed.

Object colocation for the garbage collection of the young generation uses the isColocationChild bit in the mark word of the object header to tag child objects (see Section 5.2 on page 52). Otherwise, a child could be copied before its parent and colocation would fail. This bit is also set by the co-allocation instruction for child objects. As a result, the group of objects established by the co-allocation is never separated by the garbage collector.

The algorithm for co-allocation is only loosely coupled with the rest of the object inlining system. Co-allocation can be performed for all objects independently of the hot-field tables. Information from the hot-field tables is used only for the correct sorting of the children. The order of the children for co-allocation must match the order in the hot-field tables, otherwise the objects would be reordered during garbage collection and optimized field loads for newly allocated objects would be incorrect.

6.3.2 Example

This section illustrates the co-allocation performed by Algorithm 6.2 for the method `Test.allocate()` from the example. Figure 6.2 replicates Figure 3.9 on page 37 and shows the HIR of this method. The method contains four allocation instructions `a2`, `a8`, `a30`, and `a35`, as well as three field store instructions `a31`, `a34`, and `a42`.

```
bci_____tid__instruction_______________________________________
0       a2   new at.ssw.PolyLine
4.5     a8   new java.util.ArrayList
4.9.5   i11  10
4.9.7   a30  new java.lang.Object[i11]
4.9.10  a31  a8._16 := a30 java.util.ArrayList.elementData
4.12    a34  a2._8 := a8 at.ssw.PolyLine.points
4.16    a35  new at.ssw.Color
4.23    a42  a2._12 := a35 at.ssw.PolyLine.lineColor
// Do something with a2
```

Figure 6.2: HIR fragment of method `Test.allocate()`

The algorithm iterates over the field store instructions. The first field store is `a31`, which stores the value `a30` into a field of the object `a8` at offset 16. Both the object and the value are allocation instructions in the same method. All other criteria for co-allocation are also satisfied, so `a30` is added as a child of `a8`. We also store the field store instruction `a31` as an edge linking `a8` and `a30` because we need information about the field store later on.

The second field store `a34` adds the previously detected parent `a8` as a child of `a2`. This is allowed because the resulting structure is still acyclic. The third field store `a42` adds a second child to `a2`. The resulting structure can be visualized as a tree, as shown in Figure 6.3. The nodes are allocation instructions, the edges are field store instructions.

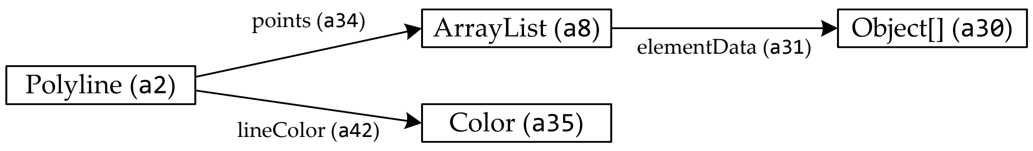


Figure 6.3: Co-allocation tree built for method `Test.allocate()`

In this example, all allocation instructions of the method are in a single tree, so only one co-allocation is constructed in the second step of the algorithm. The root of the tree is instruction `a2`. It is added to the co-allocation instruction during the first invocation of FILL. No field store is recorded for the root, and the offset is 0. Then, the children of `a2` are sorted according to the hot-field table (see Figure 4.2 on page 46). This ensures that the allocation `a35` for the `Color` object is added before the allocation `a8` for the `ArrayList`. These two allocations are added in recursive calls of FILL, together with the field store instructions and the offsets 16 and 32, respectively.

Finally, the allocation `a30` of the `Object[]` array is added with the offset 56. The offset is equal to the sum of the sizes of all previously added objects. Table 6.2 shows the resulting table of the co-allocation instruction. The first three columns are directly stored in the table, the other columns show indirect information available through the allocation instructions (*size* and *allocation class*) or the field store instructions (*field name*).

Allocation	Field Store	Offset	Size	Allocation Class	Field Name
a2	–	0	16	at.ssw.Polyline	–
a35	a42	16	16	at.ssw.Color	lineColor
a8	a34	32	24	java.util.ArrayList	points
a30	a31	56	56	Object[]	elementData
		total size:	112		

Table 6.2: Details of the co-allocation instruction in method `Test.allocate()`

The original allocation and field store instructions are no longer necessary. No LIR operations are created for them. They are subsumed by the co-allocation instruction. Figure 6.4 contains the LIR operations that are emitted for the co-allocation instruction of the example. The right hand side shows the resulting group of objects.

```
// Allocate memory for whole object group
alloc_raw ecx, esi, size:112 -> eax

// Initialize object headers
move obj:at.ssw.Polyline -> esi
move int:1 -> [eax + 0]       // mark word
move esi -> [eax + 4]         // class pointer
move obj:at.ssw.Color -> esi
move int:129 -> [eax + 16]    // mark word
move esi -> [eax + 20]        // class pointer
move obj:java.util.ArrayList -> esi
move int:129 -> [eax + 32]    // mark word
move esi -> [eax + 36]        // class pointer
move obj:Object[] -> esi
move int:129 -> [eax + 56]    // mark word
move esi -> [eax + 60]        // class pointer
move int:10 -> [eax + 64]     // array length

// Compute object addresses and set fields
lea  [eax + 0] -> R51
lea  [eax + 16] -> R52
move R52 -> [eax + 12]    // Polyline.lineColor
lea  [eax + 32] -> R53
move R53 -> [eax + 8]     // Polyline.points
lea  [eax + 56] -> R54
move R54 -> [eax + 48]    // ArrayList.elementData
```

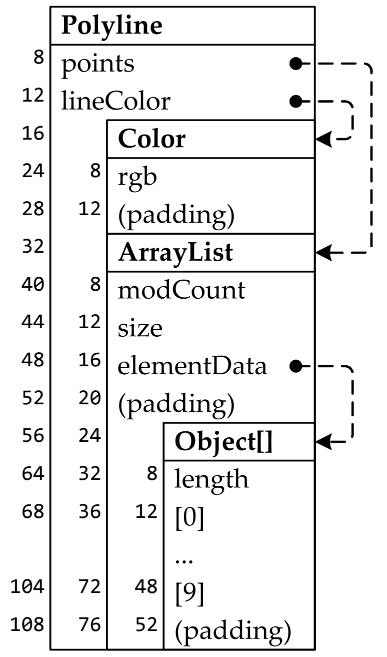


Figure 6.4: LIR for co-allocation in method `Test.allocate()`

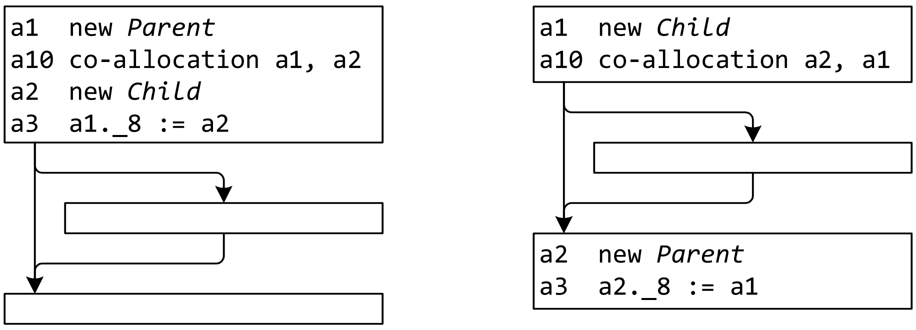
First, one chunk of memory with 112 bytes is allocated on the heap for all four objects. Then, the object headers, i.e. the mark words and the class pointers as well as the array length of the `Object[]` array, are installed. The offsets of the objects relative to the start of the memory chunk are available from the co-allocation instruction. Finally, the addresses of the individual objects are computed using the address arithmetic operation `lea` (*load effective address*). These addresses are installed in the fields and are available in virtual registers for later LIR operations.

The mark word is initialized differently for the parent object and the child objects. The mark value 1 is the default for newly created unlocked objects (see Figure 2.2 on page 12). If the bit 7 is set as well, i.e. if the value 128 is added to the default mark value, the object is tagged as a child object. We use this bit for the tagging in the garbage collection of the young generation. The bit is taken from the hash code bits and thus reduces the range of hash codes.

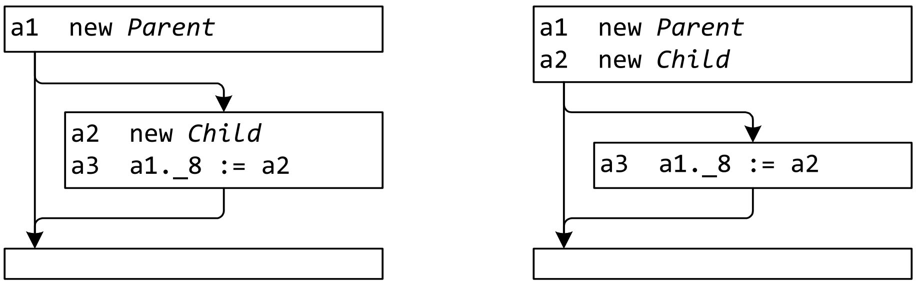
6.3.3 Control Flow and Memory Flow

The call of checkMemoryFlow in Algorithm 6.2 ensures that the allocations of the parent and the child object as well as the field store that connects them are executed in all possible code paths. Restricting co-allocation to allocations performed in the same block would be overly conservative. A single conditional instruction or if-block between two allocations would prohibit co-allocation and therefore also object inlining for the field that connects the objects.

Figure 6.5 illustrates some cases of legal and illegal control flow for co-allocation. In the simplest case, all instructions are in the same block. The co-allocation instruction a10 is inserted after the first allocation because there could be other instructions between a1 and a2 that reference a1, so the object must be available.



a) Legal control flow



b) Illegal control flow

Figure 6.5: Control flow for co-allocation

It is also allowed that the instructions are in different blocks as long as they are executed in all code paths. The if-block in the second example does not inhibit co-allocation. Also, the reversed order of the allocation instructions does not affect the algorithm. The co-allocation instruction `a10` is again inserted after the first allocation, which is the allocation of the child object in this case.

Figure 6.5 b) shows examples where co-allocation is not possible. In both cases, the field is `null` when the if-branch is not taken. In the second example, it is irrelevant that the parent and the child object are allocated in the same block. An optimized field access is not possible when the field could be `null` because the `null` check would be as expensive as the field access and would make the optimization useless.

Figure 6.6 shows an example for a legal and an illegal memory flow. After the field store instruction, the parent object can be used without restrictions as shown in Figure 6.6 a). However, between the allocation and the field store, the field is still `null`. If the parent object is passed as a method parameter like in Figure 6.6 b), the called method could perform an optimized field load, which is not allowed before the field store has been executed.

```
a1  new Parent
a10 co-allocation a1, a2
a2  new Child
a3  a1._8 := a2
v4  invoke foo(a1)
```

a) Legal memory flow

```
a1  new Parent
a2  new Child
v3  invoke foo(a1)
a4  a1._8 := a2
```

b) Illegal memory flow

Figure 6.6: Memory flow for co-allocation

Because we do not use an interprocedural analysis, we handle such cases conservatively. We do not perform co-allocation if the parent object could be accessed between the allocation and the field store. In contrast, there are no restrictions for child objects. They can be accessed freely and also passed to other methods immediately after allocation.

6.4 Guards for Field Stores

The second precondition for object inlining specifies that an inlined field must not be modified after it was assigned for the first time. This is ensured by guarding field stores. All methods that contain a store of the field must be compiled. The list of these methods is available from the method table described in Section 6.1. If the value of a field is changed, the inlining of the field must be revoked because the new child that is installed is not colocated to the parent. The revocation is done by a runtime function that is called before the actual field store. It causes methods that contain an optimized field load of this field to be deoptimized.

It is not possible to replace the run-time check by a compile-time check. In many cases, methods with guarded field stores are compiled even though the compiled code is never executed. Following the usual code pattern in Java, fields are initialized with child objects in the constructor of the parent object. The constructor is invoked immediately after the allocation and usually inlined into the allocating method. Nevertheless, the constructor continues to exist also as a separate method that could be invoked, e.g. via reflection. We compile the constructor with field store guards so that object inlining can be revoked in such a case.

In our example, the fields `lineColor` and `points` of the class `Polyline` are stored once per object in the constructor of the class `Polyline`. The constructor is small and therefore inlined into methods that allocate `Polyline` objects, e.g. into the method `Test.allocate()`. Section 6.3.2 showed that co-allocation is performed for the `Polyline` object and its children when this method is compiled. When the constructor itself is compiled, co-allocation is not possible because the `Polyline` object is passed to the constructor as a method parameter.

Figure 6.7 replicates Figure 3.10 on page 38 and shows the HIR for the constructor `Polyline.<init>()`. The field stores `a40` and `a32` assign non-colocated objects to the fields `lineColor` and `points`. Object inlining of the fields needs to be reverted before the field stores are performed. However, `a28` can be co-allocated with `a6` because both allocations and the field store between them are in the same method.

```
bci___tid__instruction_________________________________
5     a6   new java.util.ArrayList
9.5   i9   10
9.7   a28  new java.lang.Object[i9]
9.10  a29  a6._16 := a28 java.util.ArrayList.elementData
12    a32  a1._8 := a6 at.ssw.PolyLine.points
16    a33  new at.ssw.Color
23    a40  a1._12 := a33 at.ssw.PolyLine.lineColor
26    v41  return
```

Figure 6.7: HIR of constructor `Polyline.<init>()`

Figure 6.8 shows a fragment of the LIR for this method. The method parameter, i.e. the `Polyline` object, is stored in the virtual register `R41`. Operation 50 allocates the `Color` object, and operation 58 stores this object into the field `lineColor` using the offset 12 relative to the `Polyline` object. Immediately before this store, operation 56 calls the runtime function `putfieldGuard` to revoke object inlining. The identification number of the inline request (see Section 6.2) for the field `lineColor` is passed as an argument. Operation 54 stores this number (which is 0 in our case) on the method stack according to the calling conventions.

```
nr__operation______________________________________________
// R41 contains first method parameter (the Polyline object)
48  move obj:at.ssw.Color -> edx
50  alloc_obj edx, ecx, esi, size:16 -> eax
52  move eax -> R50
54  move int:0 -> [esp + 0]
56  call putfieldGuard
58  move R50 -> [R41 + 12]
```

Figure 6.8: LIR fragment for constructor `Polyline.<init>()`

The method `putfieldGuard` revokes object inlining for the specified field. It deoptimizes all methods that contain an optimized load of this field because this is no longer safe after the field store. The machine code of these methods is discarded. If such methods are currently running, the execution is continued in the interpreter (see Section 2.5 on page 23). It is not necessary to modify the heap in `putfieldGuard` because our object inlining preserves field pointers and object headers. Unoptimized field loads work correctly without changing the objects. Furthermore, object colocation is still performed in the garbage collector to improve the cache behavior.

Deoptimization is an expensive operation. Therefore, we use heuristics to limit the number of fields for which object inlining is initiated. For example, if the number of methods that contain stores of a field is high, it is likely that the field is changing at run time, so the optimization process is never started.

6.5 Transition to Object Inlining

Co-allocation and guarded field stores ensure that the parent-child relationship of newly allocated objects is established and retained. Object colocation in the garbage collector preserves the groups of colocated objects. However, the heap can still contain non-colocated objects, for example in the infrequently collected old generation. Such objects are colocated by the next full garbage collection, so it is necessary to delay the subsequent optimization steps.

Normally, the mark-and-compact algorithm used for a full garbage collection does not change the order of objects. It only removes gaps between objects by moving them towards the beginning of the heap. However, the garbage collection that completes the object inlining process for a field must re-order objects to establish the colocation (see Section 5.3 on page 54 for the special handling of object colocation in this algorithm).

Once the inlining process for a field is completed, methods that allocate parent objects or store the field must not be executed in the interpreter. Only the compiled machine code with co-allocation and field store guards guarantees the preconditions. If code without these transformations is still running, e.g. further up in the method call chain, it is necessary to wait until the execution has finished.

For example, a method m that allocates a parent object could invoke a long-running operation before the actual allocation is performed. Assume that object inlining is initiated after execution of m started in the interpreter. All relevant methods are compiled with co-allocation and field store guards (including m), but m is still running in the interpreter. Object inlining cannot succeed while m is interpreted because m will allocate new non-colocated objects.

Therefore, we scan the method stacks of all threads and search for methods that contain relevant allocations or field stores but whose execution started before they were compiled with co-allocation and field store guards. The first full garbage collection where no such methods are running anymore is used to complete the inlining process.

6.6 Optimized Field Loads

When the preconditions for a field are satisfied, loads of the field can be replaced by address arithmetic. This is performed by the just-in-time compiler. It does not pay off to optimize field loads in the interpreter because the overhead of interpreting is much higher than the possible gain of optimized field loads. In contrast to the previous sections, it is also not necessary to compile all methods that load an inlined field. Only frequently executed methods that load the field are optimized. These are methods that were previously compiled because of an invocation counter overflow. Such methods are recompiled to apply object inlining.

There are two possibilities to optimize field loads: *load folding* and *address computation*. Additionally, other compiler optimizations benefit from the information collected during object inlining. It is guaranteed that an inlined field is non-`null` and that it is never changed. Also, the exact type of the object referenced by the field is known, which can be used to eliminate type checks and to convert dynamic binding of method calls to static binding.

6.6.1 Load Folding

In many cases, a field load that yields a child object is immediately followed by another field access that uses the result. Load folding merges the two memory accesses. The resulting access uses a larger offset, which is the memory distance between the parent and the child plus the offset of the second field access. Figure 6.9 shows load folding for the example method `Polyline.getLineColor()`. This method contains two successive field loads. First, the field `Polyline.lineColor` is used to load a `Color` object. Then, the field `Color.rgb` of the loaded object is accessed. The resulting integer value is returned.

```
tid__instruction______________________
a2   a1._12 at.ssw.PolyLine.LineColor
i3   a2._8 at.ssw.Color.rgb
i4   ireturn i3
```

a) Unoptimized HIR

```
tid__instruction____________________
i3   (a1+16)._8 at.ssw.Color.rgb
i4   ireturn i3
```

c) Optimized HIR

```
nr__operation_________________________
10  move [ecx + 12] -> eax
12  move [eax + 8] -> eax
16  return eax
```

b) Unoptimized LIR

```
nr__operation________________________
10  move [ecx + 24] -> eax
14  return eax
```

d) Optimized LIR

Figure 6.9: Load folding in `Polyline.getLineColor()`

The unoptimized HIR contains the two field load instructions. They are converted to two move operations in the LIR with the field offsets 12 and 8. Load folding eliminates the first field load. The address of the `Color` object is never explicitly present. Instead, the field `Color.rgb` is accessed with a larger offset relative to the `Polyline` object. The `Color` object is located immediately after the `Polyline` object, which has a size of 16 bytes. The combined field offset is 16 + 8 = 24. Figure 6.10 visualizes these offsets.

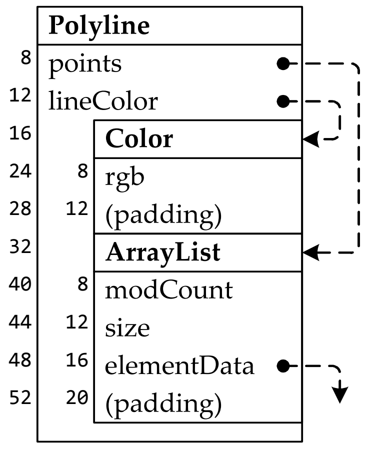


Figure 6.10: Field offsets of inlined objects

Load folding benefits from method inlining. Following the encapsulation principle, many field loads occur in small accessor methods. When these accessor methods are inlined, the field loads get exposed in the HIR of the calling method and load folding is possible. The second field access can be a load or store of any type. Load folding can merge a load of an inlined field and a store into the child object to a single store with a larger offset. Similarly, more than two field accesses can be folded to a single one when inlining children are nested, i.e. when an indirect child of a parent object is accessed.

6.6.2 Address Computation

Load folding can only be applied if the address of the child object is not needed as an explicit value. However, some operations require the address of the child object, e.g. when the child object is passed as a method parameter or when a synchronization operation is performed on it. In this case, the load of the inlined field cannot be eliminated, but the memory access can be replaced by address arithmetic. The distance between the parent and the child in memory is added to the address of the parent.

Figure 6.11 shows the address computation in the HIR and LIR for the example method `Polyline.getPoint()`. The load of the inlined field `Polyline.points` yields an `ArrayList` object. The method `ArrayList.get()` is called on this object, i.e. the `ArrayList` object is used as a parameter in a method call. Assume that the called method is too large to be inlined.

```
tid  instruction
a3   a1._8 at.ssw.Polyline.points
a4   a3.invokeinterface(i2)
        java.util.List.get
a6   areturn a4
```

a) Unoptimized HIR

```
nr  operation
12  move [ecx + 8] -> ecx
18  virtual_call ecx -> eax
26  return eax
```

b) Unoptimized LIR

```
tid  instruction
a3   (a1+32) at.ssw.Polyline.points
a4   a3.invokestatic(i2)
        java.util.ArrayList.get
a6   areturn a4
```

c) Optimized HIR

```
nr  operation
12  lea  [ecx + 32] -> ecx
18  static_call ecx -> eax
26  return eax
```

d) Optimized LIR

Figure 6.11: Address computation in `Polyline.getPoint()`

The unoptimized HIR contains a load of the field `Polyline.points`, which is converted to a LIR `move` operation with a memory access on the left hand side. In the optimized HIR, the field load is replaced by address arithmetic. The offset of the `ArrayList` object relative to the `Polyline` object, which is 32 according to Figure 6.10, is added to the address of the `Polyline` object. In the LIR, the `lea` (*load effective address*) operation provided by the Intel IA-32 architecture is used. It is similar to an addition, but can place the result in a different register than the source operands. In summary, the total number of executed instructions is not reduced, but nevertheless one memory load is eliminated.

6.6.3 Additional Optimizations

The analysis for object inlining increases the amount of static type information for inlined fields. The co-allocations guarantee that an inlined field is initialized with a child of the same type in all parent objects, and the guarded field stores ensure that this field is not changed later on. Therefore, the dynamic type of the field is known, which is more precise than the declared type defined at compile time in the Java bytecodes. The additional information can be used by the just-in-time compiler to apply optimizations.

In our example, the field `points` of the class `Polyline` has the declared type `List`, which is a generic interface of the Java collections library. There are several implementations of the interface, e.g. `ArrayList` or `LinkedList`. When the field is declared using the `List` interface, the implementation class can be changed later on by modifying only one line, i.e. the allocation in the constructor of `Polyline`. However, the interface type `List` complicates compiler optimizations because the just-in-time compiler does not know the actually used implementation class. Therefore, a virtual call of the interface method `List.get()` is necessary in the example of Figure 6.11 a) when a method is invoked on the field `Polyline.points`.

Co-allocation discovers that this field is always initialized with an `ArrayList` object. The compiler can use this information and replace the dynamic binding to the method `List.get()` with a static binding to the method `ArrayList.get()`. This eliminates the overhead of dynamic binding and allows the compiler to inline the method. In the optimized HIR of Figure 6.11 c) the virtual call was replaced by a static call.

The client compiler inlines only statically bound methods. Heavily optimizing compilers like the server compiler can also inline the most frequently invoked virtual method. However, a run-time type check is necessary in the machine code to ensure that the inlined method is executed only for objects of the correct class. The additional type information from object inlining can eliminate such type checks. Explicit type checks in the Java bytecodes like `instanceof` and `checkcast` are eliminated in a similar way.

In addition to having more precise type information, the just-in-time compiler also knows that an inlined field is never `null`. Comparisons of an inlined field with `null` can be eliminated and replaced with unconditional jumps. Other optimizations such as global value numbering and loop invariant code motion benefit from the fact that an inlined field is invariant. Normally, such optimizations must conservatively assume that a field is changed by a method call. In contrast to that, loads of inlined fields can be eliminated or moved without precautions. However, the impact is limited because load folding eliminates the field load anyway, so other optimizations only take effect if address computation is necessary.

6.6.4 Handling Null Checks

Java is a safe language, so the specification demands that all illegal memory accesses are intercepted. For example, a `NullPointerException` must be thrown if a field is accessed via a `null` pointer. Even optimized machine code must adhere to the strict exception semantics and throw the same exceptions in the same order as the interpreter would throw them.

A `NullPointerException` usually indicates an erroneous program, so the compiler optimizes for the case that no exception is thrown. The `null` check is implicitly integrated into a memory access and uses the exception mechanism of the processor. If the memory access dereferences `null`, a hardware exception is thrown. The virtual machine catches the hardware exception and converts it to a `NullPointerException` at the Java level. Therefore, no explicit comparison with `null` is necessary before a field access.

When a memory access is eliminated by object inlining, the implicit `null` check is removed as well, i.e. the case when the variable that holds the parent object is `null` is no longer detected. Inserting an explicit `null` check would not be beneficial because it is as expensive as a memory access.

Address computation is only allowed if no `null` check is necessary. Sophisticated `null` check elimination can remove most checks. Nevertheless, some loads of inlined fields cannot be optimized. Figure 6.12 illustrates the influence of `null` values. The left hand side shows the unmodified example method `Polyline.getPoint()`. The load of the inlined field `points` operates on the `this` pointer, which can never be `null`. Therefore, the memory access can be replaced by address computation.

```
Point getPoint(int i) {
  return this.points.get(i);
}
```

a) No `null` check necessary

```
static Point getPoint(Polyline p, int i) {
  return p.points.get(i);
}
```

b) No optimization because `null` check necessary

Figure 6.12: Influence of `null` checks on address computation

In contrast to that, the field load of Figure 6.12 b) operates on the method parameter `p`, which can be `null`. The address computation would interpret `null` as the address 0 and add 32 to it, which would lead to an address that is neither `null` nor valid. Therefore, the load of the field `points` cannot be optimized, although the same HIR instructions are created for the methods in Figure 6.12 a) and b). The fact that the field `points` is never `null` is irrelevant in this case.

Load folding can handle `null` values correctly in many cases because it is possible to let the folded memory access perform the implicit `null` check. Figure 6.13 shows the same situation as above for the method `Polyline.getLineColor()`. The `this` pointer is never `null` and load folding is possible on the left hand side. On the right hand side, a `null` check is necessary for the method parameter `p`. The check can be done by the memory access that loads the field `rgb` with the offset 24 relative to the `Polyline` object, as shown in Figure 6.9. We ensure that the exception thrown by the implicit `null` check contains the information that the variable `p` was `null` and not the field `lineColor`, which can never be `null` as it is an inlined field.

```
int getLineColor() {
  return this.lineColor.rgb;
}
```

a) No `null` check necessary

```
static int getLineColor(Polyline p) {
  return p.lineColor.rgb;
}
```

b) Folding of `null` check possible

Figure 6.13: Influence of `null` checks on load folding

6.7 Run-Time Monitoring

Java applications can load new classes at run time. This complicates dynamic optimizations because new bytecodes can invalidate preconditions of previous optimizations. Analyzing all classes that are in the classpath and could possibly be loaded is neither reasonable (because it bloats the internal tables with unused classes) nor sufficient (because new classes can be loaded e.g. via a network connection). Therefore, the run-time system must track the preconditions of optimizations and trigger deoptimization if a newly loaded class does not satisfy a precondition.

Our object inlining uses the method table to support dynamic class loading. Before a method is executed for the first time, it is analyzed and inserted into the method table if it allocates new objects or performs field stores. The object inlining system tracks changes of the method table. If a method is added to an entry of an inlined field or of a class containing an inlined field, the method must be compiled with co-allocation or guarded field stores. We compile this method immediately to check whether it satisfies the preconditions. If the transformation fails, e.g. because co-allocation is not possible, the inlining of the affected field must be revoked using deoptimization.

This case is handled in the same way as a failing field store guard (see Section 6.4). Fortunately, it happens rarely. The detection of hot fields using read barriers, the compilation of methods with co-allocation and field store guards, and the full garbage collection require some time, so the application has usually reached a stable state before the first methods are compiled with optimized field loads. It is unlikely that later loaded classes affect already optimized data structures.

Beyond bytecodes, Java offers several ways to allocate objects or to modify fields. Objects can be allocated and fields can be stored using *reflection* or the *Java Native Interface* (JNI). New objects are also allocated when an object is cloned using `Object.clone()`. These techniques are more dynamic than bytecodes, i.e. they can allocate objects of arbitrary classes and modify arbitrary fields. No static information about the affected classes is available.

We handle these dynamic features conservatively because they are rarely used. If a class is instantiated or a field is modified by anything else than bytecodes, we disable object inlining for this class or field. Code that uses the JNI is usually developed in C or C++, and the reflection system as well as `Object.clone()` are part of the Java class library. It is not feasible to analyze them directly. However, all three possibilities use callbacks into the VM to resolve class names and field names to class objects and field offsets. We instrument these VM callbacks so that special code is executed before the callback returns. In this code, we invalidate the class or field in the method table, which excludes it from optimization. If an already inlined field is affected, we revoke inlining by using deoptimization.

6.8 Support for Class Hierarchies

Class hierarchies are a central concept of object-oriented programming. Subclass objects can be used as if they were superclass objects. Figure 6.14 adds two subclasses to our example: `Pattern` is a subclass of `Color`, and `Polygon` is a subclass of `Polyline`. When taking object headers and 8-byte alignment into account, both subclasses are 8 bytes larger than their superclasses, i.e. their size is 24 bytes instead of 16 bytes.

```
class Color {                               class Pattern extends Color {
  int rgb;                                    int style;
}                                             int size;
                                            }
class Polyline {
  List<Point> points;                       class Polygon extends Polyline {
  Color lineColor;                            Pattern fillPattern;

  Polyline() {                                Polygon() {
    points = new ArrayList<Point>();            super();
    lineColor = new Color();                    fillPattern = new Pattern();
  }                                           }
}                                           }
```

Figure 6.14: Java source code for class hierarchy

The class `Pattern` is a subclass of `Color`, which is the type of the inlined field `lineColor`. Subclasses of inlining children do not need special handling during object inlining, and the class hierarchy can safely be ignored. The constructor `Polyline.<init>()` explicitly allocates a `Color` object and not a `Pattern` object. If the field `lineColor` were initialized with a `Pattern` object in some cases, e.g. using an if-statement in the constructor, the field would not be inlinable because co-allocation would be impossible.

6.8.1 Reverse Object Order

The class `Polygon` adds an additional field to its superclass `Polyline`. This affects the offsets of the inlined fields `points` and `lineColor` of the class `Polyline`. The increased size of `Polygon` objects changes the offsets of the child objects that are located after the parent object. Since a variable of type `Polyline` can also refer to a `Polygon` object, the offset for optimized field loads of the fields `points` and `lineColor` are no longer fixed.

Figure 6.15 illustrates this situation. For example, the optimized field load in the method `Polyline.getLineColor()` shown in Figure 6.9 uses the offset 24 relative to the `Polyline` object to load the field `rgb`. If this method were called for a `Polygon` object, it would access the wrong memory position and load the mark word of the `Color` object instead of the value of the field `rgb`.

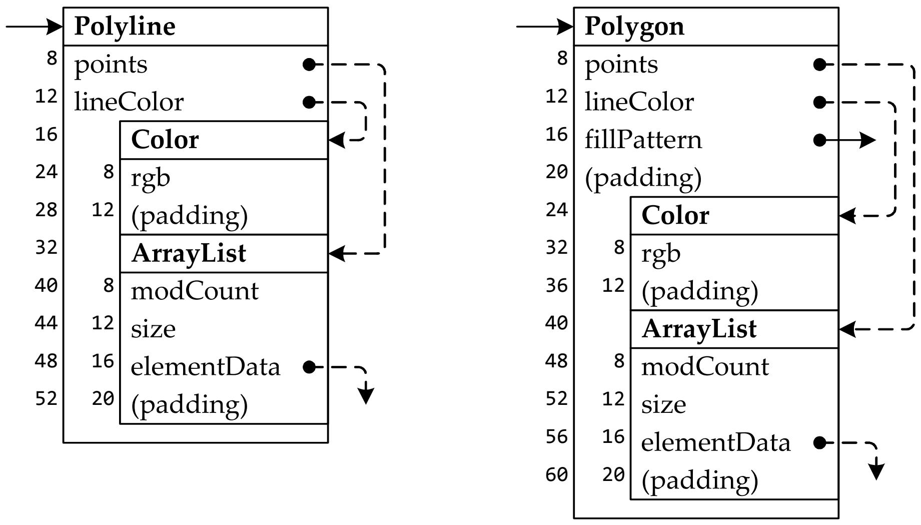


Figure 6.15: Inconsistent inline offsets with class hierarchies

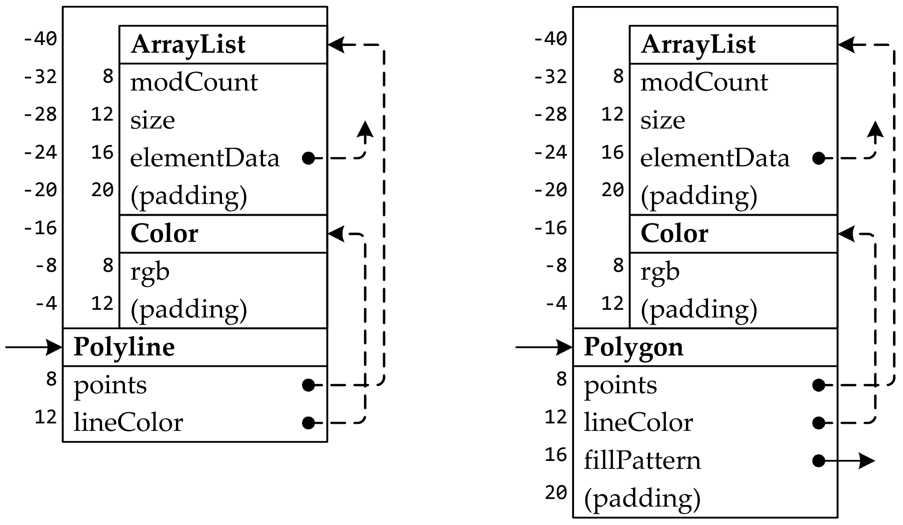


Figure 6.16: Reverse object order to support class hierarchies

We solve this problem by reversing the object order, i.e. we place the child objects in front of the parent object on the heap. With this reverse order, the offset of the children are fixed negative values. Figure 6.16 illustrates the object order and the field offsets that are used to access fields of inlined objects. For example, the optimized field access of the field `rgb` uses the offset -8, which is equal for `Polyline` and `Polygon` objects. The

8-byte alignment of objects is still necessary, i.e. a padding of 4 bytes is inserted between the field `rgb` and the `Polyline` header.

Mixing normal and reverse object order is not reasonable. For example, it would not be possible to combine a parent object optimized using the normal order and another parent object optimized using the reverse order to an inlining hierarchy later on. Therefore, we use the reverse object order in all cases.

6.8.2 Modifications for Reverse Object Order

Reversing the object order requires modifications of some algorithms presented in the previous chapters. However, no architectural changes are necessary because the objects of a group are independent. The object group does not need special header information, so it does not matter that the header of the parent object is now in the middle of the object group.

The object order is relevant for object colocation in the garbage collector (see Algorithm 5.2 on page 53 and Algorithm 5.3 on page 55) as well as for co-allocation in the just-in-time compiler (see Algorithm 6.2). In these algorithms, the loops processing the child objects must be changed such that children are processed before their parent and in reverse order. The entries of the colocation table are iterated in reverse order.

Chapter 7

Array Inlining

This chapter presents the differences between objects and arrays as far as they are relevant for inlining. On the one hand, the size of arrays is not fixed at compile time, which complicates the inlining process. On the other hand, array fields that are changed several times can be inlined because the check for modifications can be integrated into the array bounds check. Inlining array element objects into arrays is not possible without a global data flow analysis.

Arrays play an important role in object-oriented applications. While objects are used to decompose the functionality of a program into well-understandable small parts, arrays are ideal for the implementation of dynamic data structures. Objects of business logic classes use arrays to reference variable-sized lists of related objects. Such dynamic lists of children are encapsulated in collection classes, which are part of almost every class library. Because of the additional layer between the business object and the array, several memory accesses are necessary to load an array element. Array inlining reduces this overhead by folding field loads into the array accesses.

Java integrates array types smoothly into the object class hierarchy. Arrays are considered as objects and inherit from the common base class `Object`. However, there are certain differences between objects and arrays that affect inlining. When arrays are used as inlining children, it must be considered that the size of arrays is not necessarily a compile-time constant. Using arrays as inlining parents is impossible because of the limited type information in Java bytecodes for accessing array elements.

7.1 Arrays as Inlining Children

The preconditions for object inlining ensure that an object field references the same inlining child throughout the whole lifetime of the object. The class of the inlining child and therefore its size is a compile-time constant. The field is not allowed to change because the new child would not be colocated anymore. Changed object fields cannot be detected efficiently when the field is loaded, therefore we use guards to intercept field stores.

The basic algorithm for object inlining can also be applied for the inlining of array fields. The detection of hot fields and the colocation in the garbage collector remain unchanged. However, the actual inlining algorithm must take the differences between objects and arrays into account. On the one hand, the size of an array cannot be determined at compile time in many cases, which complicates array inlining. On the other hand, it is possible to integrate the check whether an array field has been changed into an array access with no additional costs by embedding it into the array bounds check. Allowing array fields to change increases the number of array fields that can be inlined. We distinguish the following three cases:

- *Fixed array inlining*: Inlining of an array field that references only arrays with the same constant length.
- *Variable array inlining*: Inlining of an array field that may reference arrays of different lengths, but is assigned only once.
- *Dynamic array inlining*: Inlining of an array field where the length of the referenced arrays may vary at run time, i.e. a field that is modified several times.

7.1.1 Fixed Array Inlining

If all objects of a parent class point to arrays with the same fixed length, the inlining of array fields can be handled in the same way as the inlining of object fields. Because their length is constant, all referenced arrays have the same size. In addition to eliminating the field access for the array field, other aspects of the array access can be optimized as well. The constant length can be used to simplify the array bounds check, which does not need to load the array length anymore. The constant array length is detected during co-allocation when all methods that allocate the parent objects and the child arrays are compiled.

In the example in Figure 7.1 a), the class `Parent` contains the array field `child` that always references an array of length 2. To inline this field, the method `allocation()` must be compiled with co-allocation. Figure 7.1 b) shows the HIR instructions of the method where the allocation instructions `a1` and `a9` and the field store instruction `a10` are combined to the co-allocation instruction `a14`. The constant `i8` with the value 2 is used for the array length. The just-in-time compiler detects that the array length is constant and informs the object inlining system. For fixed array inlining, the length must be equal for all allocation sites of the class `Parent`. Figure 7.1 c) shows the optimized object layout.

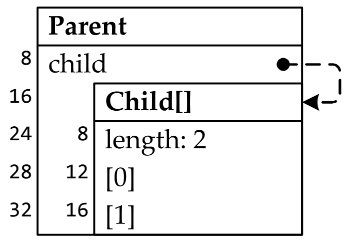
Compared to a field access, an array access needs two additional machine instructions for the bounds check. The first instruction compares the index with the length of the array. The second one branches to an out-of-line code block that throws an exception. Figure 7.2 a) shows the unoptimized HIR and LIR for the method `access()`. The LIR operations 14 and 16 check whether the array index (register `edx`) is within the bounds of the array (register `ecx`) before the array load is performed by the operation 18. These three LIR operations are emitted for the HIR instruction `a4`.

```
Parent allocation() {                Child access(Parent p, int n) {
  Parent p = new Parent();             return p.child[n];
  p.child = new Child[2];            }
  return p;
}
```

a) Java source code

```
tid  instruction
a1   new Parent
a14  co-allocation a1, a9
i8   2
a9   new Child[i8]
a10  a1._8 := a9 Parent.child
a11  areturn a1
```

b) HIR of method `allocation()`



c) Memory layout

Figure 7.1: Example for fixed array inlining

```
tid  instruction
a3   a1._8 Parent.child
a4   a3[i2]
a5   areturn a4

nr  operation
// ecx: p  edx: n
12  move [ecx + 8] -> eax
14  cmp  edx, [eax + 8]
16  branch aboveOrEqual Exception
18  move [eax + edx*4 + 12] -> eax
22  return eax
```

a) Unoptimized method `access()`

```
tid  instruction
a4   (a1+16)[i2] length:2
a5   areturn a4

nr  operation
// ecx: p  edx: n
12  cmp  edx, 2
14  branch aboveOrEqual Exception
16  move [ecx + edx*4 + 28] -> eax
20  return eax
```

b) Optimized method `access()`

Figure 7.2: HIR and LIR for fixed array inlining

The field load instruction `a3` is not necessary in the optimized HIR shown in Figure 7.2 b). Instead, the array load `a4` uses the additional fixed offset 16 to access the array element relative to the `Parent` object `a1`. The array load operation 16 therefore uses the offset 28 instead of 12. Additionally, the array load instruction is augmented with the fixed array length 2, which is used by the LIR operation 12 for the bounds check. Instead of three memory loads in the unoptimized code, only one load is necessary in the optimized code.

7.1.2 Variable Array Inlining

If the objects of a parent class point to arrays with different but fixed lengths, the field accesses can be eliminated in the same way as with object inlining. However, a parent object can only have one such inlining child. Because the size is not known at compile time, a variable-length array must be the last child. The inlining offset of a subsequent child could not be computed by the compiler.

Figure 7.3 shows a modified version of the example. The length of the allocated array is not known at compile time because it is passed as a parameter to the method `allocation()`. The HIR instruction `a9` for the array allocation uses the method parameter `i1` as the length operand. Co-allocation is still possible. The size of the memory chunk for the parent and the child is no longer fixed, but also involves the array length. It is a prerequisite for co-allocation that the length of a variable-sized array is available to the co-allocation instruction, i.e. the HIR instruction that computes the length must be located in front of the co-allocation instruction. For example, it is not allowed that the array length is returned by a method that is called after the allocation of the parent object but before the allocation of the child array. Array inlining fails in such cases.

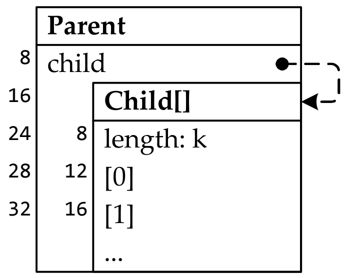
```
Parent allocation(int k) {
  Parent p = new Parent();
  p.child = new Child[k];
  return p;
}
```

```
Child access(Parent p, int n) {
  return p.child[n];
}
```

a) Java source code

```
tid__instruction____________________
a2   new Parent
a14  co-allocation a2, a9
a9   new Child[i1]
a10  a2._8 := a9 Parent.child
a11  areturn a2
```

b) HIR of method `allocation()`



c) Memory layout

Figure 7.3: Example for variable array inlining

```
tid__instruction____________________
a3   a1._8 Parent.child
a4   a3[i2]
a5   areturn a4

nr__operation_______________________
// ecx: p  edx: n
12  move [ecx + 8] -> eax
14  cmp  edx, [eax + 8]
16  branch aboveOrEqual Exception
18  move [eax + edx*4 + 12] -> eax
22  return eax
```

a) Unoptimized method `access()`

```
tid__instruction____________________
a4   (a1+16)[i2]
a5   areturn a4

nr__operation_______________________
// ecx: p  edx: n
12  cmp  edx, [ecx + 24]
14  branch aboveOrEqual Exception
16  move [ecx + edx*4 + 28] -> eax
20  return eax
```

b) Optimized method `access()`

Figure 7.4: HIR and LIR for variable array inlining

Figure 7.4 shows the HIR and the LIR for the unoptimized and the optimized access of a variable-sized array. The left hand side is equal to Figure 7.2. On the right hand side, the memory access for the array bounds check cannot be eliminated, so two memory loads are necessary in the optimized code. Both the bounds check operation 12 and the array load operation 16 use the additional offset to access the array relative to the `Parent` object.

7.1.3 Dynamic Array Inlining

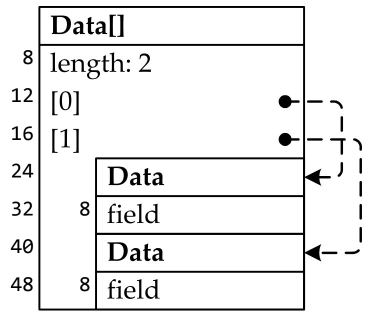
If an array field is assigned multiple times, it is no longer safe to eliminate the field access without further checks. In contrast to object inlining, it is possible to detect whether an array field has been changed at run time without additional overhead in the common case. Section 7.3 discusses the different possibilities.

7.2 Arrays as Inlining Parents

Reference arrays contain pointers to other objects or arrays. Therefore, it would be beneficial to combine an array with the objects that are referenced by the array elements, i.e. to allow arrays as inlining parents. Figure 7.5 shows the resulting object structure when the two `Data` objects referenced by a `Data[]` array would be inlined. The access `p[n].field` could be performed using the address arithmetic `p+n*16+32`. However, we claim that this optimization is impossible without a global data flow analysis because of the nature of the array access bytecodes.

```java
Data[] allocation() {
  Data[] p = new Data[2];
  p[0] = new Data();
  p[1] = new Data();
  return p;
}

int access(Data[] p, int n) {
  return p[n].field;
}
```

a) Java source code



b) Memory layout

Figure 7.5: Inlining with an array as the inlining parent

Java bytecodes are executed using an operand stack. Most bytecodes pop their arguments from the stack, perform an operation, and then push the result back on the stack. Only arguments that are constant for the Java source language compiler, such as numeric constants or offsets of local variables, are part of the bytecodes. Other examples of such constants are indices into the constant pool of the class. The bytecodes that load and store fields, `getfield` and `putfield`, include a symbolic reference to the name of the accessed field and the class to which the field belongs. The

VM linker converts the symbolic reference to a field offset. With this static information, it is possible to find out which fields of which classes are changed at run time.

In contrast, the bytecodes that load and store elements of reference arrays, `aaload` and `aastore`, have no static metadata. Both the array and the index of the accessed element are taken from the operand stack. The lack of static type information inhibits inlining of array elements. It is not possible to find out which array is modified by an `aastore` bytecode, i.e. it is not possible to insert the parent of the array element into the method table. Using the declared type of variables, e.g. the type of the method parameter if an array is passed as a parameter, would be possible in some cases, but is not sufficient in general.

Array stores would need a concept similar to our guards for field stores. As described in Section 6.4 on page 67, a static check at compile time is not possible. A field store can be guarded because precise information about the modified field, i.e. its parent class, is available from the bytecodes. In contrast, for an `aastore` bytecode we do not know the type of the affected array, so an `aastore` bytecode can possibly modify any reference array on the heap. Therefore, it is impossible to guarantee the second precondition, i.e. that a reference to a child object is not modified after it was assigned for the first time.

Figure 7.6 and Figure 7.7 illustrate the differences between the `putfield` and the `aastore` bytecodes. Figure 7.6 corresponds to the object structure presented in Figure 7.3. Assume that the field `child` of the class `Parent` should be inlined. The bytecode `putfield Parent.child` in method `m1` tells the object inlining system that the field `child` of the class `Parent` is modified here. If the method modifies an already initialized `Parent` object, the object inlining system knows that the field is not inlinable. Similarly, the bytecode `putfield Other.child` in method `m2` affects the inlining of the field `Other.child`, but not the inlining of `Parent.child`. In other words, the type in the `putfield` bytecode tells us which objects are modified.

```
void m1(Parent p, Child[] c) {      0: aload_0
  p.child = c;                      1: aload_1
}                                   2: putfield Parent.child

void m2(Other p, Child[] c) {       0: aload_0
  p.child = c;                      1: aload_1
}                                   2: putfield Other.child
```

Figure 7.6: Java source code and bytecodes for field stores

Figure 7.7 corresponds to the example presented in Figure 7.5. Because the `aastore` bytecode is not typed, the same bytecodes are emitted for the methods `m1` and `m2`. In the method `m2`, the variable `p` is declared of type `Object[]`, while in fact it might reference a `Data[]` array because `Data[]` is assignment compatible with `Object[]`. When `p[0]` is modified, we do not know whether an `Object[]` array or a `Data[]` array is affected. The method `m2` therefore prohibits array element inlining of `Data[]` and all other array types.

```
void m1(Data[] p, Data c) {          0: aload_0
  p[0] = c;                          1: iconst_0
}                                    2: aload_1
                                     3: aastore

void m2(Object[] p, Data c) {        0: aload_0
  p[0] = c;                          1: iconst_0
}                                    2: aload_1
                                     3: aastore
```

Figure 7.7: Java source code and bytecodes for array stores

Methods like `m2` are common in most applications. For example, the method `ArrayList.set()` of the frequently used collection class `ArrayList` modifies an element of an `Object[]` array. Therefore, our implementation cannot handle arrays as inlining parents.

Only a global data flow analysis can solve this problem. It is necessary to know all contexts where the method `m2` is called. It must then be checked whether a `Data[]` array can be passed to this method. Although such an analysis would be possible, global reasoning about Java classes is complicated by the dynamic features of Java like lazy class loading and reflection.

7.3 Implementation of Dynamic Array Inlining

Arrays are used to model dynamic data structures in object-oriented applications. Because the number of fields in an object is fixed, one has to allocate and link multiple objects to model e.g. a list with a variable number of elements. Using an array, all elements can be stored in a single array with the appropriate length. When elements are added and the length of the array does not suffice, the common solution is to allocate a larger array, copy all existing elements from the old to the new array, and then discard the old one. This strategy is used for example in the Java collection class `ArrayList`.

7.3.1 Basic Principle

The location and order of objects on the heap can only be influenced during allocation and garbage collection. It is not possible to move objects at other times because all references to these objects would have to be updated. We use the following approach to allow inlining of changing array fields:

- At allocation, the child array with the initial size is co-allocated with the parent object, so an optimized access is possible.
- After the field has been overwritten with a reference to a new array, an optimized access using address arithmetic is no longer possible because it would still access the old array.

- The next garbage collection colocates the new array to the parent object. Therefore, an optimized access is possible again.

Figure 7.8 shows the third version of the example started in Section 7.1. The array field `child` of the class `Parent`, which references a `Child[]` array, is inlined. In contrast to the previous examples, the field can now be changed. The array lengths are the constants 2 and 4 to simplify the example, but they could also be any non-constant value. The co-allocation of the `Parent` object and the `Child[]` array shown in Figure 7.8 b) is equal to the previous example. However, the method `resize()` in Figure 7.8 c) also contains a field store of the field `child`. It modifies an existing `Parent` object that is passed as a parameter. The method `Arrays.copyOf()` is a utility method that allocates a new array and initializes it with the elements of the old array.

```
Parent allocation() {
  Parent p = new Parent();
  p.child = new Child[2];
  return p;
}
```

```
void resize(Parent p) {
  p.child = Arrays.copyOf(p.child, 4)
}

Child access(Parent p, int n) {
  return p.child[n];
}
```

a) Java source code

```
tid__instruction____________________
a1   new Parent
a14  co-allocation a1, a9
i8   2
a9   new Child[i8]
a10  a1._8 := a9 Parent.child
a11  areturn a1
```

b) HIR of method `allocation()`

```
tid__instruction____________________
a2   a1._8 Parent.child
i3   4
a7   invokestatic(a2, i3)
     java.util.Arrays.copyOf()
a11  a1._8 := a7 Parent.child
v12  return
```

c) HIR of method `resize()`

Figure 7.8: Example for dynamic array inlining

Initially, the `Parent` object and the `Child[]` array are colocated as shown in Figure 7.9 a). When the method `resize()` installs a new `Child` object into the `Parent` object, it is not possible to immediately colocate it with the `Parent` object. The Java HotSpot VM allows neither to allocate the new `Child[]` array next to the existing `Parent` object nor to move the `Parent` object to a new location in front of the new `Child[]` array. Both operations could not be performed without influencing other objects and references to objects.

Therefore, the colocation can only be established during the next run of the garbage collector, which deallocates the old array and colocates the `Parent` object with the new `Child[]` array, as shown in Figure 7.9 c). Because the optimized access is not possible between the resize operation and the next garbage collection, an additional colocation check is necessary before an element of the inlined array is accessed.

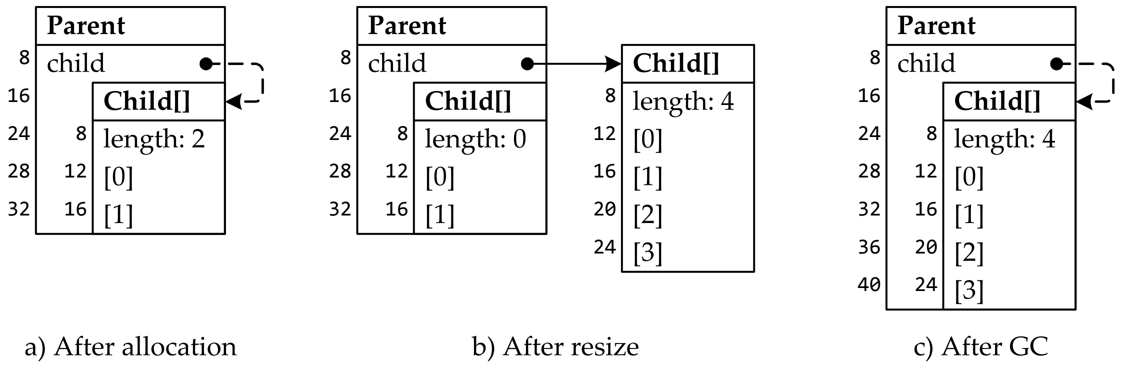


Figure 7.9: Principle of dynamic array inlining

Array inlining saves one machine instruction, i.e. the load of the inlined field `child`, therefore it is only beneficial if the check does not require additional instructions. It is necessary to combine the colocation check with the array bounds check that precedes every array access according to the Java specification.

When an inlined array field is modified, we set the length of the old inlined array to 0, which forces the array bounds check to fail. Instead of throwing an exception immediately, we check whether the field has been changed, i.e. whether it points to a non-colocated array. In this case, we access the new array and continue normally. Only if the bounds check for the new array also fails, an exception is thrown. Therefore, the length of the old `Child[]` array in Figure 7.9 b) is 0. The length is set to 0 by the HIR instruction for the field store `a11` in the method `resize()` (see Figure 7.8 c) before the field is actually modified.

The overhead for accessing the non-colocated array occurs only until the next garbage collection, at which time the new array is colocated with its parent. In order to guarantee that the colocation succeeds, we require that the new array is allocated inside the method that performs the field store and that the allocation sets the isColocationChild bit in the array header. For example, this covers the case where an array that is already the child of another parent object is used as the new array, which would lead to a child with two parents. As a result, the new array is always in the young generation. In our example, this constraint is satisfied because the method `Arrays.copyOf()` is handled by the compiler like an array allocation.

Figure 7.10 shows the normal and the optimized HIR and LIR for the array access. The optimized LIR code is split into a fast path and a slow path. The fast path code performs the optimized array access. One memory load is saved compared to the unoptimized code. When the field is overwritten with the reference to a new array, the length of the old inlined array, i.e. the memory location `[ecx + 24]`, is set to 0, causing the bounds check to always fail. This case is regarded as uncommon, so the code is placed out-of-line at the end of the method in the slow path `S1`. It contains the same code as the unoptimized machine code, i.e. it loads the field and then accesses the array using the normal offsets. Another bounds check throws the exception if necessary.

```
tid__instruction_____________________
a3   a1._8 Parent.child
a4   a3[i2]
a5   areturn a4

nr__operation_______________________
// ecx: p  edx: n
12  move [ecx + 8] -> eax
14  cmp  edx, [eax + 8]
16  branch aboveOrEqual Exception
18  move [eax + edx*4 + 12] -> eax
22  return eax
```

a) Unoptimized method `access()`

```
tid__instruction_____________________
a4   (a1+16)[i2]
a5   areturn a4

nr__operation_______________________
// ecx: p  edx: n
12  cmp  edx, [ecx + 24]
14  branch aboveOrEqual S1
16  move [ecx + edx*4 + 28] -> eax
20  return eax

S1  move [ecx + 8] -> eax
    cmp  edx, [eax + 8]
    branch aboveOrEqual Exception
    move [eax + edx*4 + 12] -> eax
    jump 20
```

b) Optimized method `access()`

Figure 7.10: HIR and LIR for dynamic array inlining

7.3.2 Non-Destructive Approach

The basic principle described above has one severe drawback: overwriting the array length destroys the old array, i.e. it is no longer accessible because the bounds checks for all following accesses fail. This is no problem if the only reference to the array was the inlined array field, because this field already points to the new array. However, the array could also be referenced by fields of other objects or by root pointers. In this case, overwriting the array length is not allowed. An additional analysis must check whether such accesses are possible before dynamic array inlining is initiated.

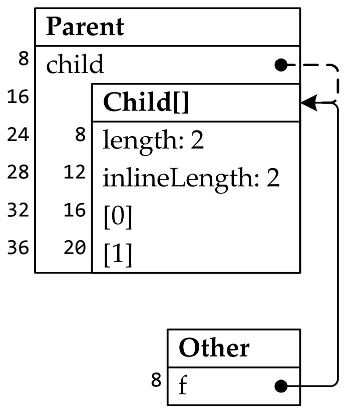
This could be done using a global data flow analysis. An investigation of frequently used classes like `ArrayList` shows that even a method-local bytecode analysis would suffice for most cases. This analysis would check that a child array loaded from the inlined field is used only for an array access and is not assigned to other fields or returned by the method. Nevertheless, additional analysis steps would be necessary and the number of inlinable array fields would be reduced.

Instead of such an analysis, we use a non-destructive approach, which avoids the problem by cloning the array length. Instead of overwriting the regular array length that is also accessed by normal bounds checks, a copy of the array length is overwritten. This copy is accessed only by the optimized machine code. Normal array accesses in compiled code and in the interpreter use the original array length.

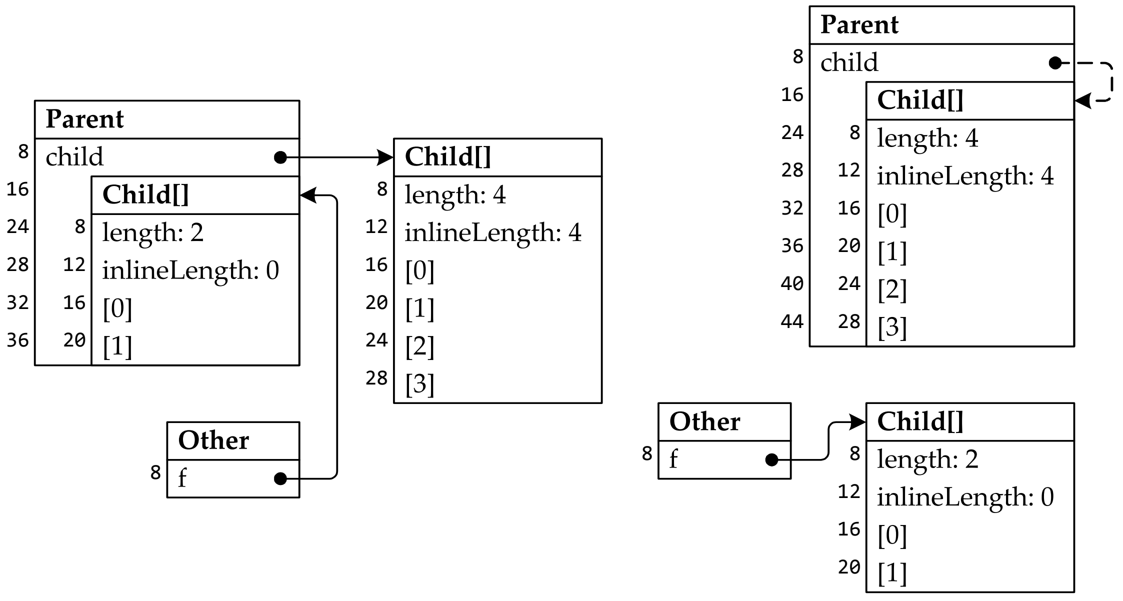
Figure 7.11 shows this approach. Each array has two length fields: `length` and `inlineLength`. In contrast to the previous example, the `Child[]` array is now also accessible from the field `f` of an `Other` object. When the field `Parent.child` is changed, the `inlineLength` of the inlined array is set to 0, but the `length` remains unchanged. Therefore, the array access `other.f[m]` in the method `accessOther()` is still possible.

```
Parent allocation(Other other) {
  Parent p = new Parent();
  p.child = new Child[2];
  other.f = p.child;
  return p;
}

void resize(Parent p) {
  p.child = Arrays.copyOf(p.child, 4)
}

Child access(Parent p, int n) {
  return p.child[n];
}

Child accessOther(Other other, int m) {
  return other.f[m];
}
```

a) Java source code



b) After allocation



c) After resize d) After GC

Figure 7.11: Non-destructive dynamic array inlining using cloned array length

The next run of the garbage collector restores the colocation of the `Parent` object and the new `Child[]` array. Because the old `Child[]` array is still accessible, it is preserved by the garbage collector and moved to a different location. The `inlineLength` of the old array remains 0, but is no longer accessed.

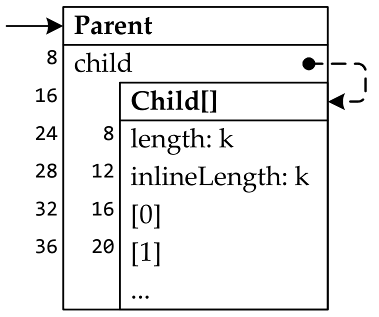
To allow a uniform array access, all arrays on the heap must have the second length field, which increases the required heap space. It would also be possible to place the copy of the length field not into the array, but at the end of the parent object. This reduces the required memory, but leads to a more complicated inlining process because the size of parent objects is changed during inlining. Our implementation uses the simple non-destructive approach where all arrays have two length fields.

7.4 Support for Class Hierarchies

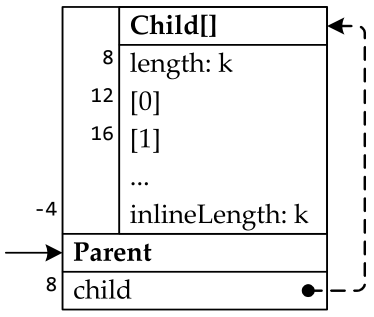
To support subclasses of the parent object's class, it is necessary to reverse the object order, i.e. to place the parent object after its child objects. Section 6.8 on page 76 showed that this does not need architectural changes for object inlining but only leads to different offsets in optimized field loads. Because the size of child objects is fixed, the fields of children can be accessed using negative offsets relative to the parent objects. Arrays with variable size complicate the reverse order because the offset of the first array element is no longer fixed. It is necessary to compute the offset at run time.

7.4.1 Reverse Order for Arrays

To compute the size of an array, the number of elements and the size of each element must be known. When the child array is placed in front of the parent object, the normal array length is not directly accessible because its offset relative to the parent object is not fixed. It is therefore necessary to place a copy of the array length at the end of the array. This is no additional overhead because dynamic array inlining needs a copy of the array length anyway. Figure 7.12 compares the memory layouts for a dynamic array. Only offsets that are compile-time constants are shown in Figure 7.12 b), all other offsets depend on the array length.



a) Normal object order



b) Reverse object order

Figure 7.12: Reverse object order for arrays

When using the reverse order, only the `inlineLength` of the `Child[]` array is directly accessible from the `Parent` object. The method `access()` introduced in the previous sections performs the array access `p.child[n]`. To load the array element at the index `n` without loading the `child` pointer, the following address arithmetic is necessary (the element size of the array is 4 bytes):

```
    p - 4 - inlineLength * 4 + n * 4
```

The computation can be transformed to:

```
p + (n - inlineLength) * 4 - 4
```

This address computation and the subsequent load of the computed memory address require two machine instructions: the subtraction `n - inlineLength` and the memory load. The multiplication by 4 and the subtraction of the constant offset are performed by the indexed addressing mode of the Intel IA-32 architecture. If implemented naively, the additional subtraction instruction increases the number of machine instructions necessary for the array access. However, this address computation can be folded into the array bounds check.

The array bounds check compares the array index `n` with the `inlineLength`. On most architectures, the comparison of two numbers is internally implemented as a subtraction. The result of the subtraction is used to set the flags register, but is then discarded. If a compare instruction is replaced by a subtraction instruction, the result is written to a register, but still used to set the flags register.

Figure 7.13 shows the LIR for the optimized array access with reverse object order. In comparison to the example of the previous section, the LIR operation 12 in Figure 7.13 b) now uses a `sub` operation instead of a `cmp` operation for the bounds check. The result `n - inlineLength` is written to the register `edx`. This value is negative because `inlineLength` must be greater than `n` for valid array accesses.

```
nr__operation_________________________
// ecx: p  edx: n
12  move [ecx + 8] -> eax
14  cmp  edx, [eax + 8]
16  branch aboveOrEqual Exception
18  move [eax + edx*4 + 12] -> eax
22  return eax
```

a) Unoptimized method `access()`

```
nr__operation_________________________
// ecx: p  edx: n
12  sub  edx, [ecx - 4] -> edx
14  branch aboveOrEqual S1
16  move [ecx + edx*4 - 4] -> eax
20  return eax

S1  add  edx, [ecx – 4] -> edx
    move [ecx + 8] -> eax
    cmp  edx, [eax + 8]
    branch aboveOrEqual Exception
    move [eax + edx*4 + 12] -> eax
    jump 20
```

b) Optimized method `access()`

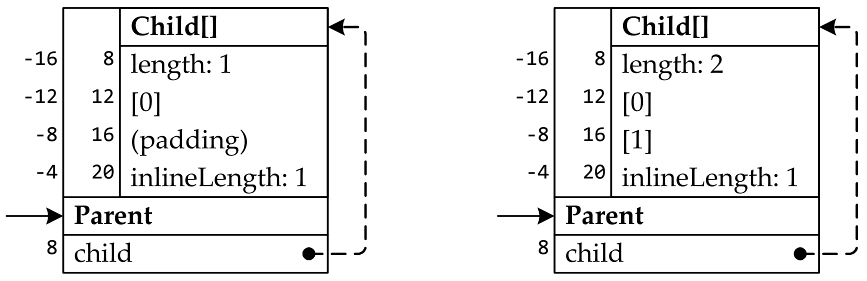
Figure 7.13: LIR for array access with reverse order

The subsequent memory access of the LIR operation 16 uses a negative offset. As usual, the slow path `S1` is taken if the array index is out of bounds or the array field has been changed, i.e. if the `inlineLength` was set to 0. The slow path must undo the subtraction by adding the subtracted `inlineLength` to the register `edx`.

7.4.2 Object Alignment

The 8-byte alignment of objects and arrays complicates the basic scheme for the reverse order of arrays. Because the size is rounded up to the next multiple of 8, arrays with different lengths can have the same size. The optimized array access must consider the padding, which is inserted between the array elements and the field `inlineLength`. To avoid conditional operations in the address arithmetic that was presented in the previous section, the padding must be incorporated into the only part of the formula that is loaded from the actual array object: the field `inlineLength`.

Figure 7.14 illustrates the exact memory layout of inlined arrays with the lengths 1 and 2. Both arrays have the same size of 24 bytes, and the offset of the first array element relative to the `Parent` object is -12 in both cases. Therefore, the field `inlineLength` for both arrays must be equal, otherwise the subtraction of the `inlineLength` could not be uniformly used. To make the array bounds check safe in all cases, the minimum of the two lengths must be used, i.e. the array with a `length` of 2 has an `inlineLength` of 1.



a) Array with length 1　　b) Array with length 2

Figure 7.14: Memory layout with 8-byte alignment

The reduced `inlineLength` does not impact the correctness because the slow path of the array access handles indices that are above or equal the `inlineLength`, but below the `length` of the array. For example, the slow path is used to access the last element of all inlined `Child[]` arrays with an even length. Since `inlineLength` is now smaller than the number of words between the field `inlineLength` and the beginning of the array, an additional constant offset of -4 must be added to the LIR operations 12 and 16 in Figure 7.13 b).

The rounding granularity of the `inlineLength` and the additional offsets for LIR operations depend on the element size of the arrays. Arrays with an element size of 1 (`byte[]` arrays) require that the `inlineLength` is rounded down to the nearest multiple of 8 plus 1. For example, all `byte`-arrays with the lengths 1 to 8 have the same size and therefore the `inlineLength 1`. In contrast, `long[]` arrays with an element size of 8 require no rounding at all.

In summary, array inlining of dynamic arrays with reverse order uses the array bounds check for several purposes:

- *Detection of field changes*: The slow path is used if the array field has been modified and an optimized array access is no longer possible, i.e. if the `inlineLength` is 0.
- *Address computation for the first array element*: The compare instruction of the bounds check is replaced by a subtraction instruction.
- *The 8-byte alignment of objects*: The slow path is used to handle the rounded `inlineLength` appropriately.
- *The actual bounds check*: Indices that are out of the valid array bounds are detected.

In the fast path of the optimized array access, the field load of the inlined array field is eliminated, so one memory access is saved. However, a slow path that performs the unoptimized array access is needed for correctness.

7.5 Limitations

Dynamic array inlining is only beneficial if the majority of array accesses use the fast path. The number of slow path accesses is high if the field is changed frequently or if the timeframe between a field modification and the next garbage collection is long. In such cases, the array access should be reverted to the unoptimized machine code. Furthermore, dynamic array inlining does not allow optimizing direct accesses to the array length, e.g. when generating code for the `arraylength` bytecode.

Loads of inlined object fields can be optimized by *load folding* or *address computation* (see Section 6.6 on page 70). For dynamic arrays, only load folding is allowed because the bounds check of the array access is required for correctness. To compute the address of the inlined child array, the same checks would be necessary, i.e. the `inlineLength` had to be accessed. This would be more complex than the access of the array field and thus not beneficial. If the array field is accessed and the array is used e.g. as a method parameter, no optimization can be performed.

7.5.1 Access of the Array Length

The array length is loaded implicitly by the bounds check before each array access. However, it is also possible to load the length explicitly using the `arraylength` bytecode and to use it e.g. for mathematical computations. The access could be optimized using the `inlineLength`, but this would be complicated and would require additional code for the correctness. The additional check whether the result of the optimized access is 0 would require a compare and a branch instruction, which is more expensive than the unoptimized access. Additionally, the `inlineLength` can be rounded due to the 8-byte alignment, which makes it unusable for explicit loads of the array length. As a result, we retain the unoptimized field load when the array length is requested explicitly.

7.5.2 Interdependencies with Garbage Collection

To be beneficial, dynamic array inlining requires that the timeframe between the modification of an array field and the next garbage collection is short. The Java HotSpot VM divides the heap into two generations: the small young generation that is collected frequently because most objects die young, and the old generation that contains only long-living objects. It is larger than the young generation and is collected less frequently.

Therefore, it makes a difference whether the parent object is in the young or in the old generation. When an array field is changed, the new array is always in the young generation because it was recently allocated. However, the colocated object order can only be restored by a collection of the young generation if the parent is also still in the young generation. Heuristics in the age calculation can keep the parent object in the young generation as long as the inlined array field is changing. For example, we use the age of the child array instead of the age of the parent object to decide whether the parent object should be promoted. When the array field is changed frequently, the age of the array is always low and the parent object is not promoted.

7.5.3 Interdependencies with Array Bounds Check Elimination

Array bounds check elimination removes checks of array indices that are proven to be in the valid range (see for example [Bodík00] and [Qian02] for Java implementations and [Würthinger07b] for an algorithm integrated into the Java HotSpot client compiler). If the index variable is guaranteed to be below the array length, the check can be omitted (*fully redundant* checks). If the check is in a loop, the array length is loop invariant, and the maximum value of the index variable is known, then the check can be moved out of the loop (*partially redundant* checks).

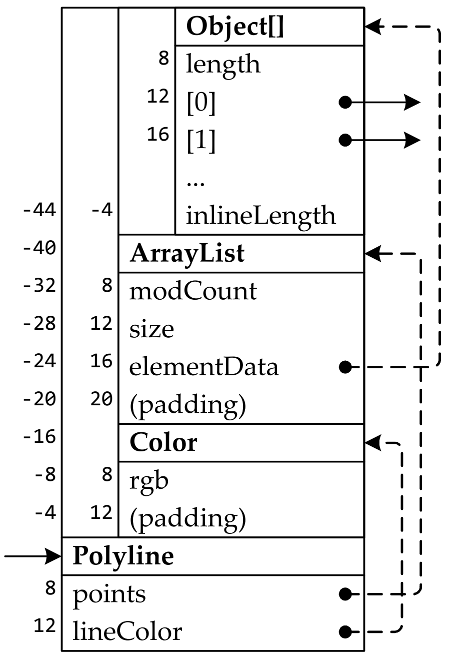
We use the bounds check to detect changes of the array field. If the bounds check is eliminated, the optimized array access is no longer possible. Therefore, there is an optimization conflict between bounds check elimination and dynamic array inlining. In practice, such conflict situations are unlikely. Bounds check elimination requires static information about the length of an array. If the array was just loaded from a field, which is the pattern optimized by array inlining, such information is usually not available and bounds check elimination is not possible. When bounds checks are moved out of a loop, the check whether the array field has been changed can still be performed outside the loop.

7.6 Java Class Library

This section shows the results of object and array inlining for classes of the Java class library. Because our optimization is performed at run time, all classes used by an application are optimized uniformly. There is no difference between application classes, library classes, and system classes.

7.6.1 ArrayList Example

The example of this thesis uses the collection class `ArrayList` to store a variable list of points of a `Polyline` (see Java source code in Figure 3.1 on page 26). Figure 7.15 shows the combined benefits of object and array inlining when accessing a point stored in the `ArrayList`. The `Polyline` object has two inlining children: the `Color` object referenced by the field `lineColor` and the `ArrayList` object referenced by the field `points`. The `Object[]` array of the `ArrayList` is an indirect inlining child of the `Polyline`. Dynamic array inlining is necessary because the array can be replaced by a larger copy when new elements are added to the `ArrayList`.



a) Memory layout

```
nr__operation________________________
// ecx: this  edx: index
12  move [ecx - 28] -> eax
14  cmp  edx, eax
16  branch greaterOrEqual 32
22  sub  edx, [ecx - 44] -> edx
24  branch aboveOrEqual S1
26  move [ecx + edx*4 - 48] -> eax
30  return eax

32  ...

S1  add  edx, [ecx – 44] -> edx
    move [ecx - 24] -> eax
    cmp  edx, [eax + 8]
    branch aboveOrEqual Exception
    move [eax + edx*4 + 12] -> eax
    jump 30
```

b) Optimized method `Polyline.getPoint()`

Figure 7.15: Combined object and array inlining for `ArrayList`

The unoptimized access of the `ArrayList` in the method `Polyline.getPoint()` requires several field loads and method invocations. At first, the field `Polyline.points` must be loaded. Because this field is declared to be of the interface type `List`, dynamic binding is necessary to invoke the method `get()` of the class `ArrayList`. This method loads the field `ArrayList.size` to check whether the accessed index is valid. Then, the field `ArrayList.elementData` is loaded and the array is accessed. Although the `ArrayList` has already performed an index check, a bounds check is necessary for the array load.

Figure 7.15 b) shows the optimized LIR of the method. At first, it is not necessary to load the field `Polyline.points`. The field is inlined, so it is known that the offset relative to the `Polyline` object is -40 and that the exact type of the referenced object is `ArrayList`. No dynamic binding is necessary, and the method `ArrayList.get()` can be inlined. LIR operation 12 accesses the field `ArrayList.size` relative to the `Polyline` object. The subsequent compare and branch operations check that the accessed index is valid. The error handling code starting with LIR operation 32 is uncritical for performance because it is normally not executed.

The LIR operations 22, 24, and 26 perform the optimized bounds check and array load. As shown in the previous sections, the bounds check uses a subtraction operation that sets the flags for the subsequent conditional branch and also modifies the index for the memory load operation 26. The `inlineLength` of the array is accessed directly from the `Polyline` object using the offset -44. Finally, LIR operation 26 performs the optimized array load.

The slow path code `S1` of the array load must undo the subtraction of the bounds check. Then, the field `ArrayList.elementData` is loaded and a normal array access is performed. In the slow path, it is again not necessary to load the field `Polyline.points`. The slow path is executed if the array field changed, but no garbage collection happened yet. It is also executed if the last element of the array is accessed and the array length is even. In this case, the `inlineLength` is smaller than the actual array length in order to handle the 8-byte object alignment.

7.6.2 Other Collection Classes

Object inlining is possible for all collection classes similarly to the example of the previous section. A field of a business object that references a collection object is usually assigned only once in the constructor or shortly afterwards. Co-allocation requires that the allocations of the business object and the collection object are in the same compiled method. This is achieved by aggressive inlining of methods.

Many collection classes use arrays for their internal data structures. Such arrays can be optimized using array inlining. Optimized array loads are possible for array-based list implementations like `ArrayList` and `Vector`, as shown in the previous section. Collection classes that are based on hash codes, e.g. the class `HashMap`, are more difficult to optimize. The data array is not indexed with a user-supplied index, but with a computed hash code. The length of the data array is used to map arbitrary hash code values to the valid range of array indices, e.g. using a modulo operation. As shown in Section 7.5.1, the optimized access of the array length is therefore not possible and the array field must be loaded. Load folding for the subsequent array access would be possible, but not profitable because the array address is already available.

7.6.3 Strings

At first glance, Java strings look like ideal candidates for array inlining. A `String` object references a `char[]` array via the field `String.value`. Because strings are immutable, the field is initialized once in the constructor and never changed afterwards. Most string operations load this field to access the characters. Unfortunately, the implementation of the `String` class prohibits array inlining. The same `char[]` array can be shared between multiple `String` objects. Figure 7.16 shows the memory representation of the string `"abcd"` and the substring `"bc"`, which is returned by the method call `"abcd".substring(1, 3)`.

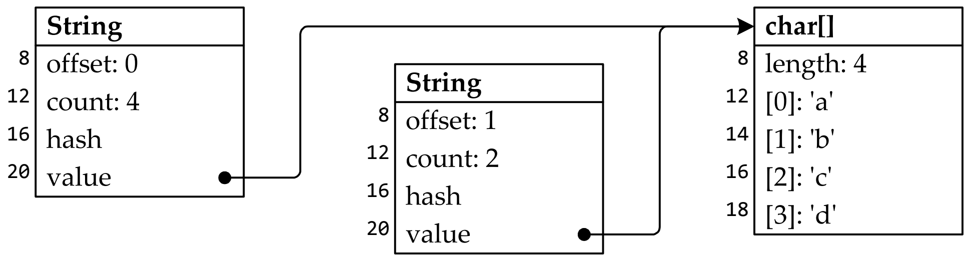


Figure 7.16: Memory representation of a string and a substring

The content of a string is not always the whole `char[]` array referenced by the field `value`. Instead, the fields `offset` and `count` specify the relevant range of the array elements. This allows an efficient implementation of methods that copy parts of a string, e.g. the method `String.substring()`. It is not necessary to create a smaller copy of the character array, but only to allocate a `String` object and set the fields `offset` and `count` accordingly.

The sharing of the `char[]` array prohibits its colocation with the `String` object. One `char[]` array would have two parents, which is not allowed. Therefore, optimized accesses of the array are not possible. In our approach, this is detected by the just-in-time compiler when co-allocation is not possible, e.g. in the method `String.substring()`. To avoid the always failing analysis, we manually exclude the class `String` from inlining. However, object colocation is performed for strings. Because strings are accessed frequently in most applications, a dedicated optimization of strings can be beneficial. A recently started project evaluates possibilities of object inlining for strings [Häubl07].

Evaluation

This chapter evaluates the implementation of object and array inlining. It compares different configurations where object colocation, object inlining, and array inlining are selectively enabled. The speedup of these configurations and the number of eliminated field loads is presented for the SPECjvm98 and DaCapo benchmark suites. Additionally, the compile-time impact of the optimizations is evaluated.

Our implementation of object and array inlining is integrated into Sun Microsystems' Java HotSpot VM. At the time of writing, the most recent version was the early snapshot release b21 of the upcoming JDK 7 [SunJava7]. We use the default configurations for client applications, i.e. the client compiler for just-in-time compilation and the generational, non-parallel, and non-concurrent garbage collector (see Chapter 2 on page 9). In addition to the optimizations of the current product version, our client compiler also performs array bounds check elimination based on the algorithm of Würthinger [Würthinger07b].

The optimizations described in this thesis can be selectively enabled using command line flags. We use the following four configurations for the evaluation:

- *Baseline*: All of our analyses and algorithms are disabled.
- *Colocation*: This configuration combines the impact of read barriers to detect hot fields (see Chapter 4 on page 41) and object colocation (see Chapter 5 on page 47) during garbage collection to improve the cache behavior. Objects and arrays are optimized uniformly by the garbage collector.
- *Object Inlining*: In addition to the previous configuration, inlinable object fields are detected and field loads are removed (see Chapter 6 on page 57). As a prerequisite, method tracking is also enabled. Array fields are not inlined, but still colocated by the garbage collector.
- *Array Inlining*: Both object fields and array fields are inlined in this configuration. We use the non-destructive reverse dynamic array inlining scheme that requires the duplication of the array length field (see Chapter 7 on page 79).

We report three different kinds of results for the benchmarks. First and most important are the benchmark times, i.e. the speedup that can be achieved by the optimizations. We do not report absolute times because they are highly architecture dependent, but only relative speedups compared to the baseline. Secondly, we count the number of compiled methods and the number of optimized fields in the different configurations. These are static counters that are incremented in the just-in-time compiler or the object inlining subsystem.

Furthermore, we use dynamic counters to collect field access counts. These counters are incremented by the generated machine code, similarly to the read barriers. However, they are active for the whole benchmark run and thus impose a significant run-time overhead, i.e. the benchmarks are significantly slower. Therefore, separate measurements are necessary for the speedup and the dynamic field access counters. Enabling the counters has an impact on object inlining because it changes the timing of our optimization phases, i.e. it changes the time when methods get compiled and therefore when the optimizations take effect. However, we verified that the same fields are optimized in both configurations.

All measurements were performed on an Intel Core 2 Quad processor Q6600 with 2.4 GHz, running Microsoft Windows XP with service pack 2. Each of the four cores has a separate L1 data cache of 32 KByte. Two cores together share a 4 MByte L2 cache, so there are 8 MByte L2 cache in total. All caches have a cache line size of 64 bytes. The main memory of 2 GByte is uniformly accessed by all cores. The results were obtained using a 32-bit operating system and a 32-bit VM.

8.1 Benchmark Results for SPECjvm98

The SPECjvm98 benchmark suite [Spec98] is commonly used to assess the performance of Java runtime environments. It consists of seven programs derived from real-world applications that cover a broad range of scenarios. SPECjvm98 measures the overall performance of a Java VM including class loading, garbage collection, and loading the input data of the benchmarks from files. The programs are executed several times (five times for all numbers reported in this thesis) to allow the VM to apply run-time optimizations. Two results are reported for each benchmark: the *slowest run* and the *fastest run*.

- The slowest run is always the first run in our measurements. It contains the time necessary for class loading, initial interpretation of methods, and just-in-time compilation. Therefore, this number is an indicator for the startup speed of the Java VM. A higher score of the slowest run denotes a faster startup of applications.
- The fastest run is usually the last run of the benchmark. All hot methods were already compiled and all run-time optimizations were applied during previous runs. The execution time has reached a fix point. This number measures the

quality of the machine code generated by the just-in-time compiler as well as the quality of the garbage collector. A higher score of the fastest run denotes a higher peak performance of applications.

8.1.1 Impact on Run Time

Figure 8.1 shows how object colocation, object inlining, and array inlining affect the performance of SPECjvm98. We present the results for the individual benchmarks as well as the geometric mean of all benchmarks. The slowest and the fastest runs are shown in the same figure on top of each other: the gray bars refer to the fastest runs, the white bars to the slowest. Both runs are shown relative to the same baseline, i.e. the fastest run with all our optimizations disabled.

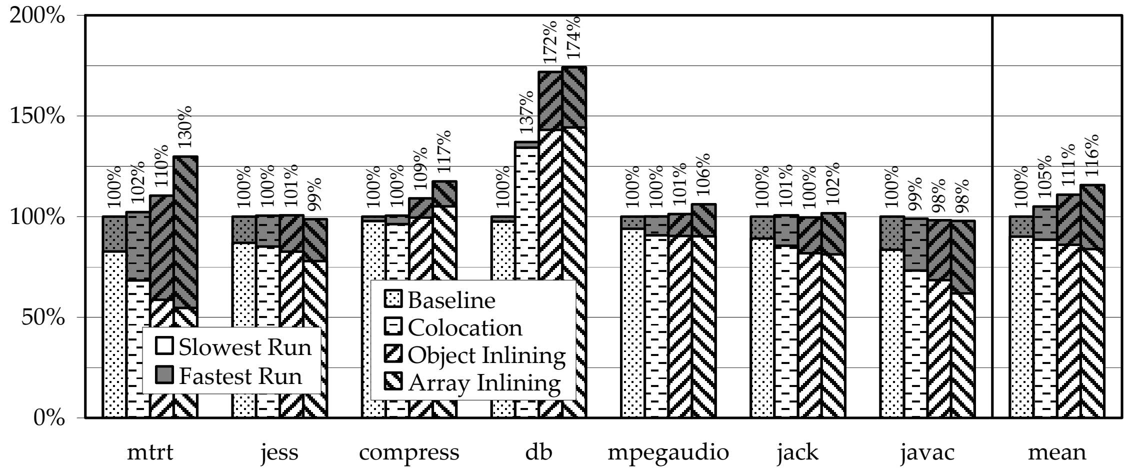


Figure 8.1: Speedup compared to baseline for SPECjvm98 (taller bars are better)

The differences between the slowest and the fastest runs in the first column point out the overhead of just-in-time compilation. The overhead is higher for complex object-oriented benchmarks such as jack and javac. However, the difference is also influenced by the problem size of the benchmark. The more time the first run takes, the lower is the impact of just-in-time compilation. Long-running benchmarks such as mpegaudio have a low difference although many methods are compiled.

The fastest run in the second column shows the benefits of object colocation. This optimization has a major impact on the db benchmark, which accesses a large number of objects. There is also a speedup for the mtrt benchmark. The slowest run shows the overhead of the read barriers, i.e. the overhead of the field counters that are incremented at run time. To reduce this overhead, the read barriers are removed later by recompiling methods, which also impairs the slowest run but eliminates the overhead for the fastest run. Therefore, the slowest run is negatively affected for most benchmarks.

The third column shows the impact of object inlining. The analyses for guaranteeing the preconditions impact the slowest run negatively. However, this overhead is justified by the improved peak performance of the fastest run. The benchmarks that benefit from colocation, db and mtrt, are further improved. Additionally, the compress benchmark shows a speedup.

Array inlining, which is shown in the fourth column, increases the impact of object inlining. The fastest runs of the array-intensive benchmarks mtrt, compress, and mpegaudio are significantly improved. Again, the impact on the slowest runs depends on the benchmark characteristics: the slowest run of the mtrt benchmark is negatively affected because of the increased compilation overhead, while the slowest run of the compress benchmark is improved because the optimization succeeds early enough in the first run.

The javac benchmark shows a slowdown in all configurations because the overhead is higher than the benefit. The reason for the slight regression of peak performance is mostly the overhead of object colocation during garbage collection. This overhead is similar for all benchmarks, but normally outweighed by the improved cache behavior. The javac benchmark has a flat distribution of the field accesses and is not improved by object colocation and inlining.

8.1.2 Field Access Counts

The speedup of object and array inlining is directly related to the number of eliminated field loads. Figure 8.2 shows the distribution of field and array loads and the impact of object and array inlining. The same configurations as in the previous section are used, only the colocation configuration is omitted because it is equal to the baseline. The configurations are abbreviated as *b* (baseline), *o* (object inlining), and *a* (array inlining).

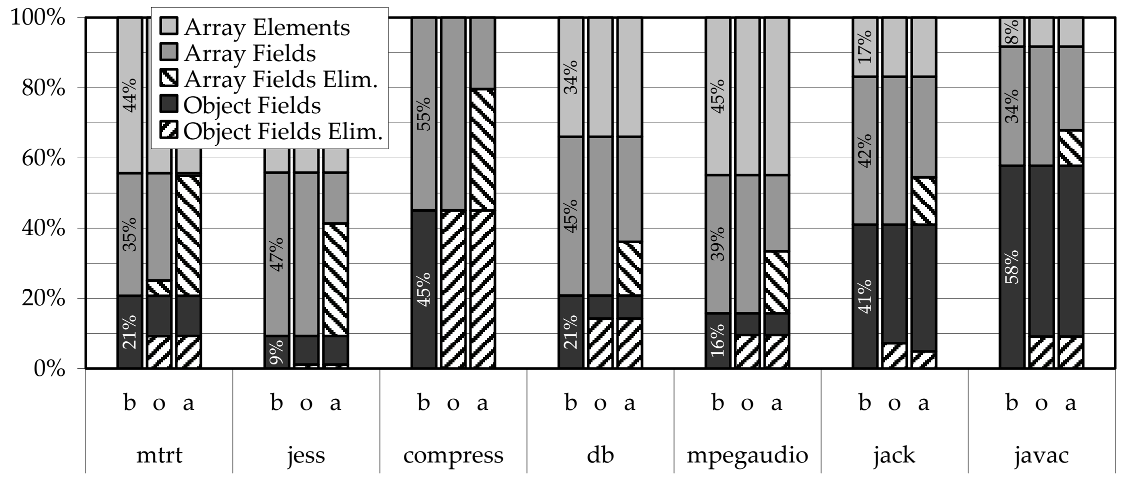


Figure 8.2: Dynamic field access statistics for SPECjvm98

We count all loads of references from the heap, i.e. all loads that could be optimized by inlining, and distinguish between three kinds: *object fields*, *array fields*, and *array elements*. The figure does not contain loads of scalar values such as `int` fields or elements of `int[]` arrays. The first column shows the distribution of the three kinds. This distribution is identical in all configurations.

The subsequent columns highlight the percentage of loads that are eliminated by object inlining and array inlining. The higher the striped bars are, the more field loads are eliminated. As described in Section 7.2 on page 83, array elements cannot be optimized without a global data flow analysis. Therefore, the topmost bar shows no eliminated loads in any configuration.

Object inlining optimizes only object fields, i.e. the bottommost of the three kinds. For the benchmarks mtrt, db, and mpegaudio, a significant part of the object field loads is eliminated. For compress, the whole object graph is combined to one object group, so all reference field loads are eliminated. For mtrt, the number of accessed array fields is slightly reduced by object inlining though no array fields are optimized. This is a beneficial side effect of object inlining on other optimizations such as global value numbering. If it is known that a field never changes, more subsequent array loads are identified as redundant and can therefore be eliminated.

All benchmarks except javac perform more array field loads than object field loads. This demonstrates the importance of array inlining. The benchmarks mtrt, compress, and mpegaudio frequently access arrays whose lengths are compile-time constants. Because the bounds checks can be simplified for such array accesses, the speedup due to array inlining is comparatively high. The other four benchmarks jess, db, jack, and javac mostly use array-based collection classes that can be optimized.

For the jack benchmark, array inlining reduces the number of eliminated object field loads. The benchmark contains a parent class with three object fields that can be inlined using object inlining. Each of these fields references a collection. With dynamic array inlining, the data arrays of the collections are also inlined. Therefore, the offsets of the second and the third child object are no longer fixed. Inlining them fails, so only one object field is inlined.

8.1.3 Number of Optimized Fields

Our analysis performs inlining on a field-per-field basis at run time. To limit the overhead, it is important that no time is wasted analyzing fields that are infrequently accessed. Table 8.1 shows that detecting hot fields using read barriers acts as an effective filter. The first column *initial* shows the overall number of fields in all loaded classes. Based on the declared type, we distinguish object fields, array fields, and scalar fields. Scalar fields are not relevant for object inlining, so they are ignored in the subsequent columns. Most of the several hundred fields are only accessed by methods that run in the interpreter.

	Initial			Counted		Colocated		Inlined	
	Obj.	**Arr.**	**Scalar**	**Obj.**	**Arr.**	**Obj.**	**Arr.**	**Obj.**	**Arr.**
mtrt	380	77	315	40	19	18	11	4	4
jess	410	77	407	67	24	19	10	2	4
compress	352	76	296	13	16	7	9	7	5
db	350	74	289	23	17	4	6	2	2
mpegaudio	412	93	346	73	35	18	19	9	11
jack	395	80	330	80	23	20	12	2	3
javac	492	92	344	148	36	24	11	2	4
empty	315	54	208	0	1	0	0	0	0

Table 8.1: Static field access statistics for SPECjvm98

The column *counted* shows the number of fields for which read barriers are emitted in the compiled code. The read barrier counters are used to determine whether a field is frequently accessed. The number of frequently accessed fields is shown in the column *colocated*. Only 1% to 5% of the object fields and 8% to 20% of the array fields are colocated. This is essential to keep the overhead during garbage collection low.

The column *inlined* shows the number of inlined fields when object and array inlining is performed. Ideally, all colocated fields should also be inlined. However, this is not possible because of the strong preconditions necessary for object inlining. Only for the compress benchmark, all hot object fields can be inlined. For some other benchmarks, we optimize the relevant fields. For example, the 8 inlined fields of the mtrt benchmark lead to a significant speedup. Likewise, the 4 inlined field of the db benchmark are also the most important ones. For other benchmarks such as javac, our analysis cannot inline the most frequently accessed fields.

Even small applications load many classes. The last row of the table shows the field counts when executing an application that consists only of an empty method `main()`. In this case, all loaded fields are part of the class library and are defined in classes that set up the infrastructure for running an application. Even before `main()` is called, Java code is executed. Some methods of the class `String` are frequently executed and therefore compiled. These methods access the array field `String.value`.

8.1.4 Compile-Time Impact

The just-in-time compiler plays an important role in our analysis. We use it to emit read barriers, to perform co-allocation, to guard field stores, and finally to optimize field loads. Therefore, it is necessary to compile more methods, and also to compile some methods multiple times. Table 8.2 shows the number of compiled methods in the different configurations. Unfortunately, it is not possible to measure the overall compilation time because compilation is performed in the background parallel to the execution of the application. The thread scheduling introduces too much noise to the compilation time so that no stable results can be measured.

	Baseline	Colocation		Object Inlining		Array Inlining	
mtrt	126	168	+33%	203	+61%	232	+84%
jess	157	279	+78%	318	+103%	364	+132%
compress	37	57	+54%	75	+103%	91	+146%
db	59	92	+56%	99	+68%	109	+85%
mpegaudio	152	351	+131%	418	+175%	468	+208%
jack	221	341	+54%	379	+71%	430	+95%
javac	527	1046	+98%	1086	+106%	1111	+111%
empty	3	3	+0%	3	+0%	3	+0%

Table 8.2: Number of compiled methods for SPECjvm98

The removal of read barriers for already hot fields requires that methods containing those barriers are recompiled. Because object colocation depends on the read barriers but does not need additional compilations itself, the column *colocation* accurately reflects this overhead. Most methods contain field accesses that are initially counted, so the number of compiled methods increases for all benchmarks.

Object inlining is only initiated for the few frequently accessed fields. Therefore, the additional compilation overhead of the column *object inlining* is modest, even though multiple methods must be compiled to guarantee the preconditions for inlining a field and to optimize the field loads. The count is further increased when *array inlining* is performed. The compilation overhead is not a bottleneck of our implementation because the just-in-time compiler is fast enough and compilation is done in the background while the application continues to run. Therefore, we do not use advanced heuristics to reduce the number of compiled methods.

8.2 Analysis of SPECjvm98

The previous section presented the overall results for the SPECjvm98 benchmark suite. This section analyzes four of these benchmarks in detail: mtrt, db, compress, and jess. They represent different scenarios of object inlining. The remaining three benchmarks are not analyzed because their source code is not available.

8.2.1 mtrt

The mtrt benchmark is a multi-threaded ray tracing application that simulates the course of light rays. It renders a three-dimensional scene, which is represented by a large in-memory data structure. The most frequently executed method checks whether the light ray intersects with a certain `Face` of an `OctNode` object. Figure 8.3 shows the object structure of this data model. Because of the object-oriented programming style, the data is separated into multiple objects. For example, the coordinates of a point are maintained in a separate `Point` object that provides several methods for mathematical operations on the point.

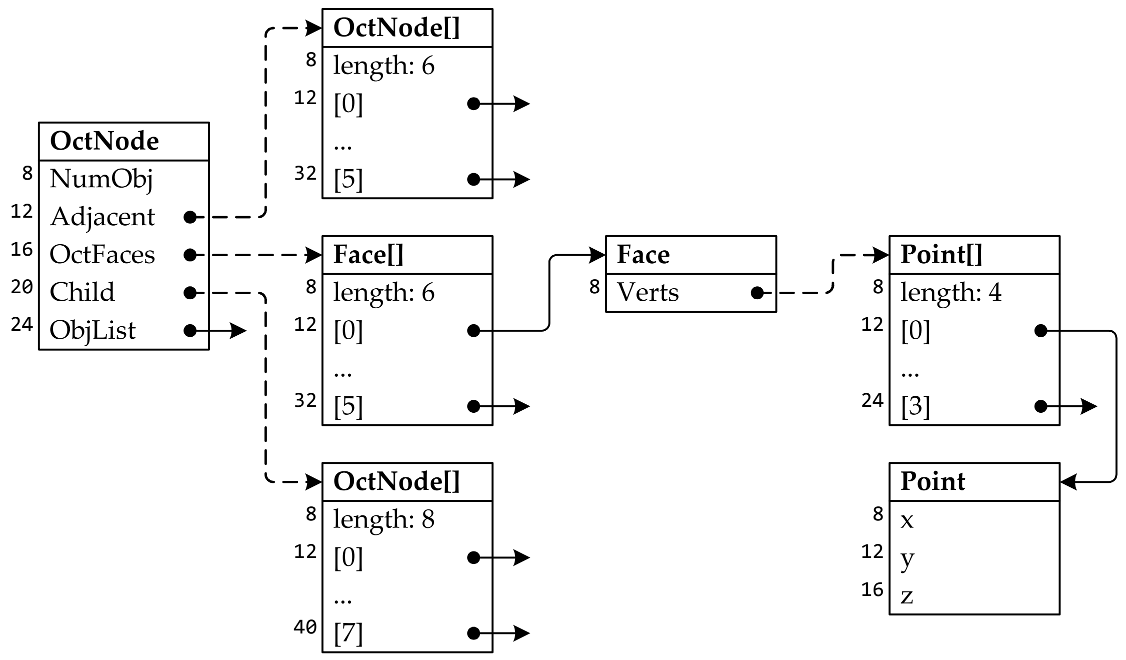


Figure 8.3: Object graph of benchmark mtrt

The method `OctNode.intersect()` accesses the coordinates of a `Point` starting from the `OctNode` object, i.e. it repeatedly performs the following field and array accesses: `this.OctFaces[i].Verts[j].x`. The values for `i` and `j` vary, and all three `Point` coordinates `x`, `y`, and `z` are accessed. The fields `Direction` and `Origin` of `Ray` objects are also accessed in this method to perform the hit testing. Table 8.3 shows that these 6 fields and array elements are the most frequently accessed ones in the benchmark. They account for 80% of all field and array accesses (scalar values are not counted in these statistics).

Field or Array	n*1000	%	Σ	Kind	State
Face[]	32,310	20%	20%	Array element	Colocated
Face.Verts	32,143	20%	40%	Array field	Inlined, length 4
Point[]	32,016	20%	60%	Array element	Colocated
OctNode.OctFaces	14,937	9%	69%	Array field	Inlined, length 6
Ray.Direction	8,968	6%	75%	Object field	Inlined
Ray.Origin	8,899	6%	80%	Object field	Colocated
OctNode[]	6,784	4%	84%	Array element	Colocated
OctNode.Child	6,336	4%	88%	Array field	Inlined, length 8
IntersectPt.Intersection	2,725	2%	90%	Object field	Inlined
ObjNode.theObject	2,461	2%	92%	Object field	Colocated
OctNode.Adjacent	1,778	1%	93%	Array field	Inlined, length 6

Table 8.3: Access statistics for benchmark mtrt

The data model shown above represents a static structure. It is loaded once at startup from a file and is never changed by the ray tracing algorithm. Therefore, all fields are assigned only once in an initialization method and the arrays are allocated with a fixed length. The most important fields can be inlined. Although the loads of array elements cannot be eliminated by our algorithm, the preceding array bounds checks are optimized using the constant array lengths.

The two fields `Direction` and `Origin` of the class `Ray` both reference a point, but the implementation differs. The direction uses value semantics: when the direction is changed, the new `x`, `y`, and `z` coordinates are assigned to the existing object. The field `Direction` is assigned only once in the constructor, so object inlining is possible. In contrast, the origin uses reference semantics: whenever the origin is changed, another `Point` object is assigned to the field `Origin`. Therefore, object inlining of this field is not possible.

We collect the dynamic field access counts by emitting a counter increment in front of each field access in the just-in-time compiler. Object and array inlining are disabled in this configuration. However, the number of field accesses is affected by other compiler optimizations such as value numbering. A field access that is eliminated by such an optimization is also not counted. We believe that this better reflects the actual program behavior than an instrumentation of the interpreter.

To illustrate the differences, Table 8.4 shows the counters with and without value numbering. The rightmost column shows the difference between these two configurations. For example, the access count of the field `OctNode.OctFaces` is more than twice as high without value numbering. Most other studies, such as the analysis of inlinable fields by Lhoták [Lhoták02], use bytecode instrumentation, therefore their numbers match our configuration without value numbering.

	With Value Numbering			**No Value Numbering**			
Field or Array	**n*1000**	**%**	**Σ**	**n*1000**	**%**	**Σ**	**Difference**
Face[]	32,310	20%	20%	32,310	16%	16%	100%
Face.Verts	32,143	20%	40%	32,143	16%	32%	100%
Point[]	32,016	20%	60%	32,076	16%	48%	100%
OctNode.OctFaces	14,937	9%	69%	32,513	16%	64%	218%
Ray.Direction	8,968	6%	75%	17,147	8%	72%	191%
Ray.Origin	8,899	6%	80%	17,069	8%	80%	192%
OctNode[]	6,784	4%	84%	7,224	4%	84%	106%
OctNode.Child	6,336	4%	88%	6,794	3%	87%	107%
IntersectPt.Intersection	2,725	2%	90%	5,188	3%	90%	190%
ObjNode.theObject	2,461	2%	92%	3,276	2%	92%	133%
OctNode.Adjacent	1,778	1%	93%	1,820	1%	93%	102%

Table 8.4: Impact of value numbering on benchmark mtrt

8.2.2 db

The db benchmark loads an address database from a file and performs different query and modification operations on the in-memory data structure. Figure 8.4 shows the object structure of the database. A single `Database` object maintains an array of `Entry` objects. Each `Entry` stores a pointer to a list of strings, which are maintained using the collection class `Vector`. Each `String` stores its value in an array of `char`. In total, there are about 15,600 `Entry` objects that refer to about 125,000 `String` objects. The benchmark spends most of its time sorting the entries. Six pointers must be accessed to retrieve the characters of a `String`.

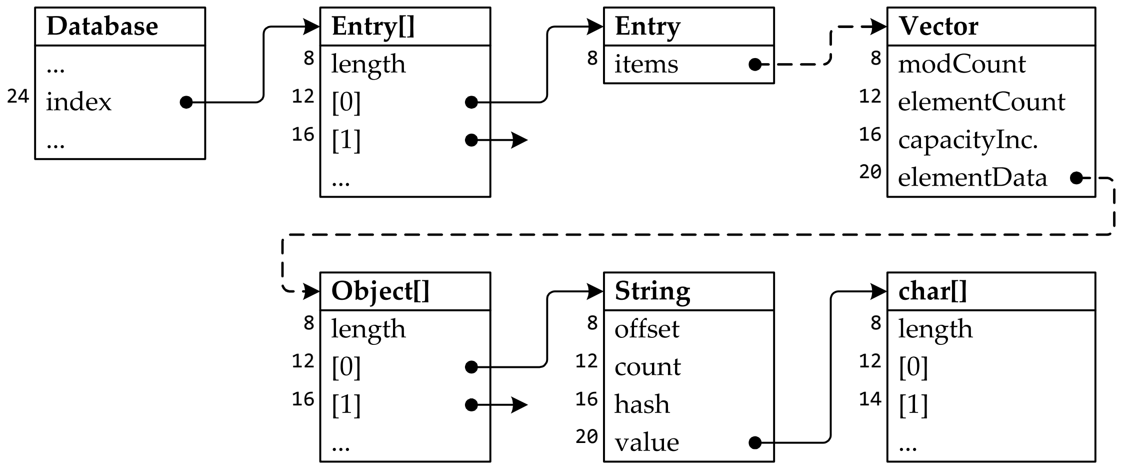


Figure 8.4: Object graph of benchmark db

Table 8.5 shows the field and array access statistics for this benchmark. Two fields can be inlined: the object field `Entry.items` and the array field `Vector.elementData`. Dynamic array inlining is necessary because the `Object[]` array of the `Vector` can be replaced by a larger copy when new elements are added. However, the initial capacity is sufficient for all database entries. Therefore, only the co-allocated array is accessed and the slow path for the optimized array access is never executed.

Field or Array	n*1000	%	Σ	Kind	State
Entry[]	66,564	19%	19%	Array element	Colocated
Database.index	61,060	17%	36%	Array field	Colocated
Vector.elementData	54,943	15%	51%	Array field	Inlined, dynamic length
Object[]	54,663	15%	66%	Array element	Colocated
Entry.items	50,947	14%	80%	Object field	Inlined
String.value	45,648	13%	93%	Array field	Colocated
Vector$1.this$0	23,126	6%	99%	Object field	Not optimized

Table 8.5: Access statistics for benchmark db

It is not possible to inline the field `Database.index` because it can be `null`. The `null` value is used internally as a marker that no sorted database index is available, i.e. the field is set to `null` whenever the database is modified. Our array inlining does not support `null` values because a `null` check before an optimized field load would be as expensive as the field load itself. The field `String.value` cannot be optimized as described in Section 7.6.3 on page 97.

The class `Vector$1` is an anonymous inner class of `Vector` that implements the `Enumeration` interface and is used to iterate over all elements. Its synthetic field `Vector$1.this$0` provides the back link from the iterator to the iterated `Vector`. Object inlining is not possible because multiple iterator objects are allocated for the same `Vector`. Furthermore, object colocation is not beneficial because iterator objects have a short lifetime. Because these arguments hold for most synthetic fields, we exclude them from the analysis and do not even emit read barriers for them.

Short-living temporary objects are the main optimization target for escape analysis. It eliminates the allocation of an object that does not escape its allocating method and replaces its fields by local variables, which removes all field accesses to this object. For example, the escape analysis algorithm of Kotzmann for the Java HotSpot client compiler [Kotzmann05a] optimizes the `Vector` iterator and therefore eliminates all accesses of the field `Vector$1.this$0`.

8.2.3 compress

The compress benchmark uses the well-known LZW algorithm [Welch84] for compression and decompression of several input files. The algorithm uses large arrays for the input and output data as well as for several temporary tables. Figure 8.5 shows the data structure used for compression. The decompression uses similar objects. In contrast to the benchmarks of the previous sections, only one instance of this graph is alive at the same time.

The field and array access statistics in Table 8.6 do not differentiate between compression and decompression because both parts are performed alternately. Most of the fields can be inlined, so the whole data structure is combined to one large object group. Only the actual data arrays for the input and output data are not inlinable because they are allocated much earlier than the other objects. The three arrays that are internally used by the algorithm have a fixed length. Because they are large, object colocation does not improve the cache behavior. Inlining eliminates all object field loads and nearly two thirds of array field loads. Additionally, the bounds checks can be simplified using the constant array length. The whole object group with an overall size of 414,240 bytes is allocated at once.

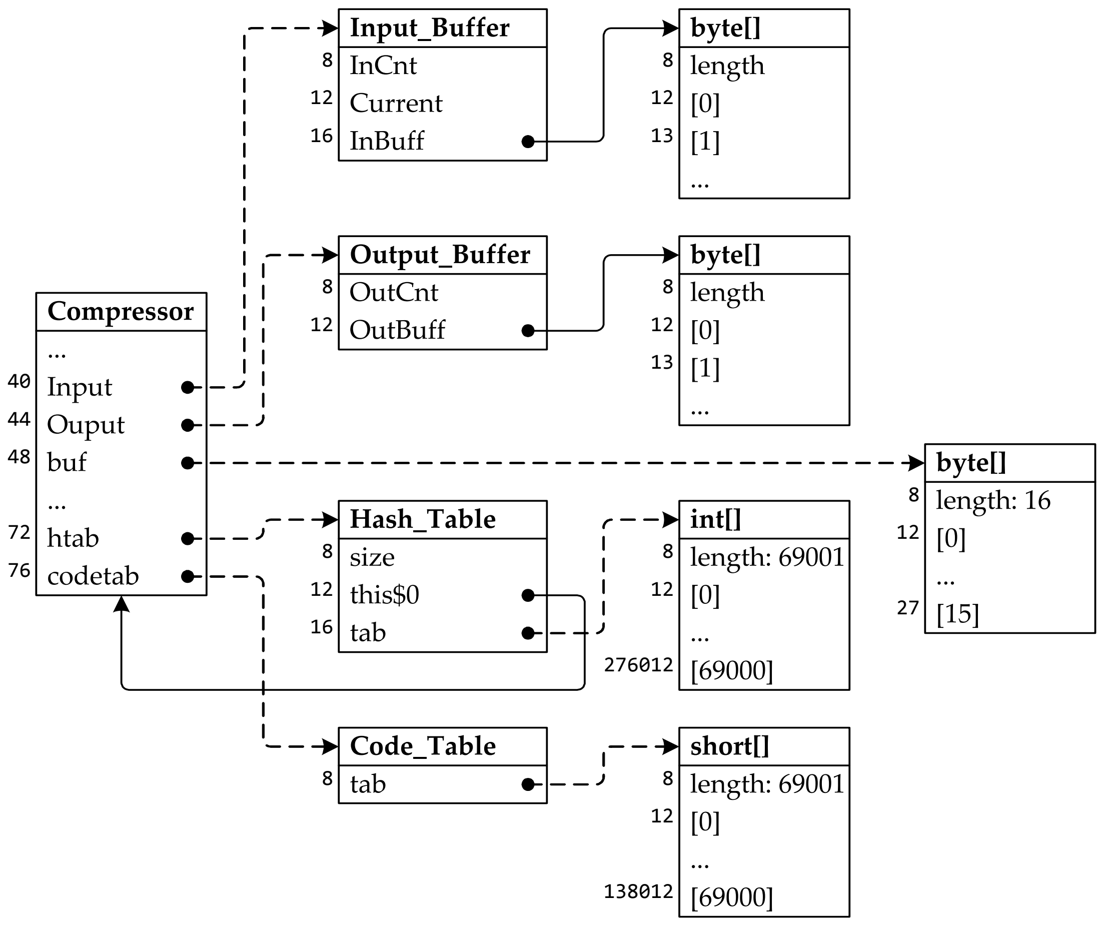


Figure 8.5: Object graph of benchmark compress

Field	n*1000	%	Σ	Kind	State
Code_Table.tab	92,339	12%	12%	Array field	Inlined, length 69001
Hash_Table.tab	66,369	8%	20%	Array field	Inlined, length 69001
Input_Buffer.InBuff	65,616	8%	28%	Array field	Colocated
Output_Buffer.OutBuff	65,616	8%	37%	Array field	Colocated
Comp_Base.Output	65,614	8%	45%	Object field	Inlined, subclasses
Compressor.htab	56,019	7%	52%	Object field	Inlined
Comp_Base.Input	48,286	6%	58%	Object field	Inlined, subclasses
Decompressor.de_stack	47,053	6%	64%	Object field	Inlined
De_Stack.tab	47,053	6%	70%	Array field	Inlined, length 8000
Suffix_Table.tab	47,046	6%	76%	Array field	Inlined, length 65536
Decompressor.tab_suffix	47,039	6%	82%	Object field	Inlined
Compressor.codetab	46,157	6%	88%	Object field	Inlined
Decompressor.tab_prefix	46,144	6%	94%	Object field	Inlined
Comp_Base.buf	20,956	3%	97%	Array field	Inlined, length 16, subcl.
Compress.rmask	19,723	2%	99%	Array field	Not optimized
Compress.lmask	9,862	1%	99%	Array field	Not optimized

Table 8.6: Access statistics for benchmark compress

The handling of the input and output data is equal for compression and decompression. Therefore, the classes `Compressor` and `Decompressor` have the same superclass `Comp_Base`, which defines the fields `Input`, `Output`, and `buf`. These three fields can only be inlined using the reverse object order, otherwise the offsets of the child objects would be different during compression and decompression. A `Compressor` object has a size of 80 bytes, while a `Decompressor` object has a size of 72 bytes, i.e. the offsets would differ by 8 bytes.

The fields `rmask` and `lmask` are static fields defined in the class `Compress`. It is not possible to colocate or to inline such fields because the class descriptor that contains the fields is located in the permanent generation, while the actual arrays are placed in the young or old generation. Therefore, we exclude such fields from optimization and do not emit read barriers for them.

8.2.4 jess

The jess benchmark is an early version of the Java Expert Shell System [Jess08], which applies rules to a set of data to solve several puzzles. Figure 8.6 shows a simplified version of the data model. Some of the classes have subclasses, and `Value` objects can reference arbitrary other objects.

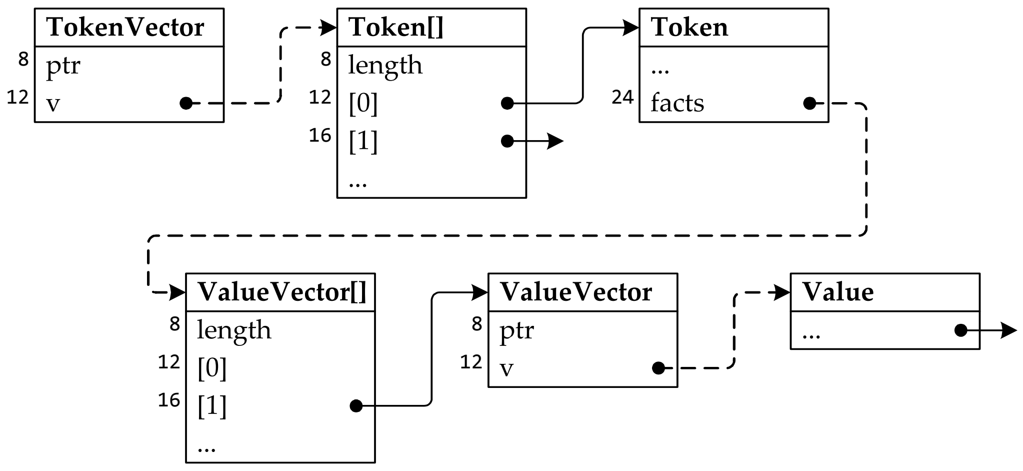


Figure 8.6: Object graph of benchmark jess

Table 8.7 shows the field access statistics. All three array fields shown in the object graph can be inlined. Dynamic array inlining is necessary because the arrays are used to manage dynamic lists of elements, similar to the collection classes. The most frequently accessed field `ValueVector.v` can only be optimized with reverse object order because the class `ValueVector` has a subclass.

During the solving process, the lists of tokens and values are frequently changed. Several arrays must be replaced by larger copies, i.e. the change rate of the array fields is high. Therefore, the slow path of the optimized array loads is taken for 13% of the loads of `ValueVector.v` and for 25% of the loads of `TokenVector.v`. The additional overhead of the slow path countervails the benefit of object inlining, so there is no speedup for this benchmark. Additionally, the successful inlining of the field `Hashtable.table` does not lead to optimized array loads because the array length is explicitly accessed before the array load (see Section 7.6.2 on page 96).

Field or Array	n*1000	%	Σ	Kind	State
ValueVector.v	39,024	22%	22%	Array field	Inlined, dyn. length, subclasses
Value[]	36,400	21%	43%	Array element	Colocated
ValueVector[]	26,339	15%	58%	Array element	Colocated
Token.facts	25,673	15%	73%	Array field	Inlined, dynamic length
TokenVector.v	5,865	3%	77%	Array field	Inlined, dynamic length
Token[]	5,840	3%	80%	Array element	Colocated
Node2.tests	5,025	3%	83%	Array field	Colocated
Object[]	4,517	3%	86%	Array element	Colocated
Hashtable$Entry[]	3,515	2%	88%	Array element	Colocated
Hashtable.table	3,515	2%	90%	Array field	Inlined, dyn. length, subclasses

Table 8.7: Access statistics for benchmark jess

8.3 The DaCapo Benchmarks

The DaCapo benchmark suite [Blackburn06] consists of eleven object-oriented applications. They are more elaborate than the SPECjvm98 benchmarks regarding code complexity, class structures, and class hierarchies. We evaluate these benchmarks similarly to the SPECjvm98 benchmarks and report the same metrics, i.e. the speedup of the different configurations as well as static and dynamic field access counts.

8.3.1 Impact on Run Time

Figure 8.7 shows the results for the DaCapo benchmarks. We executed each benchmark five times and show the slowest and the fastest runs on top of each other. The slowest run is always the first one and contains the compilation time. Because of the higher complexity of the benchmarks, more methods must be compiled in the startup phase. Therefore, the difference between the slowest and fastest runs is usually higher than for SPECjvm98.

The complexity also reduces the impact of our optimizations. The benchmarks require a larger heap size, which increases the overall time spent in the garbage collector. Our object colocation algorithm is embedded into the garbage collector and introduces a small but measurable overhead. This overhead outweighs the benefit for some of the benchmarks. For example, the benchmark antlr is slightly slower with object colocation, but then benefits from object and array inlining. For some other benchmarks such as fop and hsqldb, the overhead is higher than the benefit for all configurations. However, no benchmark has a slowdown of more than 2%.

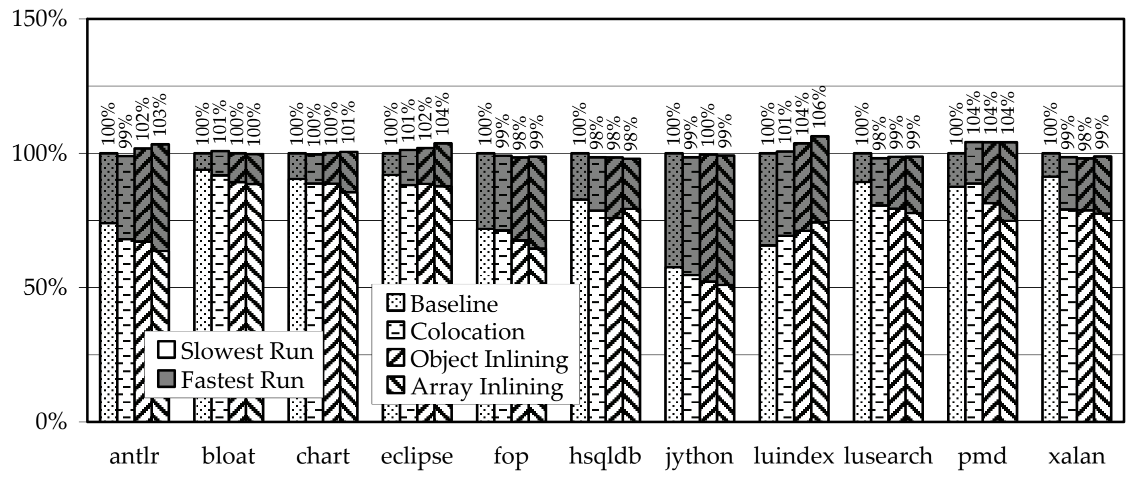


Figure 8.7: Speedup compared to baseline for DaCapo (taller bars are better)

8.3.2 Field Access Counts

The lower speedup of the DaCapo benchmarks is explained by the dynamic field access counts shown in Figure 8.8. Only a small percentage of the dynamic object field and array field accesses can be eliminated. One exception is the benchmark luindex: about two thirds of the array field loads are eliminated. Other benchmarks with a significant number of eliminated field loads are chart, fop, lusearch, and pmd.

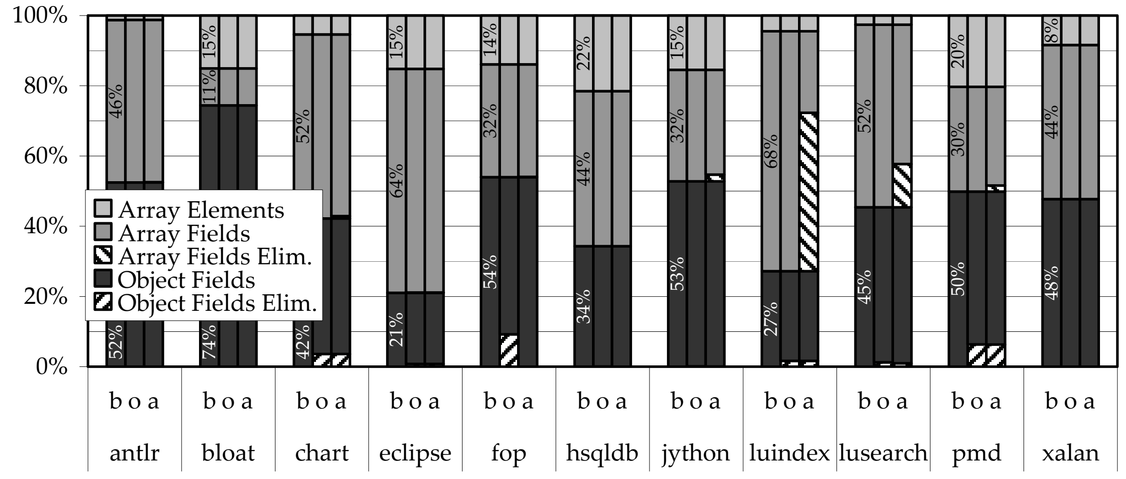


Figure 8.8: Dynamic field access statistics for DaCapo

The dynamic field access statistics reflect only one of the optimizations that are performed for inlined fields. Other benefits such as the elimination of type checks, the improved method inlining, and the simplification of array bounds checks also affect the benchmark speed. For example, the benchmark antlr shows a speedup although no significant number of field loads is eliminated.

8.3.3 Number of Optimized Fields

The higher complexity of the DaCapo benchmarks compared to SPECjvm98 is also visible in the static field access statistics shown in Table 8.8. The effective detection of hot fields using read barriers is essential to keep the run-time overhead of object colocation and inlining in an acceptable range.

	Initial			**Counted**		**Colocated**		**Inlined**	
	Obj.	**Arr.**	**Scalar**	**Obj.**	**Arr.**	**Obj.**	**Arr.**	**Obj.**	**Arr.**
antlr	795	126	606	238	43	24	13	1	2
bloat	1008	167	578	251	62	47	12	8	2
chart	1592	274	1296	207	68	32	11	4	2
eclipse	3079	767	2162	771	374	75	65	11	5
fop	1452	158	949	324	48	31	12	1	3
hsqldb	844	209	689	209	92	26	9	2	1
jython	1264	234	841	278	94	77	31	3	7
luindex	709	155	643	191	63	35	19	6	4
lusearch	684	140	623	148	55	22	15	3	4
pmd	1092	173	779	238	58	65	26	9	8
xalan	1335	183	852	449	75	13	11	0	1

Table 8.8: Static field access statistics for DaCapo

The several thousand object fields and several hundred array fields are reduced to at most 77 colocated object fields and 65 colocated array fields. However, our inlining algorithm is too conservative, so only few of the colocated fields can be inlined. The main constraint is object co-allocation where the time frame between the allocation of parent and children is too long, i.e. the allocations are not in the same compiled method so that co-allocation is not possible.

8.3.4 Compile-Time Impact

Table 8.9 shows the number of compiled methods for the different configurations. The recompilations necessary for the removal of read barriers, which are included in the column colocation, increase the number of compiled methods significantly. In contrast, the number of additionally compiled methods for object inlining and array inlining is low.

	Baseline	Colocation		Object Inlining		Array Inlining	
antlr	761	1164	+53%	1263	+66%	1283	+69%
bloat	646	901	+39%	996	+54%	1016	+57%
chart	496	660	+33%	702	+42%	729	+47%
eclipse	2563	3982	+55%	4147	+62%	4287	+67%
fop	593	978	+65%	1054	+78%	1109	+87%
hsqldb	444	744	+68%	804	+81%	819	+84%
jython	1030	1446	+40%	1519	+47%	1659	+61%
luindex	517	727	+41%	788	+52%	821	+59%
lusearch	383	615	+61%	656	+71%	648	+69%
pmd	813	1121	+38%	1242	+53%	1296	+59%
xalan	1094	1833	+68%	1842	+68%	1883	+72%

Table 8.9: Number of compiled methods for DaCapo

8.4 SPECjbb2005

The benchmark SPECjbb2005 [Spec05] emulates a client/server application and reports the executed number of transactions per second. It represents a typical three-tier business application with a database layer, a business logic layer, and a user interface. The focus is on the business logic. The database layer and the user interface are simulated by Java classes. For example, database tables are replaced by Java collections. A textual user interface is simulated by a two-dimensional character array that is filled with tabular results of the executed transactions. To benchmark the scalability of a Java VM, the number of transaction threads is variable. In our configuration, the thread count is incremented from 1 to 8.

Though some field loads can be eliminated, we fail to optimize the most important fields of SPECjbb2005. The business objects do not use Java collections themselves. Instead, either a `HashMap` or a `TreeMap` are wrapped by a `MapDataStorage` object. The field `MapDataStorage.data` cannot be inlined because it can reference either a `HashMap` or a `TreeMap` object, but we require that inlined fields reference only objects of a single class. Furthermore, the tree-based structure of the `TreeMap` cannot be optimized by object inlining because the `left` and `right` pointer of a tree node can be `null`. For all configurations of object colocation and inlining, the same number of executed transactions per second is reported, i.e. there is neither a speedup nor a slowdown.

8.5 SciMark

SciMark 2.0 [Pozo99] is a benchmark suite for scientific and numerical computing. It executes and measures five computational kernels and reports a score in Mflops. All kernels perform a large number of floating point operations in a low number of long-running and small loops. No objects are allocated in the benchmark runs, so no garbage collection is necessary. The data is stored in large scalar arrays. Because no object fields and array fields are accessed, no speedup can be expected from our optimizations. However, there should also be no slowdown due to the analysis overhead. We verified this by executing SciMark in all configurations mentioned above. All configurations showed the same results as the baseline.

8.6 Java Grande Benchmarks

The Java Grande benchmark suite [Bull00] consists of large-scale applications for numerical computing. Similarly to SciMark, most of these applications operate on large scalar arrays. We use only Section II and Section III of the sequential benchmarks. Section I consists of micro-benchmarks for primitive mathematical operations and is therefore unsuitable for our optimizations. Object and array inlining lead to no speedup, but have no negative impact either.

The only exception is the benchmark RayTracer, which renders a three-dimensional scene. The algorithm is similar to the mtrt benchmark of SPECjvm98, however the RayTracer benchmark has a much simpler internal structure than mtrt. Among other fields, our algorithms inlines the two important fields `Sphere.v` and `Sphere.c`. This eliminates about 60% of all object field loads (40% of all reference loads) and leads to a speedup of 33%.

Chapter 9

Related Work

This chapter discusses other research projects that work on similar optimizations. The focus is on related work for object inlining and for object colocation. So far, object inlining was only performed in static compilers, while object colocation was usually integrated into virtual machines. The chapter also presents some related work on dynamic profiling.

The success of programming languages that are based on a virtual machine and a garbage collector led to a large number of research projects that use this infrastructure for optimizations of the memory system. For example, various heuristics were proposed that guide the garbage collector. As a result, fields that are frequently accessed together end up in the same cache line (see Section 9.2).

In contrast to that, all previous work on object inlining was performed in static compilers, even if the target language was Java, which lends itself to dynamic optimizations in the virtual machine (see Section 9.1). We give an overview of existing approaches for object inlining and then focus on the algorithm of Dolby, which is cited most often in the context of object inlining.

Escape analysis [Blanchet03, Choi03, Kotzmann05b] is another optimization that reduces the overhead of memory accesses. It detects objects that do not escape a certain method (in which case the object fields are replaced by local variables) and objects that do not escape a certain thread (in which case they are allocated on the method stack). Escape analysis is orthogonal to object inlining because it optimizes short-living objects, whereas object inlining optimizes long-living data structures.

9.1 Object Inlining

Dolby et al. extended a static compiler for ICC++, a dialect of C++, with an algorithm for automatic object inlining [Dolby97, Dolby98, Dolby00]. The compiler provides a highly sophisticated interprocedural analysis framework. Their algorithm clones the code of methods that access optimized fields. Therefore, there can be both optimized and unoptimized objects of the same class as long as the same method always works on objects of the same kind. In order to inline a field, they analyze and modify not only all methods that assign the field, but also all methods that load the field. This is necessary because they eliminate object headers and pointers to inlined objects. The details of their analysis are presented in Section 9.1.2.

Because they use an advanced global data flow analysis, they are able to convert arrays of references to arrays of object values. Our analysis is not capable of performing such transformations. However, we can handle dynamic arrays where an array field can be changed to point to arrays of different sizes. This is not possible in any of the existing static approaches. The optimization of such dynamic data structures is a significant advantage of a run-time analysis, in addition to the benefit that no complex and time-consuming data flow analysis is necessary.

The time for their global data flow analysis varies from one quarter to half of the total compilation time. An analysis time of about 30 minutes is reported for an application with about 30,000 lines of code measured using a Pentium Pro system with 266 MHz [Dolby98]. This is several orders of magnitude too slow for being used in a just-in-time compiler. The speedup highly depends on the structure of the application. While some small benchmarks execute up to three times faster [Dolby97], a collection of medium-sized applications shows a maximum speedup of 50% [Dolby98].

Laud implemented object inlining in a static Java compiler [Laud01] that is based on the CoSy compiler construction framework [Alt94]. His algorithm can detect and handle the case when a child object is replaced with a new object. Instead of replacing the inlined field with a reference to the new object, the fields of the new object are assigned to the fields of the old object, i.e. a deep copy is performed. It is, however, not allowed that a child object is referenced by anything else than its parent object, e.g. by a field of another object. The algorithm can handle fields that are assigned multiple times, so it could be used for the inlining of dynamic arrays. However, no details regarding arrays are published. To the best of our knowledge, only the detection of inlinable fields was implemented, but not the necessary program transformations that remove loads of inlined fields. Therefore, no benchmark results are available.

Lhoták et al. provide a good introduction to object inlining and analyze the possible impact on several Java benchmarks [Lhoták05]. Depending on the access pattern, they distinguish four classes of inlinable fields, and use this classification to compare the number of fields that can be optimized by the algorithms of Dolby and Laud (see Section 9.1.1).

They focus on object fields in their study; array fields are not thoroughly evaluated. While they present the access counts of array fields and compare them with the number of object fields, their subsequent analysis and listing of inlinable fields does not cover array fields. Furthermore, they do not distinguish between arrays with constant length and arrays with variable length. Inlining of array elements is not evaluated. The study does not describe an analysis or implementation for object inlining, so no benchmark results are published, except for three hand-optimized benchmarks.

Veldema et al. present an algorithm for *object combining* that groups objects with the same lifetime [Veldema05]. Their optimizations are integrated into Manta, a static compiler for Java [Veldema01]. Object combining is more aggressive than object inlining because it also optimizes unrelated objects if they have the same lifetime. This allows the garbage collector to free multiple objects at once. Elimination of pointer accesses is performed separately by the compiler. Similar to our approach, they retain the object headers of child objects, which contain the virtual method table and a flags field. Inlining of a single variable-length array per object group is possible. Their optimizations focus on reducing the overhead of memory allocation and deallocation. This is beneficial for their system because it uses a mark-and-sweep garbage collector where these costs are high. They report a speedup of up to 34% for a set of object-oriented applications.

Ghemawat et al. use a cheap interprocedural analysis for object inlining and for other global optimizations [Ghemawat00]. Their analysis is integrated into Swift [Scales00], an optimizing static Java compiler for the Alpha architecture. They collect a variety of properties for each field, e.g. whether the field is never `null` or whether it is always assigned an object of the same type. Objects can be inlined either with their header or without their header. The header is necessary when the child object can be referenced from outside the parent. Arrays are only inlined if the length is a compile-time constant. Arrays with variable size are not optimized. There are no timing results with only object inlining enabled, so it is not possible to quantify the impact of object inlining.

Budimlić et al. present a static Java bytecode optimizer that performs object and array inlining [Budimlić97, Budimlić98]. According to our definitions, they mix the terms object inlining and escape analysis. When they *inline* an object, they eliminate the allocation and replace the fields by local variables, i.e. they perform scalar replacement of objects. For arrays, they replace an array of references by separate arrays of scalar values, one for each field of the inlining child. This is consistent with our definition of inlining because it performs some sort of address arithmetic to combine an array access and a field load into a single array access. Their algorithm requires that all array elements are initialized at the time the array is allocated and that all elements are newly allocated objects of the same class. It is not necessary that the length of the array is constant. The evaluation is limited to four small mathematical computations where their optimization is highly effective, leading to a speedup of up to 460%.

9.1.1 Classification of Inlinable Fields

The study of Lhoták et al. [Lhoták05] analyzes the field access behavior of several Java benchmarks, including SPECjvm98, to evaluate the possible impact of object inlining. They use the Soot framework [Vallée-Rai99] for bytecode instrumentation. The instrumented benchmarks emit traces of all field and array accesses. From these traces, they compute the following four predicates for each field `f`:

1. *[contains-unique]*: The field `f` of a parent object references the same child object throughout its lifetime.
2. *[unique-container-same-field]*: An object assigned to the field `f` of one object is never assigned to the same field `f` of another object.
3. *[unique-container-different-field]*: An object assigned to the field `f` of one object is never assigned to any other field of any other object.
4. *[not-globally-reachable]*: An object assigned to the field `f` of one object is never stored in a static field or an array. Together, the predicates 3 and 4 ensure that the object is not reachable by anything else than the field `f`.

Based on these predicates, they classify the field in one of the following categories. The categories determine whether the field is inlinable by the algorithm of Dolby, the algorithm of Laud, or by both algorithms. Figure 9.1 shows an example for each category.

- *Simply one-to-one field*: If all four predicates are satisfied for a field, both approaches can optimize the field.
- *Field-specific one-to-one field*: If only the predicates 1 and 2 are satisfied, the field can be inlined by the algorithm of Dolby, but not by the algorithm of Laud. The child object is referenced by different fields of different objects.
- *Unique-store field*: If only the predicates 2, 3, and 4 are satisfied, the field can be inlined by the algorithm of Laud, but not by the algorithm of Dolby. The field of the same parent object is assigned multiple times.
- *Non-inlinable fields*: If any other combination of predicates is satisfied, the field is not inlinable by any of the two algorithms.

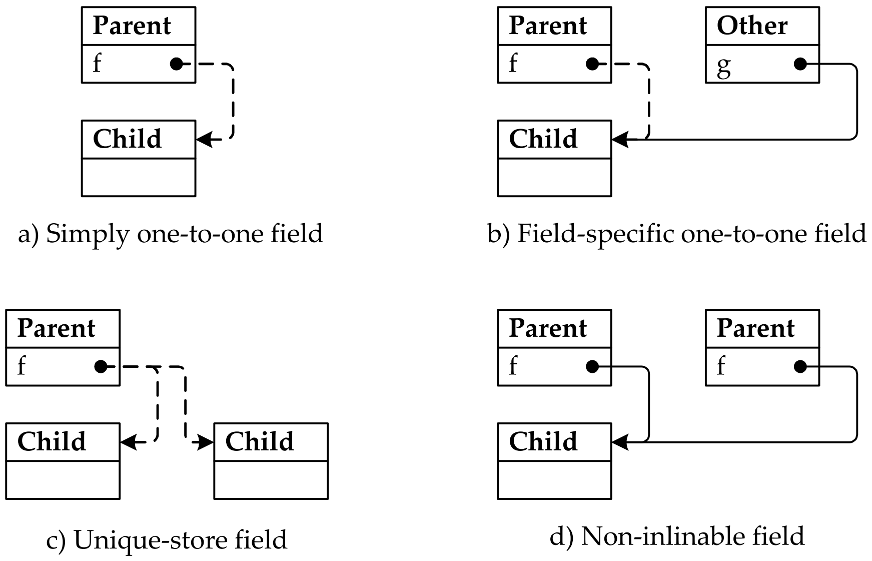


Figure 9.1: Examples for field categories

Based on this field classification, the study summarizes the number of field loads that can be eliminated by an algorithm that inlines fields of a certain category. Their main findings are that 1) a low number of fields account for a high percentage of field loads at run time, 2) most object fields are either *simply one-to-one* fields or not inlinable at all, and that 3) many array fields are *unique-store* fields. In addition to the summarized field access counts reported in [Lhoták05], the appendix of [Lhoták02] lists the detailed access counts for each field.

The study already states that the reported numbers may be overly optimistic. For example, they do not analyze whether the length of an array referenced by an array field is a compile-time constant, which could be required by an inlining algorithm. Additionally, we identified the following deficiencies:

- They neglect the problem of possible `null` values. If a field is not initialized in all code paths, explicit `null` checks are necessary before the field is accessed. Such checks are not possible for inlined fields, so the field is not inlinable. For example, they report the field `FieldDefinition.nextMatch` of the javac benchmark as inlinable, which is not correct.
- They do not analyze whether an inlined object needs an object header, e.g. whether synchronization is performed on it.
- They focus only on the number of eliminated field loads, but do not analyze other benefits of object inlining such as the increased amount of static type information for the compiler or the simplification of array bounds checks.

Our object inlining algorithm is conceptually similar to the algorithm of Dolby. We require that the same predicates are satisfied for a field to be inlinable, i.e. the predicates *[contains-unique]* and *[unique-container-same-field]*. These two predicates match our two preconditions for inlinable fields (see Section 3.3.2 on page 32).

For array fields, our approach for dynamic array inlining (see Section 7.3 on page 85) weakens the preconditions. It is not necessary that an inlined field is assigned only once, i.e. that the predicate *[contains-unique]* is satisfied. In contrast to the algorithm of Laud, we allow other fields to reference an inlined array, therefore we do not require the predicates *[unique-container-different-field]* and *[not-globally-reachable]* to be true.

9.1.2 Inlining Algorithm of Dolby

The object inlining algorithm of Dolby et al. [Dolby97, Dolby98, Dolby00] and our algorithm require the same properties to be satisfied for object inlining: the parent object must reference the same child object throughout its lifetime. Differences arise from the implementation strategies: Our dynamic optimization at run time must be more conservative to keep the analysis time low, while their static algorithm can optimize all possible candidates. Additionally, we must deal with dynamic class loading, while they can apply a global analysis and need not support dynamic loading of code that was not known at compile time.

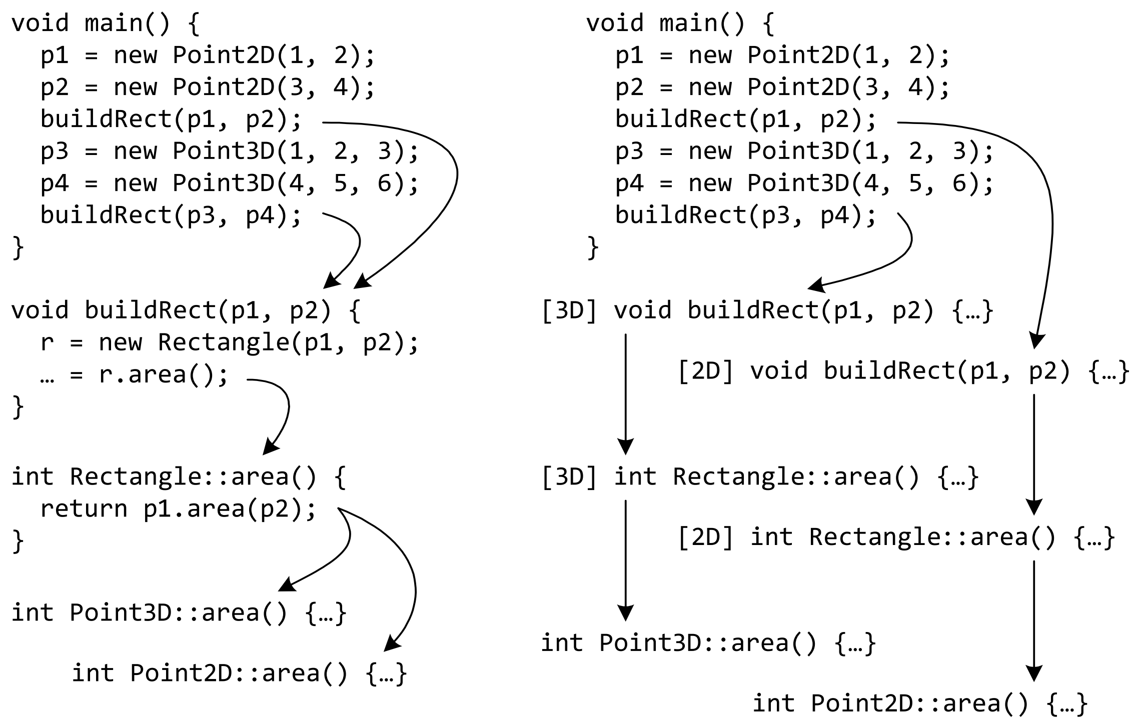
Their algorithm is integrated into the Concert compiler [Chien97]. The input language is ICC++, a dialect of C++ [Chien96]. It supports concurrency on the language level, i.e. the same source code can be compiled either to a serial or to a parallel version of the application. Some features of C++ such as value objects are missing in ICC++, therefore existing C++ applications and benchmarks must be slightly modified before they can be compiled.

The Concert compiler provides a global analysis framework for context-sensitive flow analysis. A *contour* represents the execution context of a method, e.g. the callers of the method with the run-time types of the parameters at the call site. When a method is called twice with parameters of different run-time types, the contour is split on demand so that each new contour has exact parameter types. The cloning framework of the compiler duplicates the code for the method so that the type information of the contours can be used for individual optimizations. It is also possible to clone a whole class. The following analyses and transformations are necessary for object inlining:

- *Assignment specialization*: All methods that assign the inlined field must be modified such that the assignment by reference is replaced by an assignment by value. The analysis must ensure that this copying is safe, i.e. that the original object is not accessed afterwards.
- *Use specialization*: All uses of the inlined field must be transformed such that the field access is replaced by address arithmetic. If the inlining child is passed to another method, this method must be rewritten to take the parent object as its parameter because the child object does not have a correct header.
- *Building fused classes*: Object inlining eliminates the header of the child object and the pointer from the parent to the child. This changes the layout of both the parent and the child. The layout algorithm tries to minimize the layout changes such that as few methods as possible must be duplicated.

Figure 9.2 shows an example for use specialization. Assume that the class `Rectangle` has two fields `p1` and `p2` of the declared type `Point2D`. The class `Point2D` holds the `x` and `y` coordinates of a point. The class `Point3D` is a subclass of `Point2D` and adds the `z` coordinate. The area calculation is performed differently in `Point2D.area()` and `Point3D.area()`. These methods are called from the method `Rectangle.area()` using dynamic binding. The method `buildRect()` allocates the `Rectangle` object. The two points are specified as parameters so that the method `main()` can construct both a two-dimensional and a three-dimensional rectangle.

When the fields `p1` and `p2` are inlined, it depends on the context whether the field `z` is needed for the points. The method `Rectangle.area()` can be called for an inlined object with or without this field, i.e. with objects of different sizes. The entire call graph must be cloned so that two specialized variants of the methods `buildRect()` and `Rectangle.area()` are available: one for two-dimensional rectangles and one for three-dimensional ones.



```
void main() {
  p1 = new Point2D(1, 2);
  p2 = new Point2D(3, 4);
  buildRect(p1, p2);
  p3 = new Point3D(1, 2, 3);
  p4 = new Point3D(4, 5, 6);
  buildRect(p3, p4);
}

void buildRect(p1, p2) {
  r = new Rectangle(p1, p2);
  … = r.area();
}

int Rectangle::area() {
  return p1.area(p2);
}

int Point3D::area() {…}

    int Point2D::area() {…}
```

a) Before method cloning

```
void main() {
  p1 = new Point2D(1, 2);
  p2 = new Point2D(3, 4);
  buildRect(p1, p2);
  p3 = new Point3D(1, 2, 3);
  p4 = new Point3D(4, 5, 6);
  buildRect(p3, p4);
}

[3D] void buildRect(p1, p2) {…}
         [2D] void buildRect(p1, p2) {…}

[3D] int Rectangle::area() {…}
         [2D] int Rectangle::area() {…}

int Point3D::area() {…}
              int Point2D::area() {…}
```

b) After method cloning

Figure 9.2: Example for use specialization with method cloning

Due to the flexible cloning framework, their algorithm can optimize all fields that satisfy the preconditions for object inlining. All methods that load or store the inlined field are analyzed and possibly cloned. In contrast, we analyze only methods that store the field. Methods that load the field and that are infrequently executed stay unmodified even after the optimization has finished. We require that all objects of a certain class have the same layout, so we cannot optimize the example shown above even though the preconditions would be satisfied. However, our dynamic array inlining can optimize more array fields than their algorithm. Our approach for dynamic array inlining requires run-time support of the garbage collector and thus cannot be implemented in a static compiler.

9.2 Improvement of Cache Behavior

History has shown that processor speed tends to grow faster than memory speed. Current processors spend a lot of time waiting for requested data to be loaded, especially in memory-intensive applications. Multi-level caches and memory prefetching strategies reduce this gap. The number of cache misses has a high impact on the overall application performance, so optimizations that improve the cache behavior are profitable.

Unfortunately, there is no efficient algorithm to compute an optimal memory layout with respect to a minimum number of cache misses [Petrank02]. Even if the exact sequence of memory accesses is known, finding the optimal solution has an exponential time complexity. Furthermore, it is not even possible to get close to the optimum, so it is not possible to evaluate how much of the theoretical benefit is obtained by a particular strategy.

Because of this constraint, all optimization algorithms for improving the cache behavior are based on heuristics. The usual strategy is to place data items that are accessed together next to each other in memory so that they can end up in the same cache line. When one data item is accessed, the others are automatically loaded into the cache and are thus available with a lower delay.

9.2.1 Object Colocation in the Garbage Collector

One possibility for improving the cache performance is to modify the order of objects on the heap. The object order can be influenced by garbage collection algorithms that move objects to new locations. Typically, such algorithms traverse the object graph and place all live objects next to each other in memory. The resulting object order depends on the order in which references are traversed. Hirzel evaluates ten different layouts and shows that no layout is best for a variety of benchmarks [Hirzel07]. The following three approaches are widely used in Java virtual machines. Details and examples are presented e.g. in [Siegwart06] and [Jones96].

- Breadth-first layout: The object graph is traversed breadth-first. This can be implemented without an additional data structure and is therefore simple and space-efficient, but leads to a mostly random object order.
- Depth-first layout: After an object was copied, the copying method is called recursively for the references inside this object. The object and its first child are colocated.
- Hierarchical layout: The breadth-first layout is performed independently for small memory blocks, e.g. the page size of the system. Related objects tend to end up colocated at least in the same page.

The normal copying order of these algorithms can be further refined by heuristics. When it is known that one field of a certain class is accessed frequently, the garbage collector can process this field first and place the referenced object next to its parent on the heap. The published algorithms differ mainly in the way how the most important fields are detected. For example, our object colocation algorithm described in Chapter 5 on page 47 uses profiling data collected by read barriers that increment field access counters.

Huang et al. describe a system called *online object reordering* [Huang04], implemented for the Jikes RVM. They use the adaptive compilation system of Jikes [Alpern00] that periodically records the currently executed methods. Fields accessed in the most frequently recorded methods are traversed first by the garbage collector. The decision which fields are hot is based on a static analysis of the methods. It is performed by the just-in-time compiler when compiling the methods before their first execution. The compiler distinguishes hot and cold blocks, e.g. based on the loop depth. All fields that are accessed in hot blocks are considered hot. This information is not as precise as our dynamic numbers obtained from the read barriers. By using the existing interrupts of Jikes, their analysis has a low run-time overhead of 2% to 3%.

Chilimbi et al. use generational garbage collection for *cache-conscious data placement* [Chilimbi98] and present results for the dynamically typed and purely object-oriented programming language Cecil [Chambers98]. They collect run-time information about accessed objects using a sequential buffer. When an object is accessed, its address is written into the next free position of the buffer. The granularity is at the object level and not at the field level, using the assumption that most objects are small so that it is not necessary to distinguish different fields within the same object. The object access information is converted to an object affinity graph before garbage collection. Objects that are accessed together with at most one other object access in between are added to the graph. The garbage collector places objects connected by an edge next to each other in memory. It is not necessary that the objects reference each other. They report a reduced execution time of 14% to 26% for their benchmarks.

Chen et al. use garbage collection as a proactive technique to improve the locality of objects [Chen06], i.e. they trigger garbage collection not only when the heap is full but also when the locality should be improved. They use read barriers to track all accessed objects. For the optimization of the cache behavior, the object references are inserted into a circular buffer similar to Chilimbi et al. For the optimization of page locality, objects are marked using a special bit in the object header when they are accessed. The instrumentation overhead is reduced by enabling the read barriers only during short sampling intervals. The implementation is integrated into the Common Language Runtime of Microsoft. The evaluation with several C# applications shows an average speedup of 17%, with an analysis overhead of less than 3%.

Shuf et al. improve the locality of objects in Java applications by co-allocating objects and then preserving this order during garbage collection [Shuf02b]. Their implementation is integrated into the Jikes RVM [Alpern00]. Instead of detecting frequently accessed fields, they use frequently instantiated types, called *prolific types* [Shuf02a], to guide the optimization. Similar to our co-allocation, they build parent-child relationships of objects. The allocation profiles that show the frequently instantiated classes are loaded from a file that was created by a prior profiling run of the application. The just-in-time compiler uses this information to co-allocate at most two objects if they have both a prolific type and if they are connected by a field. The garbage collector preserves this optimized order using a modified object traversal algorithm. When only co-allocation is performed, they report speedups of up to 21% with a non-copying mark-and-sweep collector where object allocation costs are high, but they observe no speedup with a copying collector. This corresponds with our experience that co-allocation itself is not beneficial if object allocations are cheap. For the configuration with a copying garbage collector, the average speedup of co-allocation combined with the modified garbage collector that preserves the object order is about 5%.

Calder et al. perform *cache-conscious data placement* not only for dynamically allocated objects, but also for global variables, constants, and the method stack [Calder98]. They optimize C, C++, and Fortran programs that do not use a garbage collector, so they cannot reorder objects. Instead, they modify the object allocation such that the initial position of a memory element is optimized. The profiling data is collected by a preceding profiling run of the instrumented application. Because they only simulate the resulting optimized data layout, they report only the impact on the cache miss rates and no speedups for the benchmarks.

Guyer et al. define *dynamic object colocation* in a different way than we do. They modify object allocation such that new objects that will be referenced by an object of the old generation are immediately allocated in the old generation and not in the nursery space [Guyer04]. The garbage collector is not modified. This colocation reduces garbage collection time and improves locality. They integrated the optimization into the Jikes RVM [Alpern00] and report a speedup of up to 10%.

9.2.2 Field Reordering and Object Splitting

A complementary approach to modifying the object order is modifying the objects themselves. If an object has multiple fields with different access frequencies, it is either possible to reorder the fields such that all frequently accessed ones are at the beginning of the object, or to split an object into a part with the frequently accessed fields and a second part with the infrequently accessed ones.

Chilimbi et al. evaluate both techniques, but they do it in different systems [Chilimbi99]. They perform structure splitting for structures of a size comparable to the cache line size. With splitting, the hot fields of multiple objects end up in the same cache line. They use profiling data collected by an instrumented version of the application to guide a static compiler for Java. Together with cache-conscious object colocation performed by the garbage collector, they report a speedup of 18% to 28%. Large objects with several hot fields cannot benefit from object splitting. Instead, it is beneficial to reorder the fields so that all hot fields are at the beginning of the object and therefore in the same cache line. They do not perform this optimization automatically, but implemented a tool that outputs reordering hints for the programmer.

Kistler et al. change the order of fields automatically at run time to improve the memory performance [Kistler00]. They collect profiling data and swap in a modified code image when the optimization was applied. This continuous program optimization is active for the whole application run [Kistler01]. They build a temporal relationship graph that contains the information how the fields of an object are accessed. The graph is then partitioned into aggregates that fit into one cache line. Inside these aggregates, the optimal field order is computed so that the load latency is minimized. The optimization is integrated into the Oberon system [Wirth92]. They report speedups of up to 96% for their set of Oberon applications. However, the overhead of continuous profiling is higher than the speedup in some cases.

Rubin et al. present a profiling framework for optimizations of the data layout [Rubin02]. They search the space of possible layouts using profiling data collected by an instrumented application. Instead of compiling and running the optimization candidates, they simulate the cache behavior on a representative trace of memory accesses. Simulation also yields the objects that are responsible for a poor performance, which is used to narrow the search space for the best layout. Using this framework, they perform field reordering and custom memory allocation. The experimental results show that the iterative search performed by the framework outperforms the existing single-pass heuristic optimizations.

Zhong et al. split structures and regroup arrays based on a model of *reference affinity* [Zhong04]. Reference affinity measures the relationship between memory accesses of a group of data in an execution trace, i.e. how many other memory accesses are performed between the optimization candidates. They use instrumented source code to collect the access profiles for a set of C applications, as well as a modified C compiler that can handle only type-safe C applications to perform structure splitting. They report an average speedup of about 10% for several tree-based benchmarks.

9.3 Dynamic Profiling Techniques

Many profiling techniques have been proposed to collect accurate profiling data with a low overhead [Arnold05]. Information can be obtained by monitoring run-time services of a virtual machine, by using hardware performance counters, by sampling the running application, by program instrumentation, or by a combination of these techniques. Most optimizations of the cache behavior described above are guided by profiling data.

Blackburn et al. evaluate the dynamic impact of read and write barriers on different platforms [Blackburn04]. They focus on current garbage collection algorithms that require barriers for correctness. Such barriers cannot be removed after a short measurement interval. The paper therefore reports the overhead that would arise by continuous profiling. For example, a complex conditional read barrier shows an average slowdown of 16% on a Pentium 4 processor, with a maximum slowdown of over 30% for certain benchmarks. Because such an overhead can hardly be amortized by any optimization, the overhead must be reduced by enabling profiling selectively or by removing the barriers when they are no longer needed.

Arnold et al. present a general sampling framework for reducing the costs of instrumented code [Arnold01]. The framework dynamically switches between the original uninstrumented code and the instrumented code in a fine-grained manner. This requires duplication of the code. A check at all method entries and backward branches switches between the two code versions. The tradeoff between accuracy and overhead can be adjusted at run time. They report an average overhead of 3% for profiles that overlap 93% to 98% with a perfect profile.

Hirzel et al. extend the above framework to collect profiling data on longer traces [Hirzel01]. In contrast to Arnold, they are not limited to intraprocedural and acyclic paths, but they collect information on paths that can span procedure boundaries and loops. To limit the overhead, they reduce the number of checks that switch between instrumented and uninstrumented code, while still guaranteeing that a check is performed in a bounded timeframe. They directly instrument machine code and report a run-time overhead of 3% to 18%.

Chapter 10

Summary

This chapter summarizes the basic principles of feedback-directed object inlining. It presents the main contributions and recapitulates the optimization phases, which are integrated into the just-in-time compiler and the garbage collector. Finally, the thesis is concluded with an outlook on future work that could increase the number of inlined fields.

In this thesis, we presented algorithms for object inlining and array inlining in a Java virtual machine. The project is part of a long-standing and ongoing collaboration between Sun Microsystems and the Institute for System Software at the Johannes Kepler University Linz. Because our work is integrated into Sun Microsystems' Java HotSpot virtual machine, several design decisions are influenced by existing VM subsystems such as the metadata model for classes and objects, the client compiler for just-in-time compilation, and the deoptimization framework for undoing optimistic optimizations.

10.1 Contributions

Although many feedback-directed optimizations for virtual machines have been proposed in literature, this area is still under active research because it offers novel and promising possibilities for optimization. To the best of our knowledge, our approach is the first that applies object inlining at run time in a virtual machine without requiring actions on the part of the programmer. Additionally, we are not aware of any system that allows inlining of array fields that are changed at run time. In this thesis, we contribute the following:

- We propose to integrate automatic object inlining and array inlining as a feedback-directed optimization into a Java virtual machine.
- We use just-in-time compilation and garbage collection for optimizations that cannot be performed by a static compiler.
- We use read barriers inserted by the just-in-time compiler to get a dynamic field access profile and remove read barriers when they are no longer needed.

- We perform object colocation in a system with dynamic class loading and generational garbage collection.
- We guarantee the preconditions for inlining by dynamically compiling and recompiling methods with co-allocation and field store guards, thus avoiding a global data flow analysis.
- We present an approach for inlining array fields that are assigned multiple times. It can optimize dynamic data structures such as collections.
- We handle class hierarchies by reversing the inlining order such that inlining offsets are constant even if a parent class has subclasses.
- We fully support Java's dynamic class loading by using the deoptimization framework of the Java HotSpot virtual machine.
- We build our implementation on a production-quality Java virtual machine that is highly tuned for performance.
- We evaluate our implementation and compare different configurations of object colocation, object inlining, and array inlining using several benchmark suites.

10.2 The Big Picture

Object inlining reduces the costs of field accesses by replacing memory accesses with address arithmetic. While the actual optimization that transforms field loads is quite simple and straightforward, the preceding analysis steps that detect inlinable fields and guarantee the necessary preconditions are challenging. We rely on algorithms that are mostly integrated into the just-in-time compiler and the garbage collector. Figure 10.1 summarizes the steps that are necessary for the inlining of a field. The flow chart shows the dependencies of the subsystems and the order in which methods are analyzed and compiled to guarantee the preconditions.

In order to find out which methods allocate objects and which methods store fields, the bytecodes of all methods must be analyzed before they are executed for the first time. This fills the method table—a hash table that maps class names and field names to methods that instantiate the class or store the field. The analysis of a method can be done at any time between loading the method's class and executing the method for the first time. To exclude methods that are loaded but never executed, we analyze them as late as possible, i.e. at link time.

All methods start being executed in the interpreter. Only frequently executed methods are scheduled for compilation. Our modified compiler inserts read barriers that count field accesses at run time. When a field counter exceeds a certain threshold, the field is considered important, added to the hot-field tables, and optimized. All successive steps are performed only for the few hot fields.

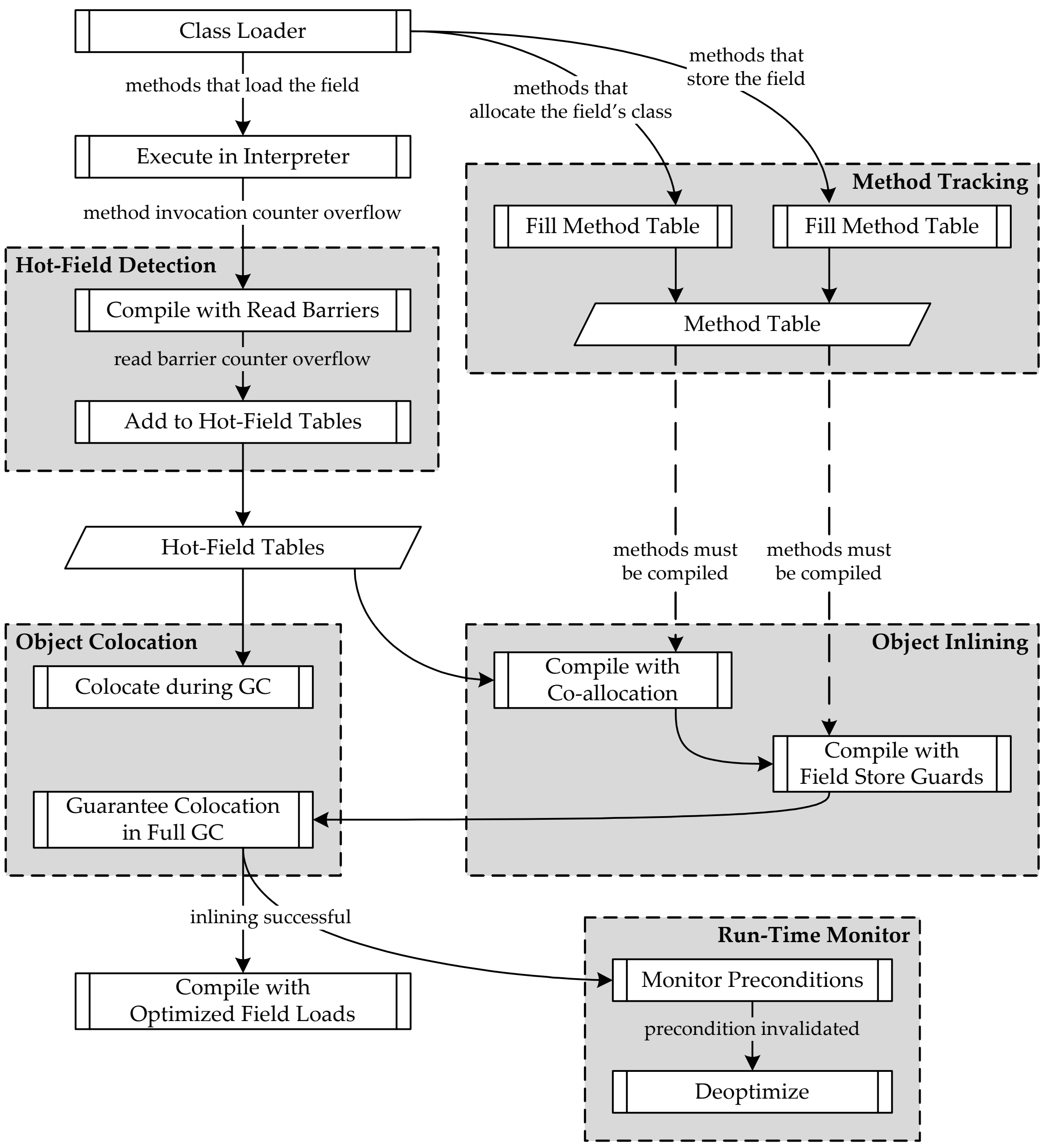


Figure 10.1: Phases of automatic object inlining

The hot-field tables are used by the garbage collector to perform object colocation, i.e. to place objects connected by hot fields next to each other in memory. This improves the spatial locality of objects and therefore the cache performance. It is a statistical optimization, i.e. it is desirable that a large number of objects of a certain class are colocated, but non-colocated objects do not affect correctness.

In contrast to that, object inlining demands that all objects of a certain class are colocated. To guarantee this, all methods that allocate parent objects as well as all methods that store inlined fields are analyzed. We combine the analysis and the necessary transformations of the methods by using the just-in-time compiler, i.e. we compile all such methods. The compiler reports feedback whether the transformation succeeded. The information which methods must be compiled is available from the method table. Methods that allocate parent objects are compiled with co-allocation, i.e. allocations of parent and child objects are merged into a single allocation. Methods that store the field are compiled with field store guards, i.e. field stores are preceded by calls of a runtime function that reverts object inlining.

When the compilation of the methods succeeds, the preconditions are satisfied for all objects allocated by the newly created machine code. However, it is still necessary to wait until no old machine code is executed anymore and until a run of the mark-and-compact garbage collector performed object colocation for the entire heap. After this, optimized field loads are possible.

A field load can be optimized by load folding (if the address of the child object is not needed) or by address computation (if the address of the child object is needed explicitly). Care must be taken to preserve an implicit `null` check that is integrated into the memory access. Additionally, the increased amount of static type information can be used for optimizations.

Array inlining, i.e. the optimization of fields that reference an array instead of another object, does not need additional analysis steps. Arrays can be handled mostly like objects. However, some differences must be considered. On the one hand, the size of an array is not a compile-time constant but depends on the actual array length. This restricts some optimizations, e.g. only one array with a variable length can be part of an object group. On the other hand, the array bounds check that precedes each array access can be used to optimize array fields that are stored multiple times. For example, this enables the optimization of collection classes.

Class hierarchies pose another problem for object inlining: If the class of a parent object has a subclass, an inlining child cannot be placed at a fixed offset after the parent because the parent’s size is different for the superclass and the subclass. This problem is solved by placing the child objects in front of the parent object on the heap, which leads to negative but constant offsets. However, the reverse order complicates array inlining and requires a copy of the array length at the end of the array. With this length field, it is possible to integrate the address arithmetic for the optimized array access into the array bounds check without inserting additional machine instructions.

Dynamic class loading as well as object allocations and field stores that are done via reflection, the Java Native Interface, or object cloning can violate a precondition at any time after the optimization was performed. Such cases are detected via run-time monitoring. They trigger deoptimization of all methods that contain optimized field loads so that these fields are loaded from memory again. However, object colocation in the garbage collector is still possible to improve the cache behavior.

Method execution, just-in-time compilation, and garbage collection are performed asynchronously, therefore the subsystems are not invoked sequentially. All phases mentioned above are partly overlapping, so they are properly synchronized and operate correctly even if new classes are loaded later on. This leads to a sometimes conservative design that misses possible optimizations, but is inevitable for a feedback-directed optimization that is performed at run time.

10.3 Future Work

Our extension of the Java HotSpot VM is a reliable implementation of object and array inlining. The benchmark results show that removing field loads is profitable for many applications, and that the overhead is reasonably small if no optimization is possible. The implementation is stable enough to execute the benchmarks, but should not be considered production-quality because some corner cases could lead to unexpected behavior. Additionally, we are conservative and do not optimize some cases where the expected benefit was too small compared to the implementation complexity. The following small-scale improvements could be implemented without changing the overall architecture of our system:

- Some algorithms handle complicated cases conservatively, mostly the algorithm for object co-allocation. Currently, we fail to co-allocate objects if the code between the allocation of the parent object and the allocation of the child object is too complex, e.g. if arbitrary methods are called in between. Improvements of co-allocation would directly lead to more inlinable fields. Additionally, we perform object colocation only for the first element of an array, even though colocating some or all array elements could be profitable.
- We do not optimize fields that are accessed using the dynamic features of Java, i.e. reflection, the Java Native Interface, and object cloning. It would be possible to handle these features less conservatively. Object cloning would be the simplest of the three because the semantics of `Object.clone()` are clearly defined, so this method could be replaced by specialized machine code created by the just-in-time compiler. Reflection would be more complicated but still in the scope of Java, while analyzing the impact of the Java Native Interface would require the analysis of arbitrary application-specific machine code.
- In order to guarantee the preconditions for object inlining and to remove unnecessary read barriers, many methods have to be compiled. A better heuristic for scheduling the compilations could reduce this number. For example, we compile a method several times when it is a precondition for several fields. It would be possible to perform all transformations in a single compilation.
- In general, inlining of array elements is not possible with our approach. However, some special cases could be handled, e.g. rectangular multi-dimensional arrays. This optimization would replace an array of arrays by a single large array. Such arrays are frequently used in scientific applications.

Some constraints result from the basic design of our approach. We do not support certain optimizations because they would introduce too much complexity that cannot be handled in our dynamic approach. A future implementation of object inlining could rely more on static analysis in favor of more optimizations.

For example, we do not allow optimized and unoptimized objects of the same class to coexist. This is the most severe restriction because a single allocation site where co-allocation is not possible prevents the inlining of a field in all objects. Especially library classes such as collections are frequently allocated. If instances of such classes were separated into disjoint groups, the optimization of a single group would be possible. This would require some sort of global analysis. While a global data flow analysis is complicated and expensive because of dynamic class loading, a limited form could be sufficient for this purpose.

Our approach eliminates neither the object headers of inlined objects nor the pointers to inlined fields. This is necessary because we use a dynamic optimization model that smoothly transitions between unoptimized and optimized machine code. This approach does not allow structural changes of the heap. To support such changes, explicit phases would have to be introduced. For example, removing a field from all objects of a certain class would require a single transition point at which the heap is rewritten.

After this point, it would not be allowed to access the field anymore, i.e. all methods that access the field would have to be rewritten at this point. This would require information about all places where the optimized field is loaded, in addition to our information where the field is stored. If deoptimization is necessary, the heap would have to be rewritten again to reintroduce the field. Future work could investigate whether the complexity of such phase changes is justified by the reduced memory consumption of the optimized application.

In summary, we think that a compromise between the static solutions of object inlining presented in the related work and our fully dynamic solution could lead to a higher number of optimized fields while still being suitable for a virtual machine.

10.4 Conclusions

From the perspective of software engineering, the comparison of Java with C or C++ shows the progress of programming languages over time. The features of Java [Gosling05] such as strong typing, portability, exception handling, language support for synchronization, garbage collection, and many more simplify application development. Additionally, the precise specification of the execution environment [Lindhom99] and the memory model [Manson05] are clear benefits of Java.

Java has been considered slow for a long time. The introduction of just-in-time compilation, the implementation of novel optimizations for these compilers, and the enhancement of the garbage collection algorithms have improved the performance significantly, so Java is used for a wide variety of desktop and server applications today.

After traditional compiler optimizations were successfully applied inside Java virtual machines, research shifted to optimizations that go beyond the possibilities of static compilers. Feedback-directed optimizations that build on profiling data collected at run time allow virtual machines to adapt to the actual workload of an application, while static compilers can optimize only for an average workload.

Instead of viewing just-in-time compilation and garbage collection as a run-time overhead stealing time that could have been spent executing the application, they should be considered as a powerful vehicle for dynamic optimizations. Our algorithm for inlining array fields that are modified at run time is one example of an optimization that is not possible in C or C++.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Agesen99] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, D. White: *An Efficient Meta-lock for Implementing Ubiquitous Synchronization*. In P*roceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,* pages 207–222. ACM Press, 1999. doi:10.1145/320384.320402

[Agesen00] Ole Agesen, David Detlefs: *Mixed-mode Bytecode Execution*. Technical Report TR-2000-87, Sun Microsystems Laboratories, 2000.

[Alpern00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, J. Whaley: *The Jalapeño Virtual Machine*. In *IBM Systems Journal*, volume 39, issue 1, pages 211–238. IBM Press, 2000.

[Alt94] Martin Alt, Uwe Aßmann, Hans van Someren: *Cosy Compiler Phase Embedding with the CoSy Compiler Model*. In *Proceedings of the International Conference on Compiler Construction*, LNCS 786, pages 278–293. Springer-Verlag, 1994. doi:10.1007/3-540-57877-3_19

[Anderson97] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, W. E. Weihl: *Continuous profiling: where have all the cycles gone?* In *ACM Transactions on Computer Systems*, volume 15, issue 4, pages 357–390. ACM Press, 1997. doi:10.1145/265924.265925

[Arnold01] Matthew Arnold, Barbara G. Ryder: *A Framework for Reducing the Cost of Instrumented Code*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179. ACM Press, 2001. doi:10.1145/378795.378832

[Arnold05] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, Peter F. Sweeney: *A Survey of Adaptive Optimization in Virtual Machines*. In *Proceedings of the IEEE*, volume 93, issue 2, pages 449–466. IEEE Computer Society, 2005. doi:10.1109/JPROC.2004.840305

[Bacon98] David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano: *Thin Locks: Featherweight Synchronization for Java*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268. ACM Press, 1998. doi:10.1145/277650.277734

[Bilardi03] Gianfranco Bilardi, Keshav Pingali: *Algorithms for Computing the Static Single Assignment Form*. In *Journal of the ACM*, volume 50, issue 3, pages 375–425. ACM Press, 2003. doi:10.1145/765568.765573

[Blackburn04] Stephen M. Blackburn, Antony L. Hosking: *Barriers: Friend or Foe?* In *Proceedings of the International Symposium on Memory Management*, pages 143–151. ACM Press, 2004. doi:10.1145/1029873.1029891

[Blackburn06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann: *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190. ACM Press, 2006. doi:10.1145/1167473.1167488

[Blanchet03] Bruno Blanchet: *Escape Analysis for Java: Theory and Practice*. In *ACM Transactions on Programming Languages and Systems*, volume 25, issue 6, pages 713–775. ACM Press, 2003. doi:10.1145/945885.945886

[Bodík00] Rastislav Bodík, Rajiv Gupta, Vivek Sarkar: *ABCD: Eliminating Array Bounds Checks on Demand*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333. ACM Press, 2000. doi:10.1145/349299.349342

[Briggs94] Preston Briggs, Keith D. Cooper, Linda Torczon: *Improvements to Graph Coloring Register Allocation*. In *ACM Transactions on Programming Languages and Systems*, volume 16, issue 3, pages 428–455. ACM Press, 1994. doi:10.1145/177492.177575

[Briggs97] Preston Briggs, Keith D. Cooper, L. Taylor Simpson: *Value Numbering*. In *Software: Practice and Experience*, volume 27, issue 6, pages 701–724. John Wiley & Sons, 1997. doi:10.1002/(SICI)1097-024X(199706)27:6<701::AID-SPE104>3.0.CO;2-0

[Budimlić97] Zoran Budimlić, Ken Kennedy: *Optimizing Java: Theory and Practice*. In *Concurrency: Practice and Experience*, volume 9, issue 6, pages 445–463. John Wiley & Sons, 1997. doi:10.1002/(SICI)1096-9128(199706)9:6<445::AID-CPE301>3.0.CO;2-L

[Budimlić98] Zoran Budimlić, Ken Kennedy: *Static Interprocedural Optimizations in Java*. Technical Report CRPC-TR98746, Center for Research on Parallel Computation, Rice University, 1998.

[Bull00] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, R. A. Davey: *A Benchmark Suite for High Performance Java*. In *Concurrency: Practice and Experience*, volume 12, issue 6, pages 375–388. John Wiley & Sons, 2000. doi:10.1002/1096-9128(200005)12:6<375::AID-CPE480>3.0.CO;2-M

[C1Visualizer] Christian Wimmer: *Java HotSpot™ Client Compiler Visualizer*, 2008. https://c1visualizer.dev.java.net/

[Calder98] Brad Calder, Chandra Krintz, Simmi John, Todd Austin: *Cache-Conscious Data Placement*. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149. ACM Press, 1998. doi:10.1145/291069.291036

[Chaitin81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, Peter W. Markstein: *Register Allocation via Coloring*. In *Computer Languages*, volume 6, issue 1, pages 47–57. Elsevier Science Ltd., 1981. doi:10.1016/0096-0551(81)90048-5

[Chambers91] Craig Chambers, David Ungar: *Making Pure Object-Oriented Languages Practical*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–15. ACM Press, 1991. doi:10.1145/117954.117955

[Chambers98] Craig Chambers: *The Cecil Language Specification and Rationale*, Version 3.0. Department of Computer Science and Engineering, University of Washington, 1998.

[Chen06] Wen-ke Chen, Sanjay Bhansali, Trishul M. Chilimbi, Xiaofeng Gao, Weihaw Chuang: *Profile-Guided Proactive Garbage Collection for Locality Optimization*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 332–340. ACM Press, 2006. doi:10.1145/1133981.1134021

[Cheney70] C. J. Cheney: *A Nonrecursive List Compacting Algorithm*. In *Communications of the ACM*, volume 13, issue 11, pages 677–678. ACM Press, 1970. doi:10.1145/362790.362798

[Chien96] Andrew A. Chien, Uday. S. Reddy, John Plevyak, Julian Dolby: *ICC++ - A C++ Dialect for High Performance Parallel Computing*. In *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software*, LNCS 1049, pages 76–95. Springer-Verlag 1996. doi:10.1007/3-540-60954-7_45

[Chien97] Andrew A. Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, Xingbin Zhang: *Supporting High Level Programming with High Performance: The Illinois Concert System*. In *Proceedings of the Internal Workshop on High-Level Programming Models and Supportive Environments*, pages 15–24. IEEE Computer Society, 1997. doi:10.1109/HIPS.1997.582952

[Chilimbi98] Trishul M. Chilimbi, James R. Larus: *Using Generational Garbage Collection to Implement Cache-Conscious Data Placement*. In *Proceedings of the International Symposium on Memory Management*, pages 37–48. ACM Press, 1998. doi:10.1145/286860.286865

[Chilimbi99] Trishul M. Chilimbi, Bob Davidson, James R. Larus: *Cache-Conscious Structure Definition*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24. ACM Press, 1999. doi:10.1145/301618.301635

[Choi03] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, Samuel P. Midkiff: *Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis*. In *ACM Transactions on Programming Languages and Systems*, volume 25, issue 6, pages 876–910. ACM Press, 2003. doi:10.1145/945885.945892

[Click95] Cliff Click, Michael Paleczny: *A Simple Graph-Based Intermediate Representation*. In *Papers from the ACM SIGPLAN Workshop on Intermediate Representations*, pages 35–49. ACM Press, 1995. doi:10.1145/202529.202534

[Click02] Cliff Click, John Rose: *Fast Subtype Checking in the HotSpot JVM*. In *Proceedings of the ACM-ISCOPE Conference on Java Grande*, pages 96–107. ACM Press, 2002. doi:10.1145/583810.583821

[Cramer97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R: Wilson, M. Wolczko: *Compiling Java Just in Time*. In *IEEE Micro*, volume 17, issue 3, pages 36–43. IEEE Computer Society, 1997. doi:10.1109/40.591653

[Cytron91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. In *ACM Transactions on Programming Languages and Systems*, volume 13, issue 4, pages 451–490. ACM Press, 1999. doi:10.1145/115372.115320

[Dean95] Jeffrey Dean, David Grove, Craig Chambers: *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 952, pages 77–101. Springer-Verlag, 1995.

[Detlefs99] David Detlefs, Ole Agesen: *Inlining of Virtual Methods*. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 1628, pages 258–277, Springer-Verlag, 1999.

[Dolby97] Julian Dolby: *Automatic Inline Allocation of Objects*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 7–17. ACM Press, 1997. doi:10.1145/258915.258918

[Dolby98] Julian Dolby, Andrew A. Chien: *An Evaluation of Automatic Object Inline Allocation Techniques*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–20. ACM Press, 1998. doi:10.1145/286936.286943

[Dolby00] Julian Dolby, Andrew A. Chien: *An Automatic Object Inlining Optimization and its Evaluation*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357. ACM Press, 2000. doi:10.1145/349299.349344

[Eclipse08] *Eclipsepedia – Rich Client Platform*, 2008. http://wiki.eclipse.org/Rich_Client_Platform

[Fink03] Stephen J. Fink, Feng Qian: *Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement*. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 241–252. IEEE Computer Society, 2003. doi:10.1109/CGO.2003.1191549

[Ghemawat00] Sanjay Ghemawat, Keith H. Randall, Daniel J. Scales: *Field Analysis: Getting Useful and Low-cost Interprocedural Information*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 334–344. ACM Press 2000. doi:10.1145/349299.349343

[Gosling05] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java™ Language Specification*, Third Edition. Addison-Wesley, 2005.

[Griesemer00] Robert Griesemer, Srdjan Mitrovic: *A Compiler for the Java HotSpot™ Virtual Machine*. In László Böszörményi, Jürg Gutknecht, Gustav Pomberger (editors): *The School of Niklaus Wirth: The Art of Simplicity*, pages 133–152. dpunkt.verlag, 2000.

[Guyer04] Samuel Z. Guyer, Kathryn S. McKinley: *Finding Your Cronies: Static Analysis for Dynamic Object Colocation*. In P*roceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 237–250. ACM Press, 2004. doi:10.1145/1028976.1028996

[Häubl07] Christian Häubl: *Optimized Strings for the Java HotSpot™ VM*. Master's thesis proposal, Institute for System Software, Johannes Kepler University Linz, 2007.

[Hegde07] Ravi Hegde: *Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers*. Intel® Software Network, 2007.

[Hirzel01] Martin Hirzel, Trishul M. Chilimbi: *Bursty Tracing: A Framework for Low-Overhead Temporal Profiling*. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[Hirzel07] Martin Hirzel: *Data Layouts for Object-Oriented Programs*. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 265–276. ACM Press, 2007. doi:10.1145/1254882.1254915

[Hölzle91] Urs Hölzle, Craig Chambers, David Ungar: *Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches*. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 512, pages 21–38. Springer-Verlag, 1991. doi:10.1007/BFb0057013

[Hölzle92] Urs Hölzle, Craig Chambers, David Ungar: *Debugging optimized code with dynamic deoptimization*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992. doi:10.1145/143095.143114

[Hölzle94] Urs Hölzle, David Ungar: *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994. doi:10.1145/178243.178478

[Hölzle96] Urs Hölzle, David Ungar: *Reconciling Responsiveness with Performance in Pure Object-Oriented Languages*. In *ACM Transactions on Programming Languages and Systems*, volume 18, issue 4, pages 355–400. ACM Press, 1996. doi:10.1145/233561.233562

[Hosking93] Antony L. Hosking, Richard L. Hudson: *Remembered Sets Can Also Play Cards*. In *Proceedings of the OOPSLA Workshop on Garbage Collection and Memory Management*. ACM Press, 1993.

[Huang04] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, Perry Cheng: *The Garbage Collection Advantage: Improving Program Locality*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80. ACM Press, 2004. doi:10.1145/1028976.1028983

[Intel07]	Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A and 2B: Instruction Set Reference*. Order numbers 253666 and 253667, 2007.
[ISO14882]	ISO/IEC: *C++*. International Standard ISO/IEC 14882, 2nd edition, 2003.
[ISO23270]	ISO/IEC: *C#*. International Standard ISO/IEC 23270, 2nd edition, 2006.
[ISO23271]	ISO/IEC: *Common Language Infrastructure (CLI)*. International Standard ISO/IEC 23271, 2nd edition, 2006.
[Jess08]	Ernest Friedman-Hill: *Jess, the Rule Engine for the Java™ Platform*, 2008. http://www.jessrules.com/
[Jones96]	Richard Jones, Rafael Lins: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.
[Kawachiya02]	Kiyokuni Kawachiya, Akira Koseki, Tamiya Onodera: *Lock Reservation: Java Locks can Mostly do without Atomic Operations*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–141. ACM Press, 2002. doi:10.1145/582419.582433
[Kawahito00]	Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani: *Effective Null Pointer Check Elimination Utilizing Hardware Trap*. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149. ACM Press, 2000. doi:10.1145/378993.379234
[Kistler00]	Thomas Kistler, Michael Franz: *Automated Data-Member Layout of Heap Objects to Improve Memory-Hierarchy Performance*. In *ACM Transactions on Programming Languages and Systems*, volume 22, issue 3, pages 490–505. ACM Press, 2000. doi:10.1145/353926.353937
[Kistler01]	Thomas Kistler, Michael Franz: *Continuous Program Optimization: Design and Evaluation*. In *IEEE Transactions on Computers*, volume 50, issue 6, pages 549–566. IEEE Computer Society, 2001. doi:10.1109/12.931893
[Kotzmann02]	Thomas Kotzmann: *Ein Just-in-Time-Compiler für Java*. Master's thesis, Institute for Practical Computer Science, Johannes Kepler University Linz, 2002.
[Kotzmann05a]	Thomas Kotzmann: *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. PhD thesis, Institute for System Software, Johannes Kepler University Linz, 2005.

[Kotzmann05b] Thomas Kotzmann, Hanspeter Mössenböck: *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 111–120. ACM Press, 2005. doi:10.1145/1064979.1064996

[Kotzmann07] Thomas Kotzmann, Hanspeter Mössenböck: *Run-Time Support for Optimizations Based on Escape Analysis*. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60. IEEE Computer Society, 2007. doi:10.1109/CGO.2007.34

[Kotzmann08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, David Cox: *Design of the Java HotSpot™ Client Compiler for Java 6*. In *ACM Transactions on Architecture and Code Optimization*. ACM Press, 2008.

[Laud01] Peeter Laud: *Analysis for Object Inlining in Java*. In *Proceedings of the JOSES Workshop*, 2001.

[Lhoták02] Ondřej Lhoták: *Run-time Evaluation of Object Inlining Opportunities in Java*. Technical Report SOCS-02.3, School of Computer Science, McGill University, 2002.

[Lhoták05] Ondřej Lhoták, Laurie Hendren: *Run-time Evaluation of Opportunities for Object Inlining in Java*. In *Concurrency and Computation: Practice and Experience*, volume 17, issue 5–6, pages 515–537. John Wiley & Sons, 2005. doi:10.1002/cpe.848

[Lindholm99] Tim Lindholm, Frank Yellin: *The Java™ Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[Loidl07] Stefan Loidl: *Compiler Data Flow Visualization*. Master's thesis, Institute for System Software, Johannes Kepler University Linz, 2007.

[Manson05] Jeremy Manson, William Pugh, Sarita V. Adve: *The Java Memory Model*. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391. ACM Press, 2005. doi:10.1145/1040305.1040336

[Mössenböck00] Hanspeter Mössenböck: *Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java HotSpot™ Client Compiler*. Technical Report 15, Institute for Practical Computer Science, Johannes Kepler University Linz, 2000.

[Mössenböck02] Hanspeter Mössenböck, Michael Pfeiffer: *Linear Scan Register Allocation in the Context of SSA Form and Register Constraints*. In *Proceedings of the International Conference on Compiler Construction*, LNCS 2304, pages 229–246. Springer-Verlag, 2002.

[Muchnick97] Steven S. Muchnick: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[NetBeans08] *NetBeans Module and Rich Client Application Development*, 2008. http://platform.netbeans.org

[Paleczny01] Michael Paleczny, Christopher Vick, Cliff Click: *The Java HotSpot™ Server Compiler*. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12. USENIX, 2001.

[Pelegri88] Eduardo Pelegrí-Llopart, Susan. L. Graham: *Optimal Code Generation for Expression Trees: An Application BURS Theory*. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 294–308. ACM Press, 1988. doi:10.1145/73560.73586

[Petrank02] Erez Petrank, Dror Rawitz: T*he Hardness of Cache Conscious Data Placement*. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 101–112. ACM Press, 2002. doi:10.1145/503272.503283

[Pettis90] Karl Pettis, Robert C. Hansen: *Profile Guided Code Positioning*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27. ACM Press, 1990. doi:10.1145/93542.93550

[Poletto99] Massimiliano Poletto, Vivek Sarkar: *Linear Scan Register Allocation*. In *ACM Transactions on Programming Languages and Systems*, volume 21, issue 5, pages 895–913. ACM Press, 1999. doi:10.1145/330249.330250

[Pozo99] Roldan Pozo, Bruce Miller: *SciMark 2.0*, 1999. http://math.nist.gov/scimark2/

[Qian02] Feng Qian, Laurie J. Hendren, Clark Verbrugge: *A Comprehensive Approach to Array Bounds Check Elimination for Java*. In *Proceedings of the International Conference on Compiler Construction*, LNCS 2304, pages 325–342. Springer-Verlag, 2002.

[Reder07] Alexander Reder: *Bytecode Visualizer*. Bachelor thesis, Institute for System Software, Johannes Kepler University Linz, 2007.

[Reder08] Alexander Reder: *Visualization of Machine Code*. Practical in Software Engineering, Institute for System Software, Johannes Kepler University Linz, 2008.

[Rubin02] Shai Rubin, Rastislav Bodík, Trishul Chilimbi: *An Efficient Profile-Analysis Framework for Data-Layout Optimizations*. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 140–153. ACM Press, 2002. doi:10.1145/503272.503287

[Russell06] Kenneth Russell, David Detlefs: *Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 263–272. ACM Press, 2006. doi:10.1145/1167473.1167496

[Scales00] Daniel J. Scales, Keith H. Randall, Sanjay Ghemawat, Jeff Dean: *The Swift Java Compiler: Design and Implementation*. WRL Research Report 2000/2, Compaq Western Research Laboratory, 2000.

[Shuf02a] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, Jaswinder Pal Singh: *Exploiting Prolific Types for Memory Management and Optimizations*. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–306. ACM Press, 2002. doi:10.1145/503272.503300

[Shuf02b] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, Jaswinder Pal Singh: *Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times*. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 13–25. ACM Press, 2002. doi:10.1145/582419.582422

[Siegwart06] David Siegwart, Martin Hirzel: *Improving Locality with Parallel Hierarchical Copying GC*. In *Proceedings of the International Symposium on Memory Management*, pages 52–63. ACM Press, 2006. doi:10.1145/1133956.1133964

[Spec98] Standard Performance Evaluation Corporation: *The SPECjvm98 Benchmarks,* 1998. http://www.spec.org/jvm98/

[Spec05] Standard Performance Evaluation Corporation: *The SPECjbb2005 Benchmark,* 2005. http://www.spec.org/jbb2005/

[Stiftner06] Bernhard Stiftner: *Comparison of Eclipse RCP and NetBeans Platform*. Bachelor thesis proposal, Institute for System Software, Johannes Kepler University Linz, 2006.

[SunHotSpot] Sun Microsystems, Inc.: *The Java HotSpot™ Performance Engine Architecture*. White Paper, 2008.

[SunJava6] Sun Microsystems, Inc.: *Java Platform, Standard Edition 6 Releases,* 2008. http://download.java.net/jdk6/

[SunJava7]	Sun Microsystems, Inc.*: Java Platform, Standard Edition 7 Snapshot Releases, 2008*. https://jdk7.dev.java.net/
[SunMemory]	Sun Microsystems, Inc.: *Memory Management in the Java HotSpot™ Virtual Machine*. White Paper, 2006.
[SunOpenJDK]	Sun Microsystems, Inc.: *Open-Source JDK Community*, 2008. http://openjdk.java.net/
[Traub98]	Omri Traub, Glenn Holloway, Michael D. Smith: *Quality and Speed in Linear-Scan Register Allocation*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151. ACM Press, 1998. doi:10.1145/277650.277714
[Ungar84]	David Ungar: Generation Scavenging: *A Non-Disruptive High Performance Storage Reclamation Algorithm*. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM Press, 1984. doi:10.1145/800020.808261
[Vallée-Rai99]	Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, Vijay Sundaresan: *Soot – A Java Bytecode Optimization Framework*. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135. IBM Press, 1999.
[Veldema01]	Ronald S. Veldema, Thilo Kielmann, Henri E. Bal: *Optimizing Java-Specific Overheads: Java at the Speed of C?* In *Proceedings of the International Conference on High-Performance Computing and Networking*, LNCS 2110, pages 685–692. Springer-Verlag, 2001. doi:10.1007/b80746
[Veldema05]	Ronald S. Veldema, Ceriel J. H. Jacobs, Rutger F. H. Hofman, Henri E. Bal. *Object Combining: A New Aggressive Optimization for Object Intensive Programs*. In *Concurrency and Computation: Practice and Experience*, volume 17, issue 5-6, pages 439–464. John Wiley & Sons, 2005. doi:10.1002/cpe.836
[Welch84]	Terry A. Welch: *A Technique for High-Performance Data Compression*. In *Computer*, volume 17, issue 6, pages 8–19. IEEE Computer Society, 1984. doi:10.1109/MC.1984.1659158
[Wimmer04]	Christian Wimmer: *Linear Scan Register Allocation for the Java HotSpot™ Client Compiler*. Master's thesis, Institute for System Software, Johannes Kepler University Linz, 2004.

[Wimmer05] Christian Wimmer, Hanspeter Mössenböck: *Optimized Interval Splitting in a Linear Scan Register Allocator*. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141. ACM Press, 2005. doi:10.1145/1064979.1064998

[Wimmer06] Christian Wimmer, Hanspeter Mössenböck: *Automatic Object Colocation Based on Read Barriers*. In *Proceedings of the Joint Modular Languages Conference*, LNCS 4228, pages 326–345. Springer-Verlag, 2006. doi:10.1007/11860990_20

[Wimmer07] Christian Wimmer, Hanspeter Mössenböck: *Automatic Feedback-Directed Object Inlining in the Java HotSpot™ Virtual Machine*. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 12–21. ACM Press, 2007. doi:10.1145/1254810.1254813

[Wimmer08] Christian Wimmer, Hanspeter Mössenböck: *Automatic Array Inlining in Java Virtual Machines*. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM Press, 2008.

[Wirth92] Niklaus Wirth, Jürg Gutknecht: *Project Oberon*. Addison-Wesley, 1992.

[Würthinger06] Thomas Würthinger: *Visualization of Java Control Flow Graphs*. Bachelor thesis, Institute for System Software, Johannes Kepler University Linz, 2006.

[Würthinger07a] Thomas Würthinger: *Visualization of Program Dependence Graphs*. Master's thesis, Institute for System Software, Johannes Kepler University Linz, 2007.

[Würthinger07b] Thomas Würthinger, Christian Wimmer, Hanspeter Mössenböck: *Array Bounds Check Elimination for the Java HotSpot™ Client Compiler*. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*, pages 125–133. ACM Press, 2007. doi:10.1145/1294325.1294343

[Würthinger08a] Thomas Würthinger, Christian Wimmer, Hanspeter Mössenböck: *Visualization of Program Dependence Graphs*. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, 2008.

[Würthinger08b] Thomas Würthinger, Christian Wimmer, Hanspeter Mössenböck: *Array Bounds Check Elimination for the Java HotSpot™ Client Compiler*. Submitted to *Science of Computer Programming*, 2008

[Yasue03] Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, Toshio Nakatani: *An Efficient Online Path Profiling Framework for Java Just-in-Time Compilers*. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 148–158. IEEE Computer Society, 2003. doi:10.1109/PACT.2003.1238011

[Zhong04] Yutao Zhong, Maksim Orlovich, Xipeng Shen, Chen Ding: *Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–266. ACM Press, 2004. doi:10.1145/996841.996872