JOHANNES KEPLER
UNIVERSITY LINZ
Research and teaching network

Thomas Kotzmann

Escape Analysis in the Context of Dynamic Compilation and Deoptimization

A PhD 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
o.Univ.-Prof. Dipl.-Ing. Dr. Hanspeter Mössenböck
a.Univ.-Prof. Dipl.-Ing. Dr. Andreas Krall

Linz, October 2005

Abstract

Escape analysis is used in compilers to identify and optimize the allocation of objects that are accessible only within the allocating method or thread. This thesis presents a new *intra-* and *interprocedural* analysis for a dynamic compiler, which has to cope with dynamic class loading and deoptimization.

The analysis was implemented for Sun Microsystems' Java HotSpot client compiler. It operates on an intermediate representation in SSA form and introduces *equi-escape sets* for the efficient propagation of escape information between related objects. Analysis results are used for *scalar replacement of fields*, *stack allocation of objects* and *synchronization removal*. The interprocedural analysis supports the compiler in inlining decisions and allows actual parameters to be allocated in the stack frame of the caller. A lightweight bytecode analysis produces interprocedural escape information for methods that have not been compiled yet.

Dynamic class loading possibly invalidates the generated machine code. In this case, the execution of a method is continued in the interpreter. This is called *deoptimization*. Since the interpreter does not know about scalar replacement, stack allocation or synchronization removal, the deoptimization framework was extended to reallocate and relock objects on demand.

Kurzfassung

Escape-Analyse identifiziert und optimiert die Allokation von Objekten, auf die nur eine Methode oder ein Thread zugreifen kann. Diese Dissertation präsentiert eine neue *intra-* und *interprozedurale* Analyse für einen Just-in-Time-Compiler, der dynamisches Laden von Klassen und Deoptimierung unterstützt.

Die Analyse wurde für den Java HotSpot Client Compiler von Sun Microsystems implementiert. Sie arbeitet auf einer Zwischensprache in SSA-Form und verwaltet voneinander abhängige Objekte effizient in Mengen. Die Ergebnisse werden verwendet, um Felder durch skalare Variablen zu ersetzen, Objekte am Stack anzulegen und Synchronisation zu entfernen. Die interprozedurale Analyse unterstützt den Compiler bei Inline-Entscheidungen und identifiziert Methodenparameter, die am Stack angelegt werden können. Eine schnelle Bytecode-Analyse liefert Informationen für Methoden, die noch nicht compiliert wurden.

Dynamisches Laden von Klassen kann bestehenden Maschinencode invalidieren. In diesem Fall wird die Ausführung der Methode im Interpreter fortgesetzt. Dies nennt man *Deoptimierung.* Da der Interpreter die Optimierungen des Compilers nicht kennt, wurde das Framework zur Deoptimierung so erweitert, dass es Objekte nachträglich am Heap anlegt und sperrt.

Contents

Acknowledgments

A lot of people have contributed to this work. First and foremost, I thank my advisor Hanspeter Mössenböck, without whose support and encouragement this thesis had not been possible. Likewise, I thank Andreas Krall from the Vienna University of Technology for the examination of my thesis.

I want to thank Kenneth Russell, David Cox and Thomas Rodriguez from the Java HotSpot compiler group at Sun Microsystems for the close collaboration and the continuous support of our project. I also thank David Detlefs from Sun Microsystems Laboratories. He contributed many ideas and suggestions for improvement.

Special thanks go to my colleague Christian Wimmer, who also works on Sun's compiler. I appreciate our discussions, his comments on the thesis and his competent advice. My other colleagues provided help in numerous ways. I thank Karin Gusenbauer, Markus Löberbauer, Albrecht Wöß, Wolfgang Beer, Herbert Prähofer, Hermann Lacheiner, Michael Kaffenda and Kurt Prünner.

I would like to express my gratitude to all reviewers for their diligence and constructive criticism. Their feedback was invaluable in improving the quality of this thesis.

Finally, I owe special thanks to my parents and my brother. I am grateful for their love and their support throughout my studies.

1 Introduction

Portability and security earned the Java programming language great popularity, even though Java programs ran slowly in the beginning. In view of the competition between different runtime environments, a lot of work and research effort has been spent on the improvement of performance. Today, Java offers modern language features combined with an adequate execution speed.

A major performance improvement was achieved by just-in-time compilation. The subject of this PhD thesis is the implementation of an optimization called escape analysis for the client just-in-time compiler of the Java HotSpot VM. This chapter describes the context of the project, the architecture of the virtual machine, and the challenges of this work.

1.1 Java

The Java programming language originated from a research project to develop software for network devices and embedded systems. Confronted with the deficiencies of C++, James Gosling developed the programming language Oak at the beginning of the nineties. After several years of experience with the language, it was targeted to the Internet and renamed to Java [40].

With the vision of writing an application once and deploying it to different platforms without recompilation, Java was designed as a portable and safe object-oriented programming language. Although its syntax is similar to C++, complex and unsafe features such as multiple inheritance or pointer arithmetic were omitted.

Java is a strongly typed language, which means that every variable and every expression has a type that is known at compile time. Arithmetic operations, type casts, array and object accesses are checked for validity before they are executed. If the program attempts to perform an invalid operation, an exception is thrown. The Java compiler ensures that exceptions are handled properly.

From the beginning on, Java incorporated support for exception handling, reflection of run-time type information, multi-threading and remote method invocation. Recently, the language was extended by constructs for parameterized types, known as *generics* [15].

To write and run Java programs, a *Java Development Kit* (JDK) must be installed. Sun Microsystems distributes a JDK free of charge. It provides a virtual machine, a set of development tools and a library of classes for input and output, graphical user interfaces, network programming and internationalization [85].

1.2 Java HotSpot VM

Java achieves portability by translating source code into platform-independent bytecodes. To run Java programs on a particular platform, a *Java virtual machine* must exist for that platform. It executes bytecodes after checking that they do not compromise the security or reliability of the underlying machine.

The Java HotSpot VM [84] is Sun's implementation of a Java virtual machine [62]. It is available for a variety of platforms and operating systems, such as Sun Solaris, Microsoft Windows and Linux, running on SPARC, the Intel IA-32 or Intel IA-64 architecture. The VM delivers high performance by using just-in-time compilation and state-of-the-art garbage collection.

The Java 2 SDK contains two flavors of the VM: the client and the server VM. They share the same garbage collector, interpreter, thread and lock subsystems, but use different just-in-time compilers. The server VM achieves maximum peak operating speed for long-running server applications, while the client VM is tuned to start up fast and require a small memory footprint.

1.2.1 Memory Management

In traditional languages such as C++, it is the responsibility of the programmer to release memory when an object is no longer needed. This approach is error-prone as it can lead to *dangling pointers* if an object is released too soon, and to *memory leaks* if the programmer forgets to free an object.

The Java VM specification requires the runtime environment to provide built-in automatic memory management. Objects are deallocated by a garbage collector, which is invoked once in a while to examine the heap and release the memory of objects if they are not referenced any more [54].

The Java HotSpot VM implements a generational garbage collector. New objects are allocated in the young generation. It is collected by a *stop-and-copy* algorithm, which copies live objects between two alternating regions. The cost of a collection is proportional to the amount of surviving data rather than to the size of the generation.

To avoid repeated copying of long-lived objects within the young generation, objects are moved to the old generation when they have survived a certain number of collection cycles. The old generation is collected by a *mark-and-compact* algorithm. Starting from the set of root pointers, it marks all reachable objects, assigns consecutive memory slots to them, adjusts pointers and finally moves the objects to their new locations.

Alternatively, the user can switch to the incremental *train algorithm*. In this case, the old generation is split into small fixed-sized blocks, so-called *cars*. A set of cars forms a *train*. The algorithm processes only a single car at a time. It moves live objects out of the car and tries to group related objects in the same train. Afterwards, the collected car is free. When a garbage structure is entirely contained within a train, it is reclaimed [41].

The Java HotSpot VM also supports concurrent and parallel garbage collection. The *parallel scavenge* algorithm still interrupts the user program but distributes garbage collection to parallel threads, whereas the *concurrent mark-and-sweep* [76] collector can run concurrently with the user program. Both algorithms better exploit multi-processor systems and minimize GC pauses.

1.2.2 Interpretation and Hot Spot Detection

Execution of a Java program starts immediately after the main class has been loaded. All methods are initially executed by the interpreter which implements a naive stack machine. It steps through the bytecodes of a method and executes a code template for each bytecode.

Interpretation is slow because almost no optimizations are applied. Performance can be improved by translating bytecodes into optimized machine code immediately before or even while the application is running. This is called *just-in-time (JIT) compilation*. Since compilation time adds to the overall run time, and typical programs spend the majority of their time executing only a small part of their code, the Java HotSpot VM compiles only the most frequently executed methods into machine code. They are referred to as *hot spots*.

The interpreter counts how often each method is called. If an invocation counter exceeds a certain threshold, the corresponding method is scheduled for compilation. Once a method has been compiled, its machine code is executed upon every future invocation. Invocation counters save the VM from wasting resources on the compilation of rarely called methods. Besides, as most classes used in a method are loaded during interpretation, information about them is already available during JIT compilation. The compiler is able to inline more methods and generate better machine code.

If a method contains a long-running loop, it may be compiled regardless of its invocation frequency. The VM counts the number of taken backward branches. When a threshold is reached, it suspends interpretation and compiles the method. A new stack frame for compiled code is set up and initialized to match the interpreter's one. Then the execution of the method continues in machine code. Switching from interpreted to compiled code in the middle of a method is called *on-stack-replacement* (OSR) [35].

When a problem occurs during the compilation of a method, the compiler *bails out*. The virtual machine gives up compilation and keeps on interpreting the method. It is even possible to revert to interpretation when the machine code already executes. This is referred to as *deoptimization* and facilitates debugging and aggressive compiler optimizations [49].

1.2.3 Server Compiler

Originally, the Java HotSpot VM was delivered with only one highly optimizing but relatively slow compiler. Today, this compiler is called *server compiler* as it provides peak performance for long-running server applications at the cost of compilation speed.

The server compiler [71] proceeds through different phases when compiling a method. At first, it parses the bytecodes and translates them into an intermediate representation in form of a static single assignment (SSA) graph [23]. Each node represents an operation and points to the nodes that produce the values of the operands. A value flows directly from its definition to its uses. Control flow is represented via a different set of instructions. This allows optimizations to change the order of nodes without worrying about control flow.

During and after parsing, the front end performs various machine-independent optimizations, such as constant propagation, global value numbering, loop unrolling and dead code elimination. A method is inlined if it can be bound statically as well as if class hierarchy analysis or profiling during interpretation determine that there is only one suitable implementation. Virtual calls that cannot be inlined dispatch to a method entry with an inline cache [48].

The machine-independent instructions are translated to machine instructions with lowest execution costs using a bottom-up rewrite system (BURS) [73]. This is done before instructions are placed into basic blocks, so no block boundaries constrain instruction selection. To increase the portability of the compiler, the characteristics of the target hardware are specified in a separate machine description file.

Register allocation is based on a graph coloring algorithm [17]. It computes live ranges, builds an interference graph and colors it. Intervals that fail to get a color are split. This process is repeated until a complete coloring is found. The subsequent peephole optimization [4] tries to improve instruction sequences with regard to the target platform. Apart from machine code, the back end produces auxiliary information for garbage collection and deoptimization.

1.2.4 Client Compiler

The server compiler produces efficient machine code, but the optimizations require time. Low compilation speed is acceptable for long-running server applications, because compilation impairs performance only during the warm-up phase and can usually be done in the background if multiple processors are available. For interactive client programs with graphical user interfaces, however, response time is more important than peak performance.

The *client compiler* (often referred to as C1) is designed to achieve a trade-off between the performance of the generated machine code and compilation speed. It implements only few high-impact optimizations and a simple but fast register allocation. Although the original version got along with a single intermediate representation, a second one was added later to facilitate machine-oriented optimizations.

In the context of this PhD thesis, we developed and implemented a new escape analysis algorithm for the client compiler. While the server compiler operates on a *sea of instruction nodes*, the client compiler builds an explicit control flow graph. This makes it more appropriate for escape analysis.

The version of the client compiler that was extended by escape analysis differs from the one shipped with the current JDK 5.0. It generates an intermediate representation in SSA form, performs value numbering across basic block boundaries, and uses an optimized linear scan register allocation algorithm instead of a local register allocation. The structure of the compiler and the layout of its intermediate representations are described in Chapter 2.

1.3 Optimizations in Just-in-Time Compilers

Early Java virtual machines fully relied on interpretation. They had to fetch, decode and execute bytecode by bytecode and thus suffered from poor performance. Execution of machine code provides higher performance, but traditional static compilation would compromise Java's portability and security. Therefore, modern virtual machines compile Java programs dynamically at run time [25].

Dynamic compilation is a relatively old idea [8]. In 1960, McCarthy suggested to compile LISP programs fast enough, so that it is not necessary to save the generated code for future use [64]. The Smalltalk programming environment by PARC was one of the first to benefit from dynamic translation [31]. In the context of Java virtual machines, dynamic compilers are usually referred to as just-in-time (JIT) compilers, because they originally compiled a method just before its first invocation.

Time spent on compiling a method is time that could have been spent on interpreting the method's bytecodes. Compilation speed is crucial, because the execution of machine code needs to compensate compilation time. Static compilers can afford to perform sophisticated optimizations, but just-in-time compilers must produce fast machine code efficiently [1, 58].

On the other hand, a just-in-time compiler can tune a program exactly for the platform that executes the program. Based on profiling information, it can decide about optimizations [7] or compile only parts of a method [92]. It is even able to perform aggressive optimizations under optimistic assumptions and later undo the optimizations if the assumptions turn out to be wrong.

Typical optimizations in just-in-time compilers include constant folding, common subexpression elimination, null check elimination, loop unrolling, method inlining and polymorphic inline caches [53]. They have shown to measurably improve code quality at reasonable costs. Apart from them, some optimizations such as register allocation or escape analysis may be adapted to the needs and limitations of just-in-time compilers.

1.4 Problem Statement

In order to keep the Java HotSpot client compiler fast and simple, new optimizations are carefully evaluated before they get integrated into the compiler. In the course of time, more and more optimizations were added that improve code quality without impairing compilation speed substantially.

A few years ago, the client compiler group at Sun Microsystems thought about new advanced optimizations. The resulting proposal included escape analysis, which had already been implemented in several other, mostly static compilers and promised a significant speedup of allocation-intensive programs.

In Java, it is not possible to influence where objects are allocated and when they are deallocated. Escape analysis identifies objects that can be allocated on the stack instead of the heap or whose allocation can be eliminated at all. It helps to reduce the costs of object allocation and to remove unnecessary synchronization.

The aim of this project is to extend the client compiler by escape analysis for evaluation purposes. The implementation should meet the needs of a dynamic compiler, which poses the following challenges:

- As all optimizations occur at run time, they must run efficiently and rather be conservative if a small gain would imply a time-consuming analysis.
- When the compiler faces not yet loaded classes or uncompiled methods, it must either make pessimistic assumptions or infer estimations from the bytecodes.
- Dynamic class loading may invalidate existing machine code and require previous optimizations to be undone.
- The allocation of objects on the stack instead of the heap makes it necessary to adapt the runtime environment, especially the garbage collector and its card marking scheme (see Section 5.1.4).

1.5 Project History

The original version of the Java HotSpot client compiler was developed by Robert Griesemer and Srdjan Mitrovic as a clean and fast alternative to the server compiler. It used a single intermediate representation and performed only few high-impact optimizations. The back end implemented a simple but efficient register allocator extended by a heuristic that assigned unused registers to local variables within loops [42].

The research collaboration of Sun Microsystems and the Institute for System Software at the Johannes Kepler University Linz was initiated by Hanspeter Mössenböck, who spent a sabbatical at Sun Microsystems between June and August 2000. He modified an experimental variant of the client compiler to produce an intermediate representation in SSA form and implemented a graph coloring register allocator [66].

Graph coloring produces nearly optimal register allocation but is usually too slow for a just-in-time compiler. For this reason, Michael Pfeiffer provided the compiler with a linear scan register allocator, which assigns registers to values in a single linear scan over lifetime intervals [67]. The algorithm saved one *scratch register* for situations when a machine instruction requires an operand to be in a register but all registers are in use.

In the context of the author's master's thesis, the original compiler was ported from C++ to Java [56]. This work was laid out as an experiment to get an idea of how such an implementation would benefit from the advanced features of Java, how it could interface the VM, and how it would perform compared to the compiler written in C++. The Java port was integrated into the VM so that it started in interpreted mode and then continuously compiled itself.

Simultaneously, the SSA-based compiler was extended by a low-level intermediate representation (LIR) in addition to the high-level intermediate representation (HIR). Christian Wimmer improved the linear scan algorithm to get rid of the scratch register, to support holes in lifetime intervals and to optimize interval splitting. Besides, he extended the compiler to generate SSE and SSE2 instructions for floating-point operations [95, 97].

The linear scan workspace was the basis for the implementation of escape analysis described in this thesis. The algorithm benefits from the SSA form and the improvements in the back end. An intermediate report on the work was published in the proceedings of the VEE conference 2005 [57].

1.6 Structure of the Thesis

The rest of this thesis is organized as follows: Chapter 2 presents the compilation process without escape analysis, the architecture of the client compiler and the layout of its intermediate representations. Chapter 3 introduces our intra- and interprocedural approach to escape analysis and explains the computation of escape states under consideration of control flow.

The next two chapters describe the optimizations based on the results of escape analysis. Chapter 4 shows that allocations of non-escaping objects can be eliminated when their fields are replaced by scalar variables. Chapter 5 presents optimizations for thread-local objects, such as stack allocation and synchronization removal.

Chapter 6 deals with the run-time support required for a safe execution of optimized methods. It describes how problems caused by dynamic class loading are solved via deoptimization, how eliminated objects are recreated and unlocked objects are relocked using debugging information, and how stack objects are treated by the garbage collector.

Chapter 7 evaluates our escape analysis and the optimizations on several benchmarks and discusses the results. Chapter 8 gives an overview of related projects and compares them to our approach. Finally, Chapter 9 recapitulates the essential parts of our algorithms, summarizes the contributions of this work and gives an outlook on future work.

2 The HotSpot Client Compiler

When the invocation counter of a method has reached a certain threshold (see Section 1.2.2), the bytecodes of the method are compiled to machine code. The *client compiler* is a simple and fast compiler dedicated to client programs, applets and graphical user interfaces. Its objective is to achieve high compilation speed at a potential cost of peak performance [42].

In the context of this project, the client compiler was extended by escape analysis and related optimizations. This chapter deals with the overall architecture of the compiler. It describes the compilation phases and the produced intermediate representations. Moreover, we introduce terms and abbreviations that are used in the subsequent chapters.

2.1 Java Bytecodes

Java allows developers to write an application once and deploy it to different platforms without redeveloping or recompiling a single line of code. For this purpose, the Java source code is compiled to platform-independent *bytecodes*. This frees the JVM from parsing and analyzing plain-text source code [62].

Bytecodes are executed using an operand stack. They pop operands from the stack, operate on them and push the result back onto the stack. Consider the following Java method which computes the integral dual logarithm, i.e. the position of the most-significant bit that is set:

```
static int log2(int n) {
  int p = 0;
  while (n > 1) {
    n = n >> 1;
    p = p + 1;
  }
  return p;
}
```

It is translated into the bytecodes below. The number in front of each bytecode specifies the start position of the bytecode within the method and is often referred to as the *bytecode index* (BCI). The comments on the right side recapitulate the original Java source code statements.

```
 0: iconst_0        // p = 0
 1: istore_1
 2: iload_0         // while (n > 1)
 3: iconst_1
 4: if_icmple 18
 7: iload_0         // n = n >> 1
 8: iconst_1
 9: ishr
10: istore_0
11: iload_1         // p = p + 1
12: iconst_1
13: iadd
14: istore_1
15: goto 2
18: iload_1         // return p
19: ireturn
```

Each instruction consists of one byte specifying the operation to be performed, followed by zero or more operands. Bytecodes that load or store a local variable refer to the variable via its index in the method's stack frame. A loop is converted into a conditional jump and a backward branch. The instruction set of a JVM consists of about 200 instructions, which can be used to

- transfer values between local variables and the operand stack,
- perform arithmetic and logical operations,
- convert between types,
- create and modify objects and arrays,
- directly manipulate the operand stack,
- jump to another position within the same method,
- call a method or return from a method,
- throw or catch an exception, or
- synchronize multiple threads.

Bytecodes usually encode type information about the operations they perform. Many of them have no operands and consist only of an operation code. Since the bytecodes are stored in a compact binary format, the method above actually occupies only 20 bytes.

2.2 Overall Compiler Architecture

The client compiler consists of a front end and a back end, each of which operates on its own intermediate representation (see Figure 2.1). Of course, it would be possible to generate machine code directly from the bytecodes as well, but some of the information needed by the back end cannot be determined in a single pass over the bytecodes. Instead of iterating over the bytecodes multiple times, an intermediate representation is built. This makes the compiler not only simpler, but also faster and more reliable.

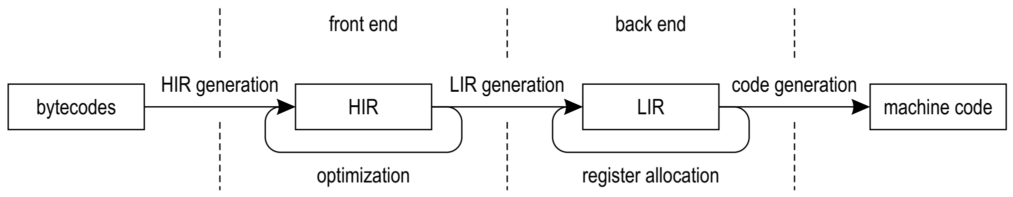


Figure 2.1: Overall architecture of the client compiler

The front end is responsible for parsing the bytecodes and constructing the *high-level intermediate representation* (HIR), which is based on the *static single assignment form* [26]. A control flow graph is built and some high-level optimizations such as constant folding, common subexpression elimination and inlining are performed.

The back end converts the optimized HIR into the *low-level intermediate representation* (LIR). Even though the LIR is almost independent of the target machine and basic blocks are still evident, structure and terms follow closely those of a real processor. Register allocation is performed on the LIR, because all operands requiring a register in machine code are explicitly visible.

After register allocation, machine code can be generated from the LIR in a rather simple and straightforward way. The machine code is finally installed into the data structures of the VM, so that it gets executed whenever the method is called in the future. Additionally, the compiler produces meta data that is used by the VM for garbage collection and deoptimization (see Chapter 6).

2.3 Front End

In order to build the HIR, the front end iterates over the bytecodes twice. At first, the *block list builder* determines the starts of all basic blocks by examining the destinations of jumps and the successors of conditional jump instructions.

New empty blocks are created and mapped to the bytecode indices of the block start instructions. During the second iteration, the *graph builder* fills the basic blocks with HIR instructions via an abstract interpretation of the bytecodes.

2.3.1 Control Flow Graph

The control flow graph (CFG) describes the instructions and branches in a method [4]. Each node corresponds to a *basic block*, i.e. the longest possible sequence of instructions without jumps or jump targets in the middle. The nodes are connected via directed edges representing the control flow in the graph.

A basic block of the HIR consists of a single-linked list of HIR instructions starting with a `BlockBegin` instruction (see Figure 2.2). The last instruction is always a `BlockEnd` instruction, which points to a possibly empty list of successor blocks. Various concrete subclasses of `BlockEnd` are available for branches, unconditional jumps and returns. They differ in their number of successors as well as in the generated machine code.

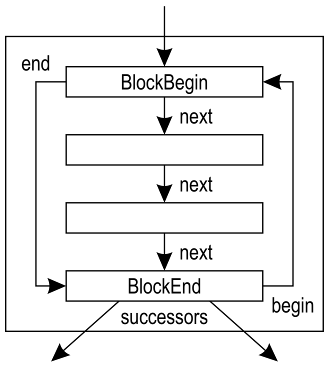


Figure 2.2: Structure of a basic block

The `BlockBegin` and `BlockEnd` instruction store a reference to each other in the `begin` and the `end` field, respectively. This enables the compiler to traverse all basic blocks of the control flow graph without having to inspect every single instruction in between.

2.3.2 Static Single Assignment Form

The front end operates on an intermediate representation (see next section) that is in static single assignment (SSA) form. This means that for every variable there is just a single point in the program where a value is assigned to it. The standard way to express a program in SSA form is to subscript the original variables, so that each assignment instruction creates a new variable name. Figure 2.3 shows the control flow graph for the `log2` method in SSA form.

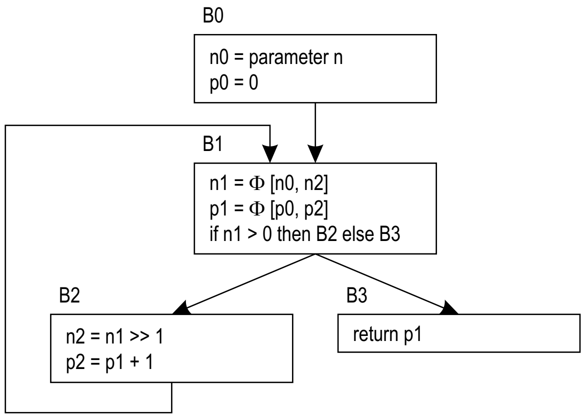


Figure 2.3: Method in SSA form

At points where control flow joins, so-called *phi functions* are inserted to merge different incarnations of the same variable. The number of operands of a phi function is equal to the number of incoming control flow edges. Each operand represents one version of the variable, and the phi function itself produces another version. In the above CFG, the values `p0` and `p2` flow into the loop header via two control flow edges and are joined into a new value `p1`. Although the variable `p1` gets different values depending on the control flow, it has a single point of definition.

In SSA form, two uses of a variable with the same name are guaranteed to refer to the same value. This simplifies data flow dependent optimizations, such as constant propagation, global value numbering and loop-invariant code motion.

2.3.3 High-Level Intermediate Representation

After the control flow graph has been built, the basic blocks are filled with instructions of the platform-independent *high-level intermediate representation* (HIR). The compiler maintains a *state object* to simulate the run-time operand stack. An instruction that loads or computes a value represents both the operation and its result, so that operands appear as pointers to prior instructions. Every instruction is identified by its type and a unique sequence number. For instance, `i1` refers to the integer value produced by instruction 1, `f2` to the floating-point value produced by instruction 2, and `a3` to an object (i.e. a reference value) produced by instruction 3.

Instructions for loading or storing local variables are eliminated during the abstract interpretation of the bytecodes. For this purpose, the state object contains a *locals array*, which keeps track of the values most recently assigned to a local variable [66]. If an instruction creates a value for the local variable with the number n, a pointer to this instruction is stored in the locals array at position n. If an instruction uses a local variable as an operand, it is provided with the corresponding value from the locals array, i.e. a pointer to the instruction where the value was created.

In this thesis, the basic blocks of the HIR are printed in the following form: The first line specifies the number of the block, the bytecode indices of the first and the last instruction in the block, and a list of successor blocks. If the block introduces new phi functions, they are printed with their operands below. Then the HIR instructions of the basic block follow. Every line corresponds to one instruction and specifies the original bytecode index, the use count, the identification number and a description. Instructions that must be executed in the order of the bytecodes are marked as *pinned*, indicated by a point in the first column. An instruction refers to other instructions via their identification numbers.

The first two basic blocks of the `log2` method are shown below. Basic block `B4` is introduced for technical reasons and has no equivalent in the bytecodes. Basic block `B0` loads the constant 0 as the initial value of the local variable `p` and then jumps to the loop header block `B1`.

```
B4 [0, 0] -> B0
__bci__use__tid____instr________________________________
. 0    0     18    std entry B0

B0 [0, 1] -> B1
__bci__use__tid____instr________________________________
  0    1    i5     0
. 1    0     6     goto B1
```

Basic block `B1` has two predecessors. Therefore, phi functions must be created for local variables as required by the SSA form. The phi function `i7` for the variable `n` merges the value `i4` (the initial value of the parameter `n`) with the value `i12` (the value of `n` computed by the shift operation in the loop body). The operands of the phi function `i8` for the variable `p` are `i5` from above and `i14`, which is computed in the loop body.

```
B1 [2, 4] -> B3 B2
Locals:
 0  i7 [i4 i12]
 1  i8 [i5 i14]
```

```
__bci__use__tid____instr________________________________
  3    1    i9     1
. 4    0     10    if i7 <= i9 then B3 else B2
```

As long as the loop condition (`i7 <= i9`) is true, the loop body `B2` is executed. It computes the new values `i12` and `i14` for the local variables `p` and `n` before it jumps back to the loop header. If the condition is false, basic block `B3` is executed in order to load and return the current value `i8` of the variable `p`.

```
B2 [7, 15] -> B1
__bci__use__tid____instr________________________________
  8    2    i11    1
. 9    1    i12    i7 >> i11
. 13   1    i14    i8 + i11
. 15   0     15    goto B1 (safepoint)

B3 [18, 19]
__bci__use__tid____instr________________________________
. 19   0    i16    ireturn i8
```

2.3.4 Optimizations

Both during and after generation of the HIR, several optimizations are performed. They benefit from the simple structure of the HIR and the SSA form. The optimizations are not really crucial for the quality of the generated code, but they are done because they are cheap and show some effect.

Every HIR instruction is put into its simplest form, referred to as the *canonical form*. This transformation covers constant folding and simplification of branches. If an arithmetic or logical operation has only constant operands, no machine code for the calculation must be emitted because the result can be computed already at compile time. If the condition of a branch is always true or always false, the branch is replaced by an unconditional jump.

A simple form of value numbering is used to eliminate common subexpressions within a basic block. Every generated HIR instruction is inserted into a hash table. Before an instruction is appended to the HIR, the hash table is searched for an existing instruction that computes the same value. If such an instruction is found, it is used instead of the new one. This way, the back end generates machine code for the instruction only once and keeps the result in a register or in memory.

If the compiler parses a method call, it tries to inline the method, i.e. replace the method call by a copy of the method body. This eliminates the overhead of method dispatching. To avoid excessive inlining, the size of the callee is restrained to a certain threshold. Additionally, the compiler must be able to unambiguously determine the actual target method. This is possible if the callee is declared static or final, or if class hierarchy analysis reveals that currently only one suitable method exists. If a class is loaded later that provides another suitable method, the machine code is invalidated and must be regenerated (see Chapter 6).

The Java language specification requires that an exception is thrown if the value null is dereferenced. For this reason, null checks must be inserted into the machine code whenever an object is accessed, unless the compiler can prove that the object is non-null. A null check can be eliminated if the object is the receiver of the current method invocation or if a null check has already been performed on the same object before.

Finally, the compiler searches the control flow graph for conditional expressions. They appear as branches that load one of two values depending on a condition and do not contain any other instructions. This pattern is replaced by a special HIR instruction, for which the back end generates more efficient machine code than for the original branch.

2.4 Back End

The back end translates the optimized HIR into the *low-level intermediate representation* (LIR) and performs register allocation and code generation. Although machine code could be generated directly from the HIR as it was done in earlier versions of the client compiler, the LIR facilitates various low-level optimizations and a more sophisticated register allocation.

2.4.1 Low-Level Intermediate Representation

The LIR is conceptually similar to machine code, but still mostly platform-independent. The contents of a basic block are stored in an array list, which allows a fast iteration over all operations. In contrast to HIR instructions, LIR operations use explicit operands instead of references to prior instructions.

The method entry block `B4` loads the parameter values. For each operand, its location and its type are specified. `[stack:0|I]` refers to the integer parameter `n` in stack slot 0. It is loaded into the virtual register `R40`. After LIR generation, virtual registers are mapped to physical ones by register allocation.

```
B4 [0, 0] -> B0
__id__operation________________________________________
   0  label [label:0x2b40b3c]
   2  std_entry
   4  move [stack:0|I] [R40|I]
   6  branch [AL] [B0]
```

LIR and machine code do not contain phi functions any more. Every phi function has been replaced by a virtual register. The operands of the phi function are assigned to this register at the end of the corresponding predecessor blocks. In the example below, the phi function for `p` in `B1` is represented by `R43`. Its first operand 0 is assigned to `R43` at the end of `B0`, and its second operand `R45` is assigned to `R43` at the end of `B2`. Similarly, the phi function for `n` is represented by register `R42`.

```
B0 [0, 1] -> B1
__id__operation________________________________________
   8  label [label:0x2b2238c]
  10  move [R40|I] [R42|I]
  12  move [int:0|I] [R43|I]
  14  branch [AL] [B1]

B1 [2, 4] -> B3 B2
__id__operation________________________________________
  16  label [label:0x2b22464]
  18  cmp [R42|I] [int:1|I]
  20  branch [LE] [B3]
  22  branch [AL] [B2]
```

Arithmetic and logical LIR operations typically specify two source and one target operand. Even though the LIR allows the use of three different operands, the Intel IA-32 architecture requires that the left source operand equals the target operand [51]. Therefore, each binary computation is preceded by a move operation that copies the left source operand into the target register first. The operations can then be directly translated into machine code.

```
B2 [7, 15] -> B1
__id__operation________________________________________
  24  label [label:0x2b400e4]
  26  move [R42|I] [R44|I]
  28  shift_right [R44|I] [int:1|I] [R44|I]
  30  move [R43|I] [R45|I]
  32  add [R45|I] [int:1|I] [R45|I]
  34  safepoint [bci:15]
  36  move [R45|I] [R43|I]
  38  move [R44|I] [R42|I]
  40  branch [AL] [B1]
```

Certain HIR instructions, such as backward branches and return instructions, represent a *safepoint*, at which the program may be stopped for garbage collection. These safepoints are explicitly visible in the LIR. Basic block B2 ends with a backward branch, so a safepoint operation is emitted right in front of the move instructions that resolve the phi functions.

The last basic block loads the current value of p from the register R43 and returns it to the caller. Since the method result must always be stored in the EAX register by convention, both the move and the return operation use this physical register instead of a virtual one.

```
B3 [18, 19]
__id__operation_______________________________________
  42  label [label:0x2b401bc]
  44  move [R43|I] [eax|I]
  46  return [eax|I]
```

2.4.2 Register Allocation

Register allocation determines which variable or temporary value will be stored in which register. This is an important optimization because registers are usually a limited resource. Today's standard register allocation algorithm is based on graph coloring, which leads to good code quality [68] but is too slow for the just-in-time compilation of interactive programs. Even heuristic implementations have a quadratic run-time complexity.

For this reason, the Java HotSpot client compiler uses a linear scan register allocator [74, 95, 97], which is faster than graph coloring and yields results of only slightly inferior quality. It operates on the LIR of a method and maps virtual registers to physical ones. The result is a modified LIR which provides the basis for code generation.

The linear scan algorithm arranges all instructions of a method in a linear order, before it computes *lifetime intervals* for virtual registers. A lifetime interval spans from the definition of a value to its last use. A data flow analysis is performed to generate lifetime information for loops and branches. Figure 2.4 illustrates the intervals for the log2 method. Each line corresponds to one virtual register. White rectangles represent live ranges and grey bars indicate where values are used.

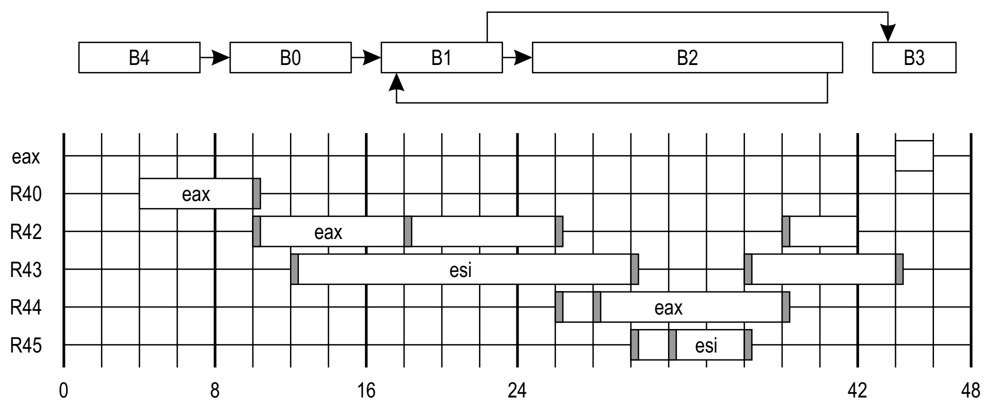


Figure 2.4: Lifetime intervals after register allocation

As shown in the figure, intervals need not be continuous. They may contain so-called *lifetime holes* [87]. For example, the virtual register R42 is read by operation 26 and written by operation 38. Between the two operations, R42 does not contain a useful value because it is not read again before it is overwritten. The lifetime hole avoids wasting a physical register for this section.

The register allocator scans over all intervals in the order of their starting points and immediately assigns physical registers to them. If two values are live at the same time, they must not end up in the same register. Two intervals that do not intersect can get the same physical register assigned.

When more intervals are live than physical registers are available, a value must be swapped out to memory. This is called *spilling*. As a simple heuristic, the value that is not used for the longest time is spilled. Later it may be loaded into a register again. To avoid loads and stores in each loop iteration, the algorithm prefers to spill values used after a loop instead of those used in the loop.

If a value changes its location due to spilling, the corresponding interval needs to be split. The negative impact of spilling can be reduced by splitting at a position that leads to a minimal number of spill loads and stores executed at run time. Optimized interval splitting considers use positions, kinds of uses and positions where operands must reside in a register [97].

2.4.3 Code Generation

Machine code is generated immediately after register allocation. The compiler traverses the LIR, operation by operation, and emits appropriate machine instructions into a code buffer. The contents of the code buffer are later copied into a *native method object*.

The back end packages methods for the generation of machine instructions and code sequences in a *macro assembler*, which is available for different platforms. This section shows the code generated for the `log2` method when it is compiled for the Intel IA-32 platform [51].

The very first machine instruction checks for a possible stack overflow in the near future. If the test fails, there is still enough space on the stack to throw an exception. The subsequent three instructions set up a stack frame for the current method invocation.

```
009F8B91  mov          dword ptr [esp-3000h], eax
009F8B98  push         ebp
009F8B99  mov          ebp, esp
009F8B9B  sub          esp, 10h
```

Afterwards, the parameter `n` is loaded into the `EAX` register and the constant 0 is assigned to the local variable `p` in the `ESI` register. No instruction is emitted to jump to the beginning of the loop, because the loop header immediately follows the current block. Since the loop header is the target of a backward branch, it is aligned at a word boundary for performance reasons by inserting `nop` instructions.

```
009F8B9E  mov          eax, dword ptr [ebp+8]
009F8BA1  mov          esi, 0
009F8BA6  nop
009F8BA7  nop
```

The following six machine instructions test the loop condition and compute new register values. The assembler selects an `inc` instruction to perform the addition of 1. Safepoint operations are implemented as an access to the absolute address 380100h. If garbage collection is required, the JVM marks the corresponding memory location as unreadable and thus causes all threads to trap and stop at the next safepoint [2].

```
009F8BA8  cmp          eax, 1
009F8BAB  jle          009F8BBF
009F8BB1  sar          eax, 1
009F8BB3  inc          esi
009F8BB4  test         dword ptr [380100h], eax
009F8BBA  jmp          009F8BA8
```

The final piece of code copies the method result from the `ESI` into the `EAX` register, removes the stack frame and returns to the caller. The return instruction is again preceded by a safepoint.

```
009F8BBF  mov         eax, esi
009F8BC1  mov         esp, ebp
009F8BC3  pop         ebp
009F8BC4  test        dword ptr [380100h], eax
009F8BCA  ret
```

2.4.4 Focus on the Common Case

Complex operations can often be split into a common and an uncommon case. For example, the allocation of a new object usually succeeds. Rarely, allocation fails because the VM runs out of memory. In this case, the garbage collector needs to be invoked.

A central design guideline in the client compiler is the *focus on the common case* [42]. Since the common case is more frequent than the uncommon one, it should be the fast path for performance reasons, even at the cost of a more complicated slow path.

Neither the HIR nor the LIR contain instructions or control flow for slow paths. This keeps the intermediate representations small and allows a faster processing. Operations with an uncommon case can still be identified during code generation. Then, the compiler emits a branch to the slow path, which calls a run-time routine in most cases.

In order to improve the instruction cache usage for the common case, the slow path should be outside the method's regular code. While machine code for the common case is immediately appended to the code buffer, *code stubs* for slow paths are collected in a list first. At the end of the method, the list is traversed and code stubs are emitted.

2.4.5 Debugging Information

During the execution of machine code, it is possible that the Java HotSpot VM runs out of memory or needs to continue the execution of a method in the interpreter. For this reason, the VM requires additional information about a compiled method, so-called *debugging information*:

- *Oop maps* specify registers and memory cells that may contain an object pointer at a particular program point (oop stands for *ordinary object pointer*). During garbage collection, these locations are treated as root pointers and get updated if the referenced object is moved.

- The *list of dependent methods* contains methods that are inlined based on class hierarchy analysis. If a class is loaded later that invalidates the inlining, the VM deoptimizes the method and continues its execution in the interpreter.
- *Scope entries* describe which local variables are stored in which registers or memory locations. This facilitates the creation of a stack frame for the interpreter in the case of deoptimization.

Deoptimization and garbage collection can run only after all threads were stopped at the nearest safepoint. Therefore, debugging information is generated for all safepoints of a method, i.e. backward branches, method calls, return instructions and operations that may throw an exception.

3 Escape Analysis

If an object that was created in a method is assigned to a global variable or to the field of a heap object (commonly referred to as *non-local variables* below), passed as a parameter to a method or returned from a method, its lifetime exceeds the scope in which it was created. Such an object is said to *escape* its scope. Knowing which objects escape their scope allows the compiler to perform more aggressive optimizations.

This chapter deals with the identification of objects that do not escape. After a short motivation for such an analysis, we examine when and how objects may escape. Then the concepts of an intraprocedural and interprocedural analysis are explained. Equi-escape sets are introduced for the efficient propagation of escape states among related objects. The final section illustrates our escape analysis algorithm by means of an example. The actual optimizations, such as scalar replacement, stack allocation and synchronization removal, are subject of the subsequent chapters.

The analysis described here was implemented in a research version of the Java HotSpot client compiler. It operates on the SSA-based high-level intermediate representation (HIR) and is as flow-sensitive as the SSA form itself. All outputs and findings stem from this implementation.

3.1 Motivation

The use of a garbage collector instead of explicit object deallocation avoids memory leaks and dangling pointers, but usually leads to a considerable drop in performance at run time. A non-concurrent garbage collector has to stop all threads at safepoints before it is allowed to clean up memory. In the context of generational garbage collection, every assignment of a reference to a field is associated with a write barrier for card marking (see Section 4.2).

Garbage collection has to deal also with objects whose last use could be determined statically from the source code. Consider the following method which computes the area of a circle with the specified radius:

```
public double circularArea(int r) {
  Circle c = new Circle(r);
  return c.area();
}
```

Every time the method is called, an object for the circle is allocated on the heap, but used only for the computation of the area and then never accessed again. The object escapes the method because it is passed as the receiver to the constructor and to the `area` method. After inlining, the compiler faces the following code:

```
public double circularArea(int r) {
  Circle c = new Circle();
  c.r = r;
  return c.r * c.r * Math.PI;
}
```

Now the object is not passed to a method and not stored in a static field. In other words, it can provably be accessed only from within this particular method and thus does not escape the method. By eliminating the allocation and replacing every field access by the value most recently assigned to the field, the method can be simplified to:

```
public double circularArea(int r) {
  return r * r * Math.PI;
}
```

Even if an object cannot be eliminated completely as in the example above, it can be allocated on the stack if it is accessed only by the creating method and its callees and never stored in a non-local variable. Memory for such objects is released implicitly when the stack frame is removed at the end of the method. As fewer objects are allocated on the heap, the garbage collector runs less frequently and requires less time in total.

If multiple threads access a shared object, they need to be synchronized with a locking mechanism. As long as one thread owns the object's lock, other threads must wait for the lock to be released. The management of the lock and the queue of waiting threads involves some overhead. If it can be guaranteed that the object is accessible only by a single thread, synchronization on the object is dispensable and may be removed.

Escape analysis is by no means intended to encourage or compensate bad programming style. In the synthetic example above, the programmer already ought to have recognized the opportunity to save an allocation, but the structure of real-world programs is normally much more complicated. The programmer is not always able to

- examine the complete source code of a program,
- inline methods without introducing redundant or inconsistent code,
- eliminate objects needed for scalability or reusability purposes, or to
- ease synchronization in classes ready for multi-threaded systems.

The JVM has in fact a wider potential for aggressive optimizations than the programmer. It can act on optimistic assumptions and recompile a method if the assumptions finally turn out to be wrong. However, sophisticated analyses are necessary even to reveal what seems obvious to a human observer.

3.2 Definition of Terms

Escape analysis requires an appropriate infrastructure within the compiler, which includes an extension of the HIR and the introduction of new types and data structures. This section deals with the different escape states of objects, how equi-escape sets are used for the propagation of escape states among related objects, and how an object depends on other objects that reference it.

3.2.1 Escape States

To optimize the allocation and synchronization of objects, the compiler has to know whether an object allocated in a method can be accessed from outside the method. This knowledge is derived from the intermediate representation via escape analysis. If an object can be accessed by other methods or threads, it is said to *escape* the current method or the current thread, respectively. In the context of our work, possible escape states are:

- `NoEscape`: The object does not escape at all, i.e. it is accessible only from within the current method. The compiler is allowed to replace its fields by scalar variables and in turn to eliminate the allocation. Objects with this escape state are called *method-local*.
- `MethodEscape`: The object escapes the creating method, but does not escape the creating thread, e.g. because it is passed to a callee which does not let the parameter escape. It is possible to allocate the object on the stack

and eliminate any synchronization on it. Objects with this escape state are called *thread-local*.

- `GlobalEscape`: The object escapes globally, typically because it is assigned to a non-local variable and can thus be referenced by other threads. No optimizations are possible and the object must be allocated on the heap.

In the HIR, an object is represented by its allocation instruction. The escape state is initialized with `NoEscape` and updated along with the construction of the HIR. It can only increase towards `GlobalEscape` but never decrease over time. The escape state of an allocation site can be regarded as the maximum possible escape state of objects created at this site. Besides, we say that a variable `p` escapes if the object referenced by `p` escapes.

Type casts, which take an object as input and represent an object themselves, share the escape state of their argument. The escape state of a phi function results from the maximum escape state of its operands, and each operand in turn adopts the escape state of the phi function. For convenience purposes, all instructions that do not represent an object value are treated as escaping globally.

3.2.2 Equi-Escape Sets

The escape states of a phi function's operands depend on each other. Assume four allocation instructions `a0` to `a3` and the three phi functions

$\Phi_0 = [\texttt{a0}, \texttt{a1}]$,
$\Phi_1 = [\texttt{a1}, \texttt{a2}]$, and
$\Phi_2 = [\Phi_1, \texttt{a3}]$.

Figure 3.1 shows the corresponding data flow. If some of the objects escaped, it would not make sense to treat the others as non-escaping and thus to replace their fields by scalar variables, because all operands represent the same variable. Instead of determining and propagating the maximum escape state, each phi function and its operands are inserted into an *equi-escape set* (EES). All elements in the set share the same escape state, namely the maximum of the elements' original states.

EES are implemented as instruction trees based on the *union-find algorithm* [81]. For this purpose, instructions define a parent field to point to another instruction in the same set. Each connected component corresponds to a different set. A special role is assigned to the root of the instruction tree. It acts as a *representative* for the set and specifies the escape state of the set's elements.

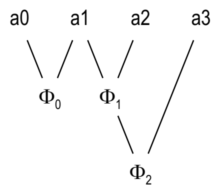


Figure 3.1: Phi functions and their operands

Whenever an object is inserted into a set, its parent field is adjusted to point to the root of the corresponding instruction tree. If the object is already part of another set, its representative is linked to the root instead. Additionally, if the inserted element has a higher escape state than the root, the escape state of the root has to be adjusted because in the future only this escape state is decisive for all elements.

Figure 3.2 visualizes the individual steps on the basis of the above example. Initially, the objects are not connected among each other. The EES of the first phi function and its operands forms a three-node tree. Assume that the phi function is the root of the tree. This choice is arbitrary; any operand could be the root just as well. The second phi function has an operand `a1` that is already contained in another set, so `a1`'s representative Φ_0 is linked to Φ_1. After the third phi function has been processed, all objects and phi functions turn out to be elements of one large set.

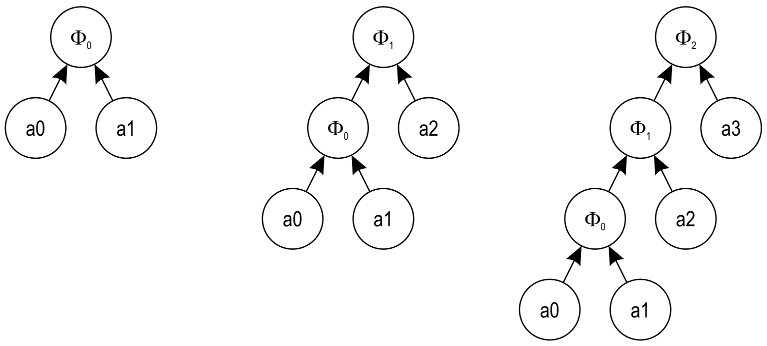


Figure 3.2: Stepwise creation of equi-escape sets

To detect whether two objects belong to the same set, their representatives are compared for equality. Starting from one of the objects, the instruction tree is walked up until its root is reached. This algorithm has a poor worst-case performance because the tree can be degenerated. The average time required to determine the representative is proportional to the average path length from every node to the root.

Based on this insight, we make another pass through the tree after the root has been found and set the parent field of each node encountered along the way to point to the root. This is called *path compression* [81]. The instruction tree gets flattened over time and future operations on the set execute faster. Figure 3.3 shows the result of path compression when applied to node `a0`.

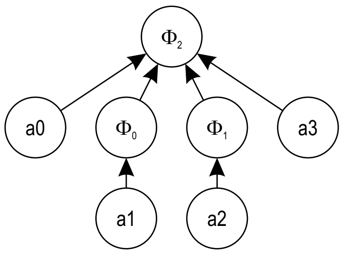


Figure 3.3: Instruction tree after path compression

3.2.3 Referential Dependency

Consider an object `A` referencing two objects `B` and `C`. If `A` is assigned to a non-local variable, not only itself becomes accessible by other threads but also `B` and `C`. Whenever the escape state of `A` changes, it must be propagated to the escape states of `B` and `C`, but not the other way round.

The dependencies are unidirectional and therefore cannot be modeled with an EES. It is not enough to propagate `A`'s escape state to the value of a field when it is assigned, because `A` may escape afterwards. Rather, `A` must keep a list of all objects possibly referenced by it during its lifetime. Every time an object is assigned to a field of `A`, it is added to this list. When the escape state of `A` changes, the list is traversed and the new escape state is propagated to the elements of the list.

Analogous to a field assignment, every object assigned to an array element is inserted into the array's list of referenced objects. There is, however, a subtle difference. The escape state of an object loaded from a field does not affect objects stored in other fields. For arrays, however, it is often not possible to prove the equality of two index calculations at compile time and thus to determine which element is loaded. If the object that represents the result of such an array load escapes, all objects referenced by the array must conservatively be treated as escaping. For example, in the method below we are not able to determine at compile time whether the rectangle or the circle will be returned, so we have to treat both figures as globally escaping.

```
public Figure getRandomFigure() {
  Figure[] f = new Figure[2];
  f[0] = new Rectangle();
  f[1] = new Circle();
  int i = random.nextInt(2);
  return f[i];
}
```

When objects are inserted into an EES, their lists of referenced objects are merged and stored with the representative. As soon as a set element escapes, the escape states of all objects in the list are changed together with the escape state of the representative. In other words, if one object escapes for some reason, not only the other objects in the same EES escape simultaneously, but also any object that is referenced by at least one of the set's elements.

In Figure 3.4, solid arrows point to parent objects in an EES and dashed arrows to lists of dependent objects. If `r` escapes, the escape states of its representative `q` and the dependent objects `q.f` and `r.f` are adjusted. Since `r.f` is contained in an EES itself, the new escape state is propagated to `p` and `p.f` as well.

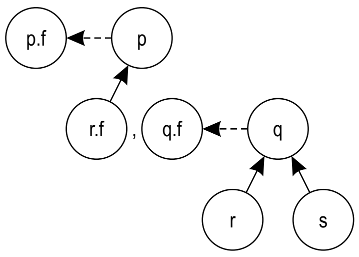


Figure 3.4: Objects and their fields in equi-escape sets

It is also possible that objects reference each other cyclically as in the example below. In this case, the state update terminates when all objects have been processed. This is guaranteed by two precautions: At first, the escape state of an object is changed if and only if the new escape state is higher than the old one. Secondly, we change the escape state of an object before we traverse its list of referenced objects.

```
T p = new T();
p.f = new T();
p.f.f = p;
```

We know that before and after escape state propagation every field has an equal or higher escape state than the containing object. So when we reach an object whose escape state needs not be changed, we do not iterate over its list of referenced objects either. Even if the dependency graph contains cycles, at some point in time all escape states have been adjusted and the state update terminates.

3.3 Intraprocedural Analysis

Escape analysis can almost completely be performed during the construction of the HIR. Initially, all objects start as non-escaping. When the compiler parses an instruction that might cause an object to escape, it adjusts the escape state of the instruction `p` representing the object. The method `p.ensure_escape(state)` sets `p.state` to `max(p.state, state)` and propagates the new escape state to the representative of the EES and to all referenced objects. The following paragraphs describe how certain instructions affect the escape state of objects:

```
p = new T()                        [ p.ensure_escape(GlobalEscape) ]
```
If the class `T` defines a finalizer, `p` must be allocated on the heap so that the finalizer can be called before the garbage collector reclaims the object's memory. If `T` has not been loaded yet, information about its fields is not available and the compiler must conservatively assume that `T` objects reside on the heap. The escape state of `p` is then set to `GlobalEscape`. In the majority of cases, however, the escape state remains `NoEscape`.

```
a = new T[n]                       [ a.ensure_escape(GlobalEscape) ]
```
The escape state of a newly allocated array `a` is `NoEscape` only if the specified length `n` is a constant. Arrays of variable length are never replaced by scalars, because the compiler cannot guarantee that the array is accessed only with valid indices and that no exception occurs at run time. They cannot be allocated on the stack either, because the maximum size of the stack frame must be known at compile time. These arrays are marked as `GlobalEscape`.

```
T.sf = p                               p.ensure_escape(GlobalEscape)
```
Every object `p` stored into a static field is marked as `GlobalEscape` because from this point on other methods and threads may access it. In other words, we cannot state anything about the lifetime of `p` or the scope where it will be referenced.

```
q.f = p                              p.ensure_escape(q.get_escape())
```
As soon as the object `p` is assigned to an instance field, it inherits the escape state of `q` provided that it is higher than its own. If other methods or threads are able to obtain a reference to `q`, they can also access `p`. Additionally, `p` is inserted into the list of dependents of `q` so that its state gets updated in case `q` escapes later on. Further actions are required in the context of scalar replacement to store the field value for future load instructions, but these are described in the next chapter.

`a[i] = p` `p.ensure_escape(a.get_escape())`
The storage of an object `p` into an array `a` has the same effects as a field assignment. `p` inherits the escape state of `a` and is added to the array's list of referenced objects.

`q = a[i]` `a.ensure_escape(MethodEscape)`
An array `a` that is accessed with a non-constant index is marked as `MethodEscape`, because the compiler does not know which element is loaded and thus cannot replace the array elements by scalars any longer. Therefore, the array elements are added to the dependency list of the object that represents the result of the array load. If the object escapes globally later on, all array elements are marked as `GlobalEscape`.

`p == q` `{p,q}.ensure_escape(MethodEscape)`
If two objects `p` and `q` are compared, they must both exist at least on the stack, so their escape state is raised to `MethodEscape`. This is also true for an object compared with `null`. Sometimes the compiler could determine whether the condition will be always true or always false, but this is rarely possible because of phi functions.

`(T) p` `p.ensure_escape(MethodEscape)`
Even if `p` does not escape, the object and the type cast cannot be eliminated. The reason is that the cast might fail and then an exception has to be thrown. Therefore, `p` is marked as `MethodEscape`. Only if the compiler can prove statically that the object is always of the requested type, the cast is eliminated and the escape state remains untouched.

`p.foo(q)` `{p,q}.ensure_escape(GlobalEscape)`
A purely intraprocedural analysis is not able to predict what happens to the parameters of a method invocation. In the worst case, the callee assigns some of them to non-local variables. So we mark all actual parameters as `GlobalEscape`. The receiver `p` of the call is treated in just the same way as any other parameter. This rather conservative approach is refined in an interprocedural analysis, where the escape states of formal parameters are computed at a method's compilation and reinspected at its call sites.

`return p` `p.ensure_escape(GlobalEscape)`
Values returned from the current method must not be allocated on the stack because they can still be accessed by the caller when the frame of the callee has already been released. For this reason, the return value `p` is marked as `GlobalEscape`. Interprocedural analysis opens up an optimization if a formal parameter is returned which does not escape otherwise. In this case the actual parameter may be allocated on the caller's stack, provided that certain constraints are fulfilled (see Section 3.4.1).

`throw p` `p.ensure_escape(GlobalEscape)`
Exception objects are always treated as escaping globally. It is nearly impossible to determine statically which exception handler will get invoked in every case. Besides, exceptions are normally used to model exceptional cases which occur infrequently. Therefore, it does not pay off to optimize them.

3.4 Interprocedural Analysis

The gain of an intraprocedural analysis is fairly limited because objects are normally not used as pure data structures within a single method. Instead, they act as parameters or receivers of at least one method call, namely the object's constructor. Inlining thus plays a major role in the improvement of escape analysis, but inordinate inlining rapidly fills the code buffer and slows down the compiler. Besides, some methods simply must not be inlined.

Therefore, interprocedural techniques are vital to tap the full potential of escape analysis. They determine which objects escape neither from the method that allocates them nor from any of its callees. The results help the compiler to decide which parameters may be allocated on the stack and which methods are worth to be inlined.

3.4.1 Escape of Parameters

The compilation of a method produces escape information not only for local variables, but also for the method's formal parameters. In an interprocedural analysis, the compiler does not discard this information but stores it with the method descriptor in a compact form so that the escape state of actual parameters can be adjusted when the method is called later on.

Apart from that, interprocedural analysis supports the compiler in making better inlining decisions. Basically not every method that can be bound statically is also inlined because the increase in compilation time may outweigh the gain in run time. Some of the restrictions, such as the maximum code size of methods to be inlined, are based on heuristics and may be weakened if we knew that inlining would prevent some objects passed as parameters from escaping.

In the course of HIR construction, formal parameters are treated the same way as objects allocated within the method. They start as non-escaping values and may be inserted into EES or stored into fields of other objects. After completion of the HIR and before scalar replacement, their escape state is encoded by two integer values which act as bit vectors and store which parameters do not escape or can be allocated on the stack, respectively. Objects assigned to fields of formal parameters are treated as escaping globally.

In the example below, the first parameter `a0` escapes globally because it is assigned to the static field `sf`, while `a1` is compared against `null` and thus marked as stack-allocatable. The last parameter `a2` is not used at all and remains non-escaping. The escape states are encoded by two bit vectors 001 and 011 denoting the set of method-local and thread-local parameters, respectively.

```
static boolean foo(Object a0, Object a1, Object a2) {
  sf = a0;
  return (a1 != null);
}
```

At each call site, the compiler tries to retrieve the escape information from the callee's method descriptor in order to decide whether to inline the method or not and to adjust the escape state of the actual parameters. Of course, it is a prerequisite that the method to be invoked can be determined at compile time and no dynamic method binding is necessary.

Apart from static and final callees, this is possible if class hierarchy analysis determines that currently only one suitable method exists. If a class is loaded later that provides another method with the same signature, maybe the wrong implementation was inlined. In this case, the generated machine code is invalidated and the method gets recompiled (see Chapter 6).

If the code size of the callee exceeds the maximum inline size within a certain limit, the escape information is used to decide if inlining is desirable nevertheless. It turned out to be a good heuristic to inline methods up to twice as large as the normal threshold if there is at least one parameter that does neither escape the caller nor the callee. The omission of inlining would enforce an allocation of the parameter on the stack or on the heap. If the object happens to escape in the caller after the method invocation, the premature inlining did not facilitate scalar replacement but still saved the dispatching costs.

For inlining decisions it is only relevant if formal parameters are method-local and not if they are thread-local. The latter information is used if a method cannot be inlined. Although a method call instruction has to be generated, not all actual parameters must necessarily be treated as escaping globally. If a formal parameter remains thread-local in the callee, the actual parameter can be allocated on the caller's stack frame. This is achieved by marking an actual parameter as stack-allocatable if the corresponding formal parameter does not escape globally in the callee.

Normally, each object returned from a method is marked as escaping globally. If, however, the receiver or a parameter is returned and does not escape otherwise as in the following example, the actual parameter can still be allocated on the caller's stack. The method is then provided with a pointer to the object on the stack, and exactly this pointer is returned by the callee again.

```
Object foo() {
  return this;
}
```

Nevertheless precaution is necessary because an implicit dependency between the actual parameter and the return value gets introduced. The `Invoke` instruction, which represents not only the method call but also its result in the HIR, is possibly just an alias of any parameter that the callee returns. If it is assigned to a non-local variable, the parameter must no longer be allocated on the stack either. For this reason, the method descriptor also records which formal parameters might be returned, and at each call site the corresponding actual parameters are inserted into an EES with the `Invoke` instruction.

Finally, interprocedural escape analysis records the maximum escape state of all objects that might be used as return values. If a caller synchronizes on the actual return value and the compiler knows from interprocedural escape information that none of the objects returned from the callee escapes globally, it can safely remove the synchronization.

3.4.2 Analysis of Bytecodes

The Java HotSpot VM does not immediately compile every method as soon as it is called, because compilation time adds to run time. Instead, it interprets a method at first and counts how often it is invoked. When the *invocation counter* reaches a certain threshold, the method is scheduled for compilation. This way, only the most frequently called methods get compiled. Consider the following Java method:

```
static Object foo(Object p, Object q) {
  if (p == q) {
    return p;
  }
  sf = q;
  return null;
}
```

Information about the escape states of `p` and `q` is generated only during the compilation of `foo`. If the compiler parses a call to `foo` before it was executed often enough to be compiled, this information is not available. It is neither possible nor reasonable to initiate the compilation of `foo` in this situation, but without interprocedural escape information the actual parameters must be treated as escaping globally.

Therefore, the compiler performs a fuzzy analysis on the bytecodes of `foo` to get provisional escape information for the parameters. It considers each basic block separately and records which parameters escape. This is more conservative but faster to compute, because no phi functions are required and control flow can be ignored. A parameter is considered as escaping as soon as it is

- stored in a local variable (because local variables are not tracked),
- assigned to a field, regardless of whether it is a static field or not,
- stored into an array,
- returned from the method, or
- thrown as an exception.

Escape information is gained via an abstract interpretation of the bytecodes, basic block by basic block. The bytecodes for the method `foo` are shown below. The operand stack is modeled as a stack of integers (see Figure 3.5). Every time an object parameter is loaded, its index is pushed onto the stack. When a bytecode is processed that operates on an object, an index is popped from the stack and the escape state of the corresponding parameter is adjusted. The analysis is aborted as soon as all parameters are seen to escape.

```
 0: aload_0
 1: aload_1
 2: if_acmpne 7
 5: aload_0
 6: areturn
 7: aload_1
 8: putstatic #2
11: aconst_null
12: areturn
```

When the analyzer parses the `if_acmpne` bytecode, it determines that both parameters must be allocated on the stack, i.e. that inlining the method will not benefit scalar replacement. It also finds out that `p` is returned from the method and that `q` must be allocated on the heap because it is assigned to a static field. When the bytecode `aconst_null` is reached, -1 is pushed onto the stack. It acts as a placeholder for constants, primitive values or objects allocated within the method. Operations on such values can safely be ignored.

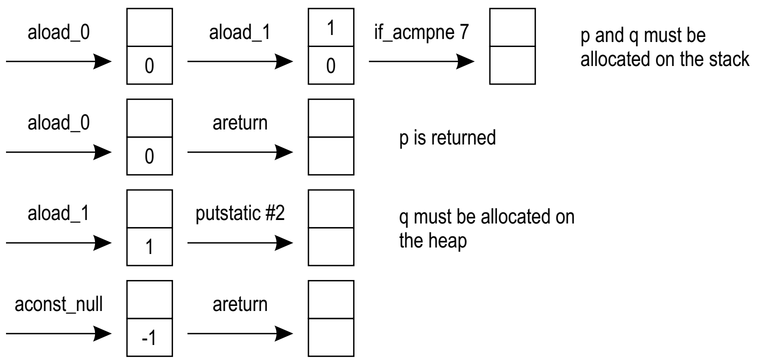


Figure 3.5: Parameters on the stack during bytecode analysis

Usually, the operand stack is empty after each Java statement. If, however, an object parameter is still on the stack at the end of a basic block, it is marked as escaping globally. Since the analysis does not take control flow into account, it does not know which basic block is executed next and thus what happens to the parameter on the stack.

Local variables that are assigned a new value are marked as dirty. If a dirty variable is loaded and returned, the compiler conservatively assumes that a globally escaping object is returned and records that synchronization on the method result must not be eliminated in the caller.

When the bytecode analyzer parses a method call, it uses interprocedural escape information to adjust the escape states of parameters of the currently analyzed method that are forwarded to the callee. If no escape information is available yet, the analyzer recursively applies itself to the callee. As for inlining, a prerequisite is that the callee can be bound statically and that its size does not exceed a certain threshold. Otherwise, the forwarded parameters are regarded as escaping.

The findings of the bytecode analysis are stored in the method descriptor and used to adjust the escape state of actual parameters. When the method is compiled later, the provisional escape information is replaced by a more precise one. Since the compiler is less conservative than the bytecode analyzer, the escape state of a parameter can only change from global escaping to stack-allocatable and not the other way round.

3.5 Example

This section summarizes the complete process of escape analysis for a larger example. The method below computes the factorial of the specified parameter, but boxes the result and the loop counter into `Int` objects:

```
static Int factorial(Int x) {
  Int p = new Int(1);
  Int i = new Int(1);
  while (i.val <= x.val) {
    p = new Int(p.val * i.val);
    i = new Int(i.val + 1);
  }
  return p;
}
```

The first block B0 of the method allocates two Int objects for the initial values of the result and the loop counter. Since the method was interpreted several times before compilation, the class Int is already loaded. Besides, we assume that the class does not have a finalizer. The constructor calls, which set the val fields, are inlined. At the end of the block, the two allocation sites a5 and a15 are still marked as non-escaping.

```
B0 [0, 17] -> B1
__bci__use__tid____instr_______________________________
. 0    2    a5     new instance Int
  4    1    i6     1
. 6    0    i13    a5._8 := i6
. 9    2    a15    new instance Int
. 6    0    i23    a15._8 := i6
. 17   0     25    goto B1
```

The subsequent block B1 represents the loop header that checks the loop condition. It is the target of a backward branch and thus has two predecessors. Phi functions are created for the result and the counter object. There is no phi function for the first local variable which stores the parameter a4 because it is not modified within the loop. The block compares the fields i.val and x.val and then jumps either to B3 or B2.

```
B1 [18, 26] -> B3 B2
Locals:
 0  a4                    // x
 1  a26 [a5 a33]          // p
 2  a27 [a15 a45]         // i
__bci__use__tid____instr_______________________________
. 19   1    i30    a27._8
. 23   1    i31    a4._8
. 26   0     32    if i30 > i31 then B3 else B2
```

The loop body B2 performs the actual multiplication and increments the loop counter. Within each iteration, two new integer objects are allocated that encapsulate the updated values and make up the second operands of the phi functions created above. Again, both constructor calls are inlined. Neither `a33` nor `a45` escape so far.

```
B2 [29, 60] -> B1
__bci__use__tid____instr_________________________________
. 29   2    a33    new instance Int
. 34   1    i34    a26._8
. 38   1    i35    a27._8
. 41   1    i36    i34 * i35
. 6    0    i43    a33._8 := i36
. 46   2    a45    new instance Int
. 51   1    i46    a27._8
  54   1    i47    1
. 55   1    i48    i46 + i47
. 6    0    i55    a45._8 := i48
. 60   0     57    goto B1
```

Finally, `a26` is returned as the factorial number of the parameter `a4`. When the return instruction gets parsed, the phi function `a26` is marked as escaping globally. Since it is not part of any EES yet, no other values are affected.

```
B3 [63, 64]
__bci__use__tid____instr_________________________________
. 64   0    a58    areturn a26
```

At the end of HIR construction, we insert the phi functions into an EES with their operands. In this example two independent instruction trees arise as shown in Figure 3.6. Each root records the maximum escape state of the set's elements. Although the escape states of `a5` and `a33` remain untouched, both values are treated as escaping globally because they point to `a26`. The phi function `a27` in contrast represents a set of non-escaping values.

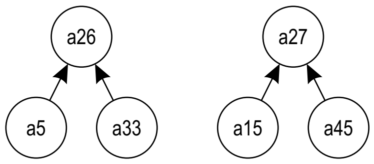


Figure 3.6: Equi-escape sets for the factorial method

Summing up, escape analysis identifies the objects for the loop counter (`a15`, `a27`, `a45`) as non-escaping. They are candidates for scalar replacement, whereas the objects for the intermediate results (`a5`, `a26`, `a33`) possibly escape and must be allocated on the heap.

Throughout the whole process of building the HIR, the formal parameter `a4` remained non-escaping. On the one hand, this means that if the compiler decides to inline the method despite its size of 65 bytes, it might save the allocation of the actual parameter. On the other hand, even if the method is not inlined, the actual parameter can probably be allocated on the stack frame of the caller. This information is stored with the descriptor of the method `factorial` and will be examined by the graph builder when it reaches one of the method's invocations.

Note that the HIR still contains instructions to allocate the non-escaping objects as well as to store and load their fields, although the integer values could be used directly instead. The instructions are removed from the HIR by scalar replacement, which is the topic of the next chapter.

4 Scalar Replacement

Escape analysis is no optimization on its own, but only provides information about the scope in which objects are accessed. The compiler can then use the results to identify method-local objects, eliminate their allocation and replace their fields with scalar variables. This is called *scalar replacement of fields*.

The objective of this optimization is to save the costs of object allocation and to relieve the garbage collector. The use of a scalar variable instead of a field eliminates one memory access because the object reference needs not be loaded. If the back end manages to keep the field in a register, no memory access is necessary at all. Moreover, a constant field value can often be directly encoded in machine instructions.

This chapter describes actions and data structures required for scalar replacement. It deals with potential problems and how they were solved in our implementation. Besides, we try to give an idea of the gain that can be achieved by the elimination of objects. The actual performance improvement for various applications and benchmarks is presented in Chapter 7.

4.1 Overview

Since scalar replacement depends on the findings of escape analysis, which are available fairly late, it is split into two phases. The first phase does not change the instructions of the HIR and can be performed in parallel with escape analysis and the construction of the intermediate representation. The graph builder

- maintains a data structure to remember the values most recently assigned to fields,
- creates special phi functions for fields at join points in the control flow, and
- annotates each instruction that loads a field with the field's current value if available.

Even if the graph builder is able to determine the current value of a field when it is loaded, this does not mean that the field can be replaced by a scalar because the object may still escape below. For this reason, the actual substitution of load instructions by the fields' values is delayed to the second phase, when the HIR is complete and we know for sure which objects escape and which do not.

Scalar replacement is mostly implemented in the front end of the just-in-time compiler. The translation of the HIR to the LIR and the generation of machine code remains nearly untouched, except that the back end includes field values in the traversal of all variables, resolves phi functions for fields, and ignores eliminated object allocations.

4.2 Object-Related Costs

To get an impression of the costs associated with objects in the JVM, we examine first which machine instructions are generated for their allocation. The single line of Java source code

```
T p = new T();
```

represents both the reservation of memory on the heap and the invocation of the appropriate constructor. Accordingly, the just-in-time compiler generates the following two HIR instructions:

```
__bci__use__tid____instr________________________________
. 0    2    a1     new instance T
. 4    0     2     a1.invokespecial()
```

Constructors initialize the instance fields of newly created objects. They can be bound statically and are usually small or even empty. The compiler is often able to inline them together with the constructors of the superclasses. Therefore, they are not examined more closely here.

On an Intel processor [51], the `new instance` instruction is translated to the machine code shown in Listing 4.1. To facilitate a fast memory allocation, the VM maintains pointers to the top and the end of a free area on the heap. In this example they are stored at the addresses 909160h and 909130h, respectively. The top pointer is copied to the `EAX` register and then incremented by the object size via the `lea` instruction. If the result in `ECX` is less than the original pointer in `EAX` due to a range overflow or greater than the end of the free area, the JVM ran out of memory and allocation is delegated to a run-time routine which invokes the garbage collector.

```
00001000  mov          eax, dword ptr [909160h]
00001006  lea          ecx, [eax+20h]
00001009  cmp          ecx, eax
0000100B  jb           0000105F
00001011  cmp          ecx, dword ptr [909130h]
00001017  ja           0000105F
0000101D  cmpxchg      dword ptr [909160h], ecx
00001024  jne          00001000
...
0000105F  // call of run-time routine
```

Listing 4.1: Assembler code for memory allocation

The `ECX` register now specifies the new top pointer, while `EAX` still stores a copy of the old one. The `cmpxchg` instruction uses `EAX` as an implicit operand. Before it writes the `ECX` register back to memory, it compares `EAX` with the value at address 909160h to ensure that no other thread has modified the top pointer meanwhile. If the check fails, `ECX` is not written to memory and allocation has to start over.

In a multi-processor environment, the `cmpxchg` instruction must be executed atomically and is thus preceded by a lock prefix. Therefore, the latest version of the JVM creates new objects in a *thread-local allocation buffer* (TLAB) so that multiple threads do not interfere. This removes the atomic operation, but introduces the cost of accessing a data structure for the current thread.

Now the `EAX` register points to the newly created object. The object header consists of four bytes and is initialized with the constant 1 denoting an unlocked object (see Listing 4.2). The next four bytes are copied from the `EDX` register and store a pointer to the class `T`. Memory for instance fields is cleared out in a loop because the Java language specification requires fields to be initialized with default values [40] and the garbage collector presumes that unassigned references are null. The `ECX` register is set to 0 via an `xor` instruction, and `EDI` acts as the loop counter. Each iteration copies the `ECX` register to two consecutive memory words. If the object size is no multiple of double words, an extra move for the remaining word is generated.

Scalar replacement does not only eliminate the allocation, but also any field access for non-escaping objects. Every load or store of a field normally requires one or two move instructions, depending on whether the address of the object is already in a register or not. The store of a field that holds a pointer is even more expensive because it is associated with a *write barrier*.

```
0000102A  mov          dword ptr [eax], 1
00001030  mov          dword ptr [eax+4], edx
00001033  xor          ecx, ecx
00001035  mov          edi, 3
0000103A  mov          dword ptr [eax+edi*8+4], ecx
0000103E  mov          dword ptr [eax+edi*8], ecx
00001041  dec          edi
00001042  jne          0000103A
```

Listing 4.2: Assembler code for object initialization

Write barriers are necessary because pointers from old objects to young objects must be entered in a *remembered set*, so that the young generation can be collected without scanning the old one. The Java HotSpot VM uses a technique called *card marking* [50]. The heap is partitioned into cards of 512 bytes. There is a byte array with one byte for every card. When a pointer is assigned to a heap location, the associated write barrier marks the modified card in the byte array. When the garbage collector runs the next time, it inspects all marked cards for cross-generation pointers and enters them into the remembered set. Listing 4.3 shows how the write barrier is implemented [47].

```
00001048  mov          dword ptr [eax+8], esi
0000104B  shr          eax, 9
0000104E  mov          byte ptr [eax+35E380h], 0
```

Listing 4.3: Assembler code for a field store with write barrier

Under the assumption that `EAX` still contains the object address, only one move instruction is necessary to store the value from the `ESI` register to the field with the offset 8. The card index is calculated via a right-shift of the object address. Then the corresponding byte in the vector at address 35E380h is set to 0 denoting that the card has been modified. Scalar replacement eliminates the field access together with the write barrier and potentially reduces the number of cards to be scanned for pointers.

As a third point, the garbage collector itself benefits from the removal of allocation sites because it has less objects to process and may run less frequently. Candidates for scalar replacement are usually short-lived objects, and although more long-lived objects accumulate if the intervals between collections are longer, some of them probably die again before the garbage collector is invoked. Therefore, less objects get copied to the old space, which reduces the cost of the old generation's collection.

Some of the performance gains mentioned here also apply to objects allocated on the stack, e.g. the object address needs not be loaded because fields can be accessed relative to the stack base pointer `EBP`. Besides the garbage collector does not have to deal with objects on the stack because their memory is automatically released at the end of the method.

4.3 Analysis of Basic Blocks

In contrast to the bytecodes of a method, the SSA-based HIR does not contain any instructions for loading or storing a local variable. They are eliminated during the abstract interpretation of the bytecodes. For this purpose, the compiler remembers the value most recently assigned to a variable in the locals array of the state object. If a bytecode operates on a local variable, a HIR instruction is generated that directly uses the value from the locals array. Bytecodes refer to local variables via indices and the number of variables for a certain method is stored in the class file.

Field values are handled in a similar way. One difference is that we do not know in advance which or how many fields are accessed within a method. Therefore, the state object is extended by a *field map* which allocates slots for field variables on demand. Every field map entry stores an object, a field offset and the assigned value. When a field is loaded, the field map is used to obtain the field's current value.

The example below shows Java code, bytecodes and HIR instructions with the locals array and the field map. Initially the field map is empty. At first, a new object is allocated and stored in a local variable. Then the constant 3 is assigned to the field `f` of the newly allocated object. Hence the field map is extended by an entry that maps object `a0` and field `f` to the HIR instruction `i1` representing the constant 3. Analogously, the constant value 0 is assigned to the field `g` so that the field map grows by another entry for `a0.g`.

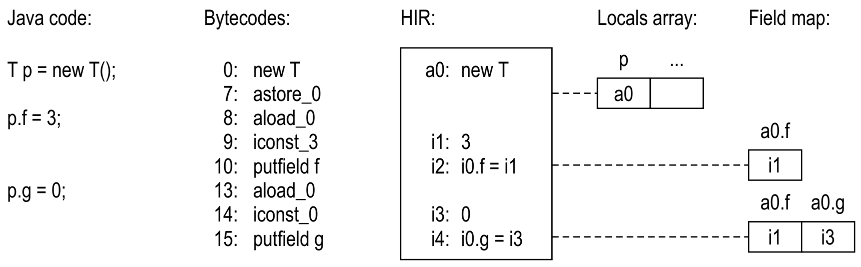


Figure 4.1: HIR and state object for a sequence of bytecodes

At the end of HIR construction, when we know which objects escape and which do not, we substitute the fields of non-escaping objects by the values stored in the field map. Provided that `a0` does not escape in the above example, the object allocation and the field initializations can be eliminated. Scalar replacement substitutes any use of `a0.f` by `i1` and any use of `a0.g` by `i3`.

Array allocations are eliminated analogously. The index of an array element is used instead of the field offset to access the field map. A prerequisite is that the array is accessed with constant indices only. Besides, the array must have a size that is known at compile time. Since the bounds check is eliminated together with an array access, the compiler must guarantee that all indices are valid and no exception will occur at run time. A non-escaping array that has a fixed size but is accessed with variable indices can be allocated on the stack.

According to the Java language specification and in contrast to local variables, instance fields may be used without a prior explicit assignment. Their default value is the type-specific interpretation of a sequence of zero valued bytes. The memory occupied by an object is cleared when it is allocated, but if the allocation of a non-escaping field is eliminated, all used field values must be explicitly visible in the HIR.

Therefore, constants for the default values are appended to the HIR and stored in the field map at an object's allocation site. In case that the object later turns out to escape, no machine code is generated for the constants because they are never used. The generation of default values can be disabled via a VM flag, which keeps the field map small but decreases the number of objects that can be replaced by scalars.

4.4 Analysis of Control Flow

Different values may be assigned to the same field in different control flow paths. For this reason, there is no global field map, but one per state object. The beginning and the end of each basic block as well as certain HIR instructions preserve a snapshot of the current state object.

At control flow branches, field values are propagated to the state objects of all successors. At join points, different field values must be merged via phi functions. This section deals with scalar replacement across block boundaries, the creation of phi functions and method inlining.

4.4.1 Phi Functions for Fields

The static single assignment form requires that uses of a value are not reached by more than one definition, even if different values are assigned to a variable in alternative control flow paths. Therefore, phi functions are introduced to merge multiple incarnations of the same variable at points where control flow joins. The back end resolves the phi functions with move instructions, so that the join block can load the actual value from a common location, regardless of the previous control flow.

The HotSpot client compiler creates phi functions for local variables at the beginning of each block with more than one predecessor. If some of them turn out to be dispensable because all of their operands are equal, they are simplified later. Local variables are accessed via their index. Therefore, the operands of a variable's phi function are located at the same index in the state objects of the predecessor blocks. A phi function only specifies the variable's index and the block it was created for. The operands can then be looked up in the locals arrays at the end of the block's predecessors (see example below).

Normally, fields are stored in memory at their assignment and reloaded before their use. They can be accessed in a uniform way across block boundaries, so phi functions are only needed for the object reference but not for the fields themselves. Scalar replacement substitutes some fields with scalar variables. These variables require phi functions, because alternative control flow paths may store the values in different registers. Consider the following piece of Java code:

```
B0: p = new T();
    p.f = 0;
    if (x > 0) {
B1:   q = p;
      q.f = 3;
    } else {
B2:   q = new T();
      q.f = 5;
    }
B3: ...
```

The compiler builds a control flow graph as shown in Figure 4.2. If there is only one edge to a block, as it is the case for `B1` and `B2`, the local variables and the field map are copied from the end of the predecessor. For blocks like `B3`, where control flow joins, phi functions are created. `a12` is a phi function for the local variable `q`. Its operands `a1` and `a8` can be found at index 1 in the locals array of the blocks `B1` and `B2`.

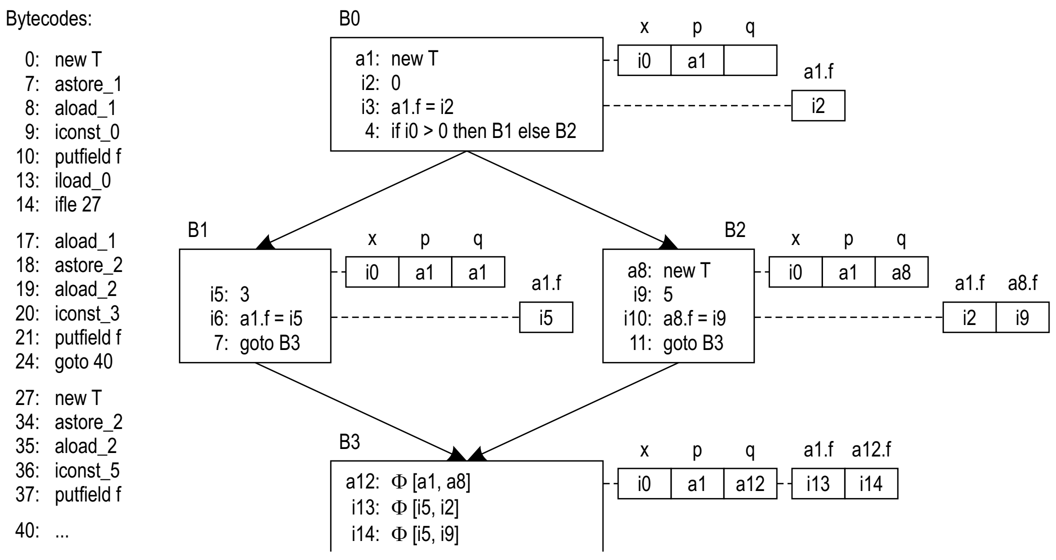


Figure 4.2: Phi functions for local variables and fields

Instruction `i14` is a phi function for the field `q.f`. In contrast to local variables, we cannot guarantee that a field is mapped to the same index throughout the method. Since the operands of the phi function for `q` are `a1` and `a8`, the value of `q.f` is `a1.f` at the end of `B1` and `a8.f` at the end of `B2`. Looking up these values in the corresponding field maps yields `i5` and `i9` as the operands of `i14`.

4.4.2 Alias Effects

An extension of the code fragment from the previous section raises a problem. The assignment of a value to the field `q.f` in the block `B3` perhaps modifies also `p.f`. We cannot determine statically whether `p.f` and `q.f` denote the same or different memory cells. They may be *aliases* or not.

```
B0: p = new T();
    p.f = 0;
    if (x > 0) {
B1:   q = p;
      q.f = 3;
    } else {
B2:   q = new T();
      q.f = 5;
    }
B3: q.f = 7;
    ... = p.f;    // unclear, which cell is read
```

In block B3 of Figure 4.2, p corresponds to a1 and q to a12. If the control flow came via B1, p and q are aliases and so are p.f and q.f. However, the field map regards a1.f and a12.f as distinct variables. If the value 7 is assigned to q.f, the compiler would store this value in the field map under the name a12.f.

The assignment to q.f should also affect p.f if the control flow came via B1 but not if it came via B2. So when p.f is read later, the compiler does not know which cell to access. Naively it would look up a1.f in the field map and would find the phi function i13 there. The operands of i13 are i5 and i2, i.e. the values 3 and 0, but the correct value of p.f is 7 or 0, respectively.

Alias effects occur only in the context of phi functions; the equality of two local variables is no problem because their values have the same name. An object has an alias if it is both the operand of a phi function and stored in a local variable, or if it is the operand of two different phi functions as in the following example:

```
B0: p = new T();
    q = p;
    if (x > 0) {
B1:   p = new T();
      q = new T();
    }
B2: q.f = 7;
    ... = p.f;    // unclear, which cell is read
```

At the beginning of B2, two phi functions are created for p and q. If x is greater than 0, p and q refer to different objects. Otherwise, p and q are aliases and so are p.f and q.f. As in the previous example, the compiler may read a wrong value from the field map when p.f is accessed afterwards.

This problem is solved in the following way: If an object is stored as the operand of different phi functions or as a phi operand and a local variable, the compiler marks it as method-escaping. Although such an object does not escape by means of a method call, it will be allocated on the stack or on the heap instead of being replaced by possibly wrong values from the field map.

4.4.3 Method Inlining

Scalar replacement and synchronization removal (see next chapter) largely benefit from method inlining. Even on short-lived objects, the constructor and usually some accessor methods are invoked. The objects escape unless the methods are inlined. Inlining embeds the body of a callee into the caller so that escape analysis is able to track what happens to the receiver and any object parameters.

If the callee cannot be inlined, the actual parameters must be allocated on the stack or on the heap even if the callee does not let them escape. The reason is that the method might also be called from a site that was forced to allocate the actual parameter on the heap. To allow a uniform object access, the callee must be provided with a pointer to the object that could not be given if the allocation were eliminated.

The primary objective of inlining is the elimination of the overhead associated with method dispatching. As far as scalar replacement is concerned, method inlining does not always pay off. If all parameters escape either in the caller or in the callee, inlining does not affect the number of objects that can be replaced by scalar variables. In its decision whether to inline a method or not, the compiler is supported by the information of the interprocedural analysis as described in Section 3.4.1.

At the start and end of each basic block and at certain instructions, copies of the state object with the current values of the local variables, the field map and the operand stack are saved. They are used for the resolution of phi functions and for the generation of debugging information. For the time of inlining, the graph builder operates on a separate state object with as many local slots as required by the callee. Every copy of the state object preserves a pointer to the caller state (see Figure 4.3).

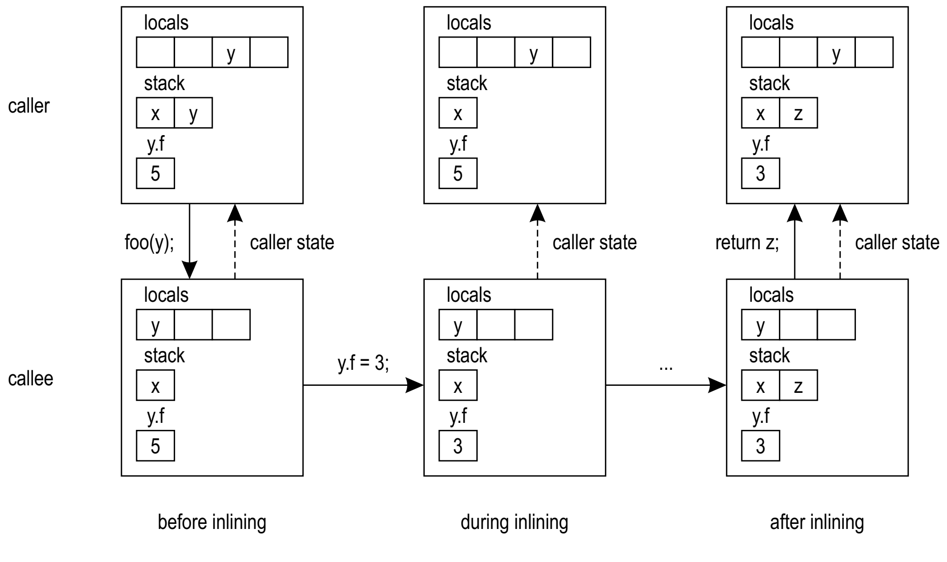


Figure 4.3: Caller and callee states during inlining

The modification of the callee's local variables or parameters does not affect the caller, but a field assignment may be visible. In Figure 4.3, the compiler takes the parameter y from the stack and stores it in the callee's locals array. The operand stack and the field map are simply copied from the caller state. Assignments to local variables and fields during inlining only change the callee state. After inlining, the compiler restores the local variables from before the method call, but keeps the possibly modified fields. At any point in the HIR, only the field values of the innermost state object are correct.

4.4.4 Selective Creation of Phi Functions

The original HotSpot client compiler did not perform a sophisticated analysis to determine where phi functions are necessary (compare [11] and [26]). Instead, it created phi functions for *all variables* as soon as a block had more than one predecessor and later simplified those whose operands turned out to be equal.

A basic block is filled with HIR instructions only after all *forward branches* to that block were processed. If no backward branch leads into a block, the phi functions can be simplified immediately before the block is parsed, because all predecessors respectively all operands are already available. The simplification of phi functions in loop header blocks, which are targeted by a backward branch, must be delayed until the complete HIR was built.

This approach makes the handling of control flow join points easy, but usually leads to a considerably high number of phi functions. Although it does not affect the quality of the machine code, because all redundant phi functions are finally simplified, it causes two problems for escape analysis:

- If the receiver of a method call in a loop is hidden by a phi function, the compiler must conservatively assume that the phi function has operands of different dynamic types. It fails to determine which method gets called and thus can neither inline the method nor retrieve interprocedural escape information for it. Therefore, the receiver and all parameters are treated as escaping globally, even if the phi function is simplified at the end.
- If an inlined method contains a loop, phi functions are created there for all formal parameters. A value that is assigned to the field `f` of a parameter `p` in the inlined method is associated with the phi function Φ_i. When the compiler continues parsing the caller, it restores the locals array before inlining which still contains the original parameter `p`. The caller accesses the field via `p.f`, but the value of the field is stored in the field map under Φ_i`.f`. Even if the parameter itself was not modified in the callee, changes of its fields get lost and it must be allocated on the stack or on the heap.

Both problems are solved by a more selective creation of phi functions. If a block is reached by forward branches only, we may delay the creation of a phi function for a variable `v` until we find out that two predecessors contribute different values for `v`.

At the beginning of a loop, phi functions must be created for all variables that are modified within the loop. However, the graph builder must decide on the creation of phi functions before it parses the loop. Therefore, the block list builder records for each basic block which local variables are modified within the block when it iterates over the bytecodes. Then the control flow graph is traversed to find out which blocks belong to a loop. Finally, all local variables modified within such a block are marked to require phi functions. The loop detection is explained here by means of the following example, which iterates over the elements of two lists in two nested loops:

```
B0: iter1 = list1.iterator();
B1: while (iter1.hasNext()) {
B2:   obj1 = iter1.next();
      iter2 = list2.iterator();
B3:   while (iter2.hasNext()) {
B4:     obj2 = iter2.next();
        // do something with obj1 and obj2
      }
    }
```

This piece of code consists of five basic blocks. Each block is associated with an initially empty bitset, where bit i denotes that the block belongs to loop i. The blocks are traversed in a depth-first manner and each block encountered along the way is marked as *active* and *visited*.

At the branch in block `B3`, we follow the first edge and reach `B1` (see Figure 4.4a). This block is already active and thus identified as a loop header (denoted by a double framed rectangle). Since it is the first loop header found, the first bit is set in the bitset of block `B1`.

Then we return to block `B3`, visit the second successor `B4`, and from there reach the active block `B3` again. It represents the second loop header, so the second bit is set (see Figure 4.4b). Since it is the only successor of `B4`, the bitset of `B3` is copied into the block `B4` (see Figure 4.4c). When we return from `B4`, we clear its active flag but remember that it has been visited to avoid processing it twice.

The basic block `B3` has more than one successor. We combine the bitsets of all successors and subtract `B3`'s own bitset. The result is a set in which only the first bit is set. It overwrites the old bitset of `B3` and gets copied into the block `B2` when we wind up (see Figure 4.4d).

Block `B1` has two successors as well. Under the assumption that the two loops are not embedded in a third one, an empty bitset is returned from the second successor. We again combine the two bitsets, subtract the one of `B1` and obtain an empty bitset. It is stored in block `B1` and returned to `B0` (see Figure 4.4e).

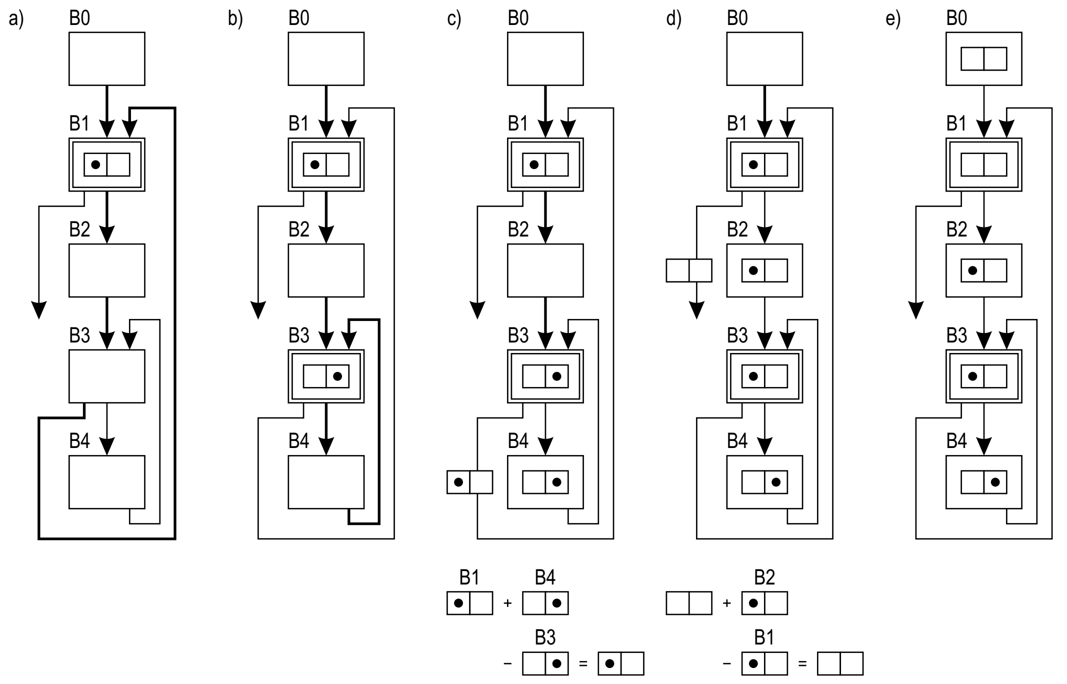


Figure 4.4: Loop detection for selective creation of phi functions

Now the complete control flow graph has been traversed. A block belongs to a loop if it is marked as a loop header *or* has a non-empty bitset. At the beginning of a loop, the compiler does not only create phi functions for variables modified within this particular loop, but for all variables modified within any loop of the method. This is more conservative but easier to implement. It is still possible that redundant phi functions are created, but they are simplified before code generation as usual.

In the example above, no phi function is created for the variable `iter1` at the beginning of `B1` any longer, because it is modified only in block `B0` which is not part of a loop. After inlining of the methods `hasNext` and `next`, the iterator can be eliminated and replaced by scalars. On the other hand, a phi function is created for `iter2`, because it is modified in block `B2` which belongs to a loop.

4.5 Adaptation of the Back End

During HIR construction, we keep track of field values under the assumption that the objects do not escape. Instructions are neither removed from the HIR nor replaced as long as the escape states are still subject to change. The actual scalar replacement is delayed until after creation of EES and alias detection.

Then the basic blocks and their instructions are traversed one more time. Phi functions that represent non-escaping objects are marked as unused to prevent the back end from generating move instructions for them. Field store and null check instructions are removed from the HIR if they operate on a non-escaping object. Instructions that load a field from such an object are substituted by the recorded value of that field.

The major part of escape analysis and scalar replacement is performed in the front end. The back end was modified only to ignore residual instructions that represent or operate on non-escaping objects as well as to take field variables into account. Adjustments were necessary where

- lifetime intervals are built,
- moves for phi functions are created,
- machine code for the computation of values is emitted, and where
- debugging information is generated.

Under certain circumstances, the VM is forced to stop the execution of a compiled method at a safepoint and fall back to interpretation. Since the interpreter does not know about scalar replacement, eliminated objects have to be recreated on-the-fly. For this purpose, the back end must generate information about the current field values for all safepoints within a method. This process is described in Chapter 6.

4.6 Verification Code

As we see from the alias problem, precautions are sometimes necessary to prevent the compiler from substituting a field access with the wrong value. To check the correctness of scalar replacement and prevent errors in future modifications, a LIR instruction for run-time assertions was implemented. If verification is enabled via a special VM flag, the compiler keeps track of field values but refrains from scalar replacement. Whenever a field is loaded from memory, the result is compared against the value that the field could be substituted with. The following two Java instructions

```java
p.f = 3;
return p.f;
```

are translated into the subsequent LIR. The first instruction assigns the value 3 to the field with the offset 8 in the object referenced by the `ESI` register. The second instruction reloads the field into the `EAX` register. The result is compared against the constant 3, which would be returned instead of the loaded field value if scalar replacement were used.

```
14  move [int:3|I] [Base:[esi|L] Disp: 8|]
16  move [Base:[esi|L] Disp: 8|] [eax|I]
18  assert [EQ] [eax|I] [int:3|I] "scalar replacement error"
24  return [eax|I]
```

The machine code that gets generated for the run-time assertion is shown below. If the `EAX` register equals 3, the postulated condition is fulfilled and error handling is omitted. Otherwise the address of the message to be displayed, the current program counter and the values of all registers are pushed onto the stack via `pushad`. For lack of an appropriate machine instruction, the program counter is pushed as the return address of a call to the next line. The actual output of the error message and the register values is delegated to a run-time routine. Finally, the `hlt` instruction stops the execution of the program.

```
00001000  cmp          eax, 3
00001003  je           0000101A
00001009  push         8276080h
0000100E  call         00001013
00001013  pushad
00001014  call         08010360
00001019  hlt
0000101A  ...
```

4.7 Example

In continuation of the example from the previous chapter, scalar replacement is applied to a method that computes the factorial and allocates objects for the loop counter and the intermediate results:

```
static Int factorial(Int x) {
  Int p = new Int(1);
  Int i = new Int(1);
  while (i.val <= x.val) {
    p = new Int(p.val * i.val);
    i = new Int(i.val + 1);
  }
  return p;
}
```

We already know from escape analysis that the loop counter object does not escape, so both its allocation and the initialization of its field are eliminated in the block `B0`.

```
B0 [0, 17] -> B1
__bci__use__tid____instr________________________________
. 0    2    a5     new instance Int
  4    2    i6     1
. 6    0    i13    a5._8 := i6
. 9    2    a15    new instance Int       (eliminated)
. 6    0    i23    a15._8 := i6           (eliminated)
. 17   0     25    goto B1
```

At this point, `a5._8` and `a15._8` are mapped to `i6`, i.e. the constant 1. Since the block `B1` is the target of a backward branch, phi functions are created for local variables as well as for fields. The compiler also creates a phi function for `p.val`, because escape analysis does not know that `p` escapes before it parses the return instruction. The field `a26._8` is mapped to `i28` and `a27._8` is mapped to `i29`.

```
B1 [18, 26] -> B3 B2
Locals:
 0   a4                     // x
 1   a26 [a5 a33]           // p
 2   a27 [a15 a45]          // i
Fields:
 0   a26._8 = i28 [i6 i36]  // p.val
 1   a27._8 = i29 [i6 i48]  // i.val
__bci__use__tid____instr________________________________
. 19   1    i30    a27._8                 (eliminated)
. 23   1    i31    a4._8
. 26   0     32    if i29 > i31 then B3 else B2
```

The loop body `B2` updates the loop counter and originally created a non-escaping object for the new value. Scalar replacement eliminates the allocation site `a45` and prevents `a27._8` from being loaded. The increased counter is computed as `i29 + i47` instead of `i46 + i47`.

```
B2 [29, 60] -> B1
__bci__use__tid____instr________________________________
. 29   2    a33    new instance Int
. 34   1    i34    a26._8
. 38   1    i35    a27._8                 (eliminated)
. 41   2    i36    i34 * i29
. 6    0    i43    a33._8 := i36
. 46   2    a45    new instance Int       (eliminated)
. 51   1    i46    a27._8                 (eliminated)
  54   1    i47    1
. 55   2    i48    i29 + i47
. 6    0    i55    a45._8 := i48          (eliminated)
. 60   0     57    goto B1
```

The final block `B3` neither allocates an object nor accesses the field of a non-escaping object. Therefore, scalar replacement does not change it.

```
B3 [63, 64]
__bci__use__tid____instr________________________________
. 64   0    a58    areturn a26
```

From a static point of view, scalar replacement eliminates two of four allocation sites. If the method is called to compute the factorial of n, $n + 1$ objects are allocated less than in the original version. As an additional optimization, the parameter object may be allocated on the caller stack. This is described in the next chapter.

5 Thread-Local Optimizations

Some objects escape the current method but not the current thread, typically because they are passed to a method that cannot be inlined. Although the fields of such objects must not be replaced by scalar variables, several other optimizations are possible:

- An object that is accessed only by one method and its callees can be allocated on the stack, which is cheaper than allocating it on the heap.
- Fields of a stack object can usually be accessed relative to the base of the current stack frame, so that the address of the object needs not be loaded.
- Synchronization on thread-local objects can be removed because they will never be accessed by another thread.
- The garbage collector does not have to free stack objects. They are deallocated at the end of the method when the stack frame is popped.

The first part of this chapter shows how objects are allocated on the stack, what costs are saved and where attention is demanded. The second part deals with object locking and synchronization removal.

5.1 Stack Allocation

In C#, the designer of a type decides where objects are allocated. Instances of *classes* are allocated on the heap, whereas instances of *structures* are allocated on the stack. The C++ language is even more flexible. Objects of one type can both be allocated on the stack and on the heap. The declaration of a local variable reserves memory on the stack, and the `new` keyword on the heap.

Java supports only one way of object creation. The `new` keyword must be used in order to allocate objects on the heap. There is no possibility for the programmer to influence this process. The just-in-time compiler, however, may generate machine code for an individual allocation site to create objects on the method stack if escape analysis guarantees that it is safe to do so.

5.1.1 Stack Layout

The *method stack* is a contiguous area in memory for the local data of currently executing methods. Every thread is given its own method stack. When a method is called, a new *frame* is pushed onto the method stack, and when the method returns to the caller, the stack frame is removed again. Every stack frame corresponds to one method invocation, i.e. recursive calls produce multiple frames.

Every processor provides some basic machine instructions for the creation, management and removal of stack frames, but the actual frame layout and its partition is left to the runtime environment. The Java HotSpot VM uses two different frame layouts for interpreted and compiled methods, which can co-exist on the same stack.

The stack frame for an interpreted method provides space for local variables and the operand stack, whereas a compiled method stores these values in registers and spill slots. Since interpreted methods do not allocate any objects on the stack, this chapter deals only with frames of compiled methods (see Figure 5.1).

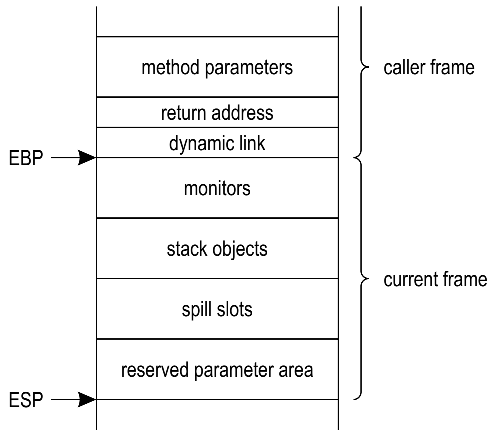


Figure 5.1: Stack frame of a compiled method

The base pointer `EBP` points to the start of the current stack frame, and the stack pointer `ESP` to its end. Upon a method call, the `EBP` register is pushed onto the stack and serves as a *dynamic link* to the caller's frame. It is reloaded when the method returns. Each stack frame is split into four areas:

- The monitor area is used for synchronization. Its size depends on the maximum number of objects that are simultaneously locked by the current method. Each entry occupies two words for the address and the header of a locked object.

- Stack objects are allocated immediately below the monitors. Since the total frame size must be known at compile time, variable-sized arrays do not qualify for stack allocation. Besides, if a non-escaping object is created within a loop, it is allocated on the stack only if the same stack slot can be reused in every iteration.
- If an operation requires more registers than are currently free, register allocation selects values that are temporarily swapped out to memory. This is called *spilling*. The values are stored in the spill slots of the current frame. Future operations can either use them directly from memory or reload them into a register.
- The area at the end of the frame is reserved for passing parameters to callees. It is large enough for any callee of the current method. The parameters are not pushed onto the stack, so the value of the `ESP` register remains unchanged. This makes it easier for the garbage collector to deal with parameters that are object references.

The method stack grows towards the beginning of memory. This means that stack slots at the top of a frame have higher addresses than those beneath. Slots of the current frame are usually addressed relative to `EBP` with a negative offset or relative to `ESP` with a positive offset. A *frame map* supports the compiler in the generation of machine code that refers to data on the stack. It stores the size of each area and converts between a stack slot index and an address.

5.1.2 Allocation of Stack Objects

For the allocation of a stack object, it is only necessary to initialize its fields. No memory needs to be reserved. The objects are allocated within the current frame which is automatically set up at the beginning of the method.

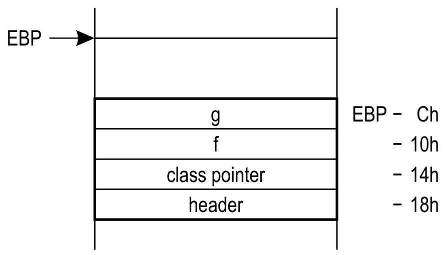


Figure 5.2: Layout of a sample stack object

During compilation, each stack allocation site is assigned a unique location in the frame. All objects allocated by the site are stored at this location. The location of a stack object is referred to as its *name* and specified as the offset from the base pointer. The name of the sample stack object in Figure 5.2 is -18h.

The machine code generated for the allocation site initializes the body of the object. It stores the value 1 into the header word, which marks the object as unlocked. A pointer to the class of the object is copied from the EDX register into the second word. The remaining words are cleared out so that the fields get their default values as described in the Java language specification:

```
00001000  mov           dword ptr [ebp-18h], 1
00001007  mov           dword ptr [ebp-14h], edx
0000100A  xor           eax, eax
0000100C  mov           dword ptr [ebp-10h], eax
0000100F  mov           dword ptr [ebp-Ch], eax
```

Within a loop, an object can only be allocated on the stack if the same space can be reused in every iteration. In other words, two objects created by the same site in different iterations must never be accessible at the same time. In the following example, no objects are allocated on the stack, because the object from the previous iteration can still be accessed via the variable q after the next one was allocated:

```
p = new T();
while (...) {
  q = p;
  p = new T();
  ...
}
```

The frame map accumulates the size of all stack objects. After register allocation, the sizes of the monitor area, stack objects, spill slots and parameter area are added up. The result is the total frame size that must be reserved on the stack upon method invocation.

5.1.3 Field Access

Apart from a cheaper allocation and deallocation, stack objects also facilitate a more efficient field access. The location of stack objects within the current stack frame is already known at compile time. Since the fields can be accessed relative to the EBP register, the address of the object needs not be loaded at run time.

When we allocate a stack object p as described in the previous section and the stack frame looks as in Figure 5.2, then the Java statement `p.f += 7` is translated into the following machine code:

```
00001000  mov          edi, dword ptr [ebp-10h]
00001003  add          edi, 7
00001006  mov          dword ptr [ebp-10h], edi
```

Elements of stack-allocated arrays are accessed in a similar way. The address includes an index register and a scale factor in addition to the base register and the offset. Assume that `arr` is an integer array of length 3 and that the index `i` is stored in the `ESI` register. The assignment of 7 to `arr[i]` yields the following machine code:

```
00001000  cmp          esi, 3
00001003  jae          00001034
00001009  mov          dword ptr [ebp+esi*4-14h], 7
...
00001034  // throw ArrayIndexOutOfBoundsException
```

Although arrays on the stack always have a fixed size, their length is stored in the header for the case that the array is passed to a callee. However, a range check within the allocating method can skip loading the array length and directly compare the index with a constant. The unsigned comparison also detects a negative index, which looks like a large unsigned positive number [51]. If the check fails, the machine code jumps to a stub that throws an exception. Sometimes the index is a constant as well. In this case, the range check is eliminated if the index is within bounds, or replaced by an unconditional jump to the stub otherwise. The move instruction performs the actual assignment. The index register is multiplied with 4, because each array element takes up 4 bytes.

As long as fields are addressed relative to the base pointer, no pointer to the stack object exists. Since heap objects referenced by the stack object must be considered during garbage collection, the stack object is registered in the oop map (see Section 6.1.4). This causes the garbage collector to treat pointers in the stack object as root pointers and to keep the referenced heap objects alive.

When a stack object flows into a phi function, its address must be loaded. This is done via a `lea` (*load effective address*) instruction. For

```
if (x > 0) {
  p = new T(...);
} else {
  p = new T(...);
}
p.f = 7;
```

the compiler generates the following code:

```
00001000  cmp          esi, 0
00001003  jle          00001028
00001009  // allocate stack object at ebp-10h
00001020  lea          edi, [ebp-10h]
00001023  jmp          00001042
00001028  // allocate stack object at ebp-20h
0000103F  lea          edi, [ebp-20h]
00001042  mov          dword ptr [edi+8h], 7
```

At compile time, each stack allocation is assigned a unique position within the stack frame. Depending on the value of `x` in the `ESI` register, an object is allocated at one of the two positions. At the end of each alternative block, the address of the stack object is loaded into the `EDI` register to resolve the phi function. The field `f` can then be accessed with a positive offset relative to this address. If the else-branch is executed for example, the stack frame looks as in Figure 5.3.

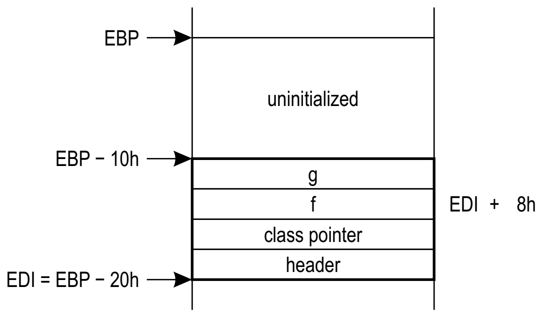


Figure 5.3: Access of stack object via a pointer

In such a situation, we are not able to precisely describe the object in the oop map, because we do not know at compile time which memory area actually contains the object. We cannot register all stack objects of the frame, because some of them may not be initialized and thus provide illegal root pointers. Instead, we rely on the garbage collector to trace the pointer for the phi function and to traverse the fields of the referenced stack object.

5.1.4 Write Barriers

Generational garbage collection is based on the observation that most objects die young and that objects which have been alive for a while are typically part of a global data structure and will be used for an even longer period of time [41]. Therefore, the memory is divided into distinct areas, called *generations*. New objects are allocated in the young generation. If an object has survived several collection cycles, it is moved into an older generation that is collected less frequently.

In order to identify the set of live objects for one particular generation, not only root pointers must be considered, but also pointers from objects contained in other generations. The garbage collector needs not worry about pointers from young to old objects, because all young generations are scanned before an old one is collected. It must, however, be possible to collect a young generation without inspecting every object in the older ones. Therefore, pointers from old to young objects are stored in the *remembered set* of the young generation.

When a pointer is installed into an object, it would be too expensive to immediately update the remembered set. Instead, the heap area that holds the object is marked as dirty. Such an area is commonly referred to as a *card*. Before the next collection, the garbage collector scans all dirty cards and updates the remembered set. The data structure that specifies which cards are dirty is called *card marking array* (see Figure 5.4).

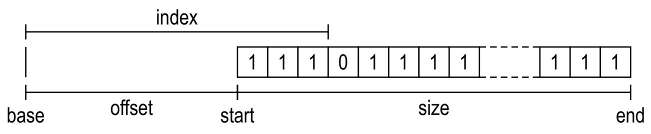


Figure 5.4: Card marking array

Every assignment to an object field is associated with a *write barrier* as shown in Listing 5.1. Assume that the `EAX` register points to the object that holds the field. Each card has a size of $2^9 = 512$ bytes, so that the index of the card containing the object can be computed via a right shift of the object address by 9 bits. Afterwards, the corresponding entry in the card marking array is set to 0.

```
00001000  shr          eax, 9
00001003  mov          byte ptr [eax+35E380h], 0
```

Listing 5.1: Assembler code for a traditional write barrier

It is more efficient to set the dirty byte from 1 to 0 instead of the other way round, because SPARC processors [91] provide a special register for the constant 0. Since the heap is not located at the beginning of memory and the first possible index is greater than 0, the card marking array is accessed relative to the base address 35E380h and not relative to its actual start address.

No write barriers are emitted for fields of stack objects allocated within the current method. On the one hand, they are not necessary, because pointers in stack objects are root pointers and must be inspected at every collection cycle anyway. On the other hand, they are not allowed, because the card marking array only covers the heap and any write barrier for a stack object would modify a byte outside the array or cause an access violation.

If a formal parameter does not escape a method or its callees, the method may be called both with a stack-allocated and a heap-allocated actual parameter. For such a formal parameter, a modified write barrier is emitted, which performs a bounds check before the card marking array is accessed (see Listing 5.2).

```
00001000  shr         eax, 9
00001003  sub         eax, 41A00h
00001009  cmp         eax, 50000h
0000100F  jae         0000101C
00001015  mov         byte ptr [eax+39FD80h], 0
0000101C  ...
```

Listing 5.2: Assembler code for a stack-aware write barrier

The write barrier computes the card index via a right shift and subtracts the offset (= start address – base address) of the card marking array (see Figure 5.4). The unsigned check of the result against the array size fails when the result is a negative number and when it is greater than the size. Both cases are detected by the `jae` (*jump above equal*) instruction. Since the offset has already been subtracted from the index, the array is accessed relative to the start address instead of the base address.

Although the heap may grow on demand, the size of the card marking array can directly be emitted into the machine code. At startup, the VM reserves address space for the maximum size of the heap and relies on facilities available in the various operating systems to avoid committing too many resources (such as physical memory or swap space) for reserved pages until they are actually used for the first time. Reserved but not yet committed pages still take up some amount of space in the page table, but the cost is fairly small. The card marking array uses the same strategy. The VM reserves memory large enough to correspond to the maximum size of the heap, but only commits regions of the array as necessary. For this reason, the base, start and end address, and the size of the array are constant.

The stack-aware write barrier takes five instead of two machine instructions, but it is required for non-escaping formal parameters only. No write barriers are emitted for stack objects within the allocating method, and traditional write barriers are emitted for the remaining objects on the heap. Since bytecode analysis (see Section 3.4.2) produces interprocedural escape information before a method is compiled, a stack object may be passed to an interpreted method as well. Thus, the interpreter must also execute the stack-aware write barrier.

5.2 Synchronization Removal

In a multi-threaded program, threads usually operate on common data to some degree. Sharing data without precaution can cause serious problems: If multiple threads modify a shared value simultaneously, they may unintentionally overwrite the changes of each other (*lost update*). Moreover, a thread may read values in an inconsistent state if another thread currently modifies them (*inconsistent read*).

The Java language provides convenient constructs for the synchronization of threads. They allow programmers to designate critical code regions, which act on a shared object and may be executed only by one thread at a time. The first thread that enters the region locks the shared object. If a second thread is about to enter the same region, it must wait until the first thread has unlocked the object again.

The synchronization constructs are pervasively used in the Java system libraries. Many data structures are synchronized to ensure correct results for the rare case that they are shared among multiple threads. Much research has been done to reduce the cost of synchronization in JVM implementations [9, 59], but complete elimination of useless synchronization is still a desirable goal.

We start with a short outline of the locking mechanism in the Java HotSpot VM. Then we show how escape analysis is used to remove unnecessary synchronization on thread-local objects. The final section describes how synchronized methods are inlined to expose additional optimization opportunities.

5.2.1 Object Locking

For every Java object, memory is allocated to store the current values of the object's fields. This memory is preceded by one word for the header and a pointer to the class of the object. The header word stores certain flags and is used to distinguish locked and unlocked objects [3].

Figure 5.5 shows how objects are internally represented by the Java HotSpot VM. As long as an object is unlocked, the last two bits of the header word have the value 01. The remaining bits store the identity hash code as well as the age of the object with respect to generational garbage collection.

When a thread synchronizes on an *unlocked* object, a pointer to the object and its header word are saved in the monitor area of the thread's current stack frame. The combination of the displaced header and the object pointer is called a *basic object lock*. A pointer to the basic object lock is stored in the object header.

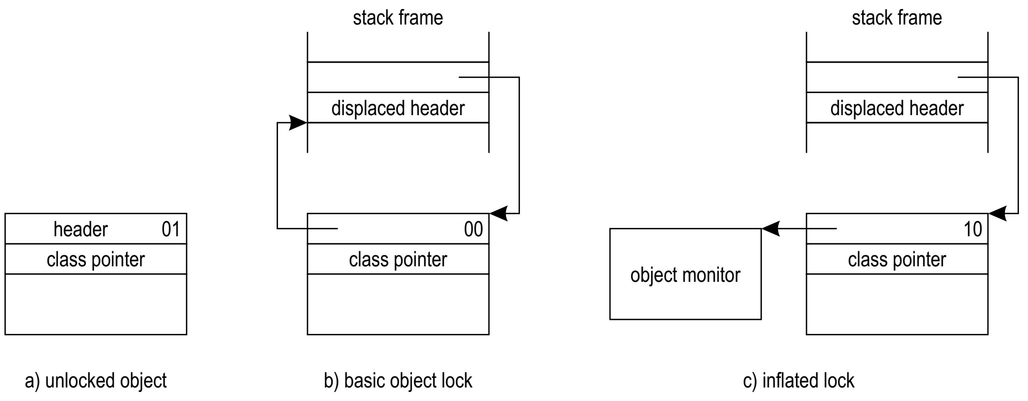


Figure 5.5: Representation of locked and unlocked objects

Since stack slots are always aligned at word boundaries, the last two bits of the header word are now 00. Note that the arrow in Figure 5.5b points to the bottom of the displaced header, because addresses increase towards the top of the frame.

If a thread tries to synchronize on a locked object, it checks if the object header points into its own stack. In this case, we speak of *recursive locking*. The thread already owns the object's lock and can safely execute the synchronized block. Nevertheless, a basic object lock is created whose displaced header is set to zero.

The first time another thread tries to obtain a lock on an already locked object, an *object monitor* is allocated for the management of waiting threads. The object header is modified to point to the monitor and the last two bits are set to 10 as shown in Figure 5.5c. Any further thread that synchronizes on the same object is added to the list of waiting threads. While setting up a basic object lock is directly embedded into the machine code by the JIT compiler, synchronization of multiple threads is delegated to a run-time routine.

The basic object lock on the stack supports the JVM in unlocking an object. If the displaced header is zero, the object was locked recursively by one thread and must not yet be unlocked. An object whose header points to the current basic object lock is unlocked by restoring the header word from the displaced header. In any other case, i.e. if the object was locked by different threads, a run-time routine is called to do the unlocking and resume a waiting thread.

5.2.2 Synchronization on Thread-Local Objects

As long as only one thread synchronizes on an object, the VM gets away with basic object locks. Although they are significantly cheaper than object monitors, they still involve some overhead. The JIT compiler can improve the performance of the generated machine code by completely removing synchronization on objects for which escape analysis guarantees at compile time that they are never accessed outside the allocating thread.

The following method allocates a new object and initializes its field `f` with the value of the parameter `x`. It increments the field in a synchronized block, which may result from inlining a synchronized method as described in Section 5.2.3:

```
static int foo(int x) {
  T obj = new T(x);
  synchronized (obj) {
    obj.f = obj.f + 1;
  }
  return obj.f;
}
```

Without escape analysis, the compiler generates the HIR shown below. The `enter monitor` instruction locks the object `a4` and `exit monitor` unlocks it again. If the object were shared among different threads, the synchronization would prevent that one thread computes the increment while another one reads the old value and finally overwrites the incremented value with its own result.

```
__bci__use__tid____instr________________________________
. 0    9    a4     new instance T
. 6    0    i10    a4._8 := i3  // inlined constructor
. 12   0     12    enter monitor(a4)
. 15   1    i13    a4._8
  18   1    i14    1
  19   1    i15    i13 + i14
. 20   0    i16    a4._8 := i15
. 24   0     17    exit monitor(a4)
. 34   1    i22    a4._8
. 37   0    i23    ireturn i22
```

Escape analysis yields that `a4` does not escape the allocating thread. Since the object will never be locked by more than one thread, synchronization can be removed. In this example, the complete allocation of the object is eliminated and its fields are replaced by scalars. The resulting HIR looks as follows:

```
__bci__use__tid____instr________________________________
  18   1    i14    1
  19   1    i15    i3 + i14
. 37   0    i23    ireturn i15
```

Without synchronization removal, the allocation could not be eliminated because locking requires the object to exist. If the object were passed to a method that cannot be inlined, scalar replacement would not be possible, but synchronization could still be removed in the allocating method as long as the callee does not let the parameter escape globally.

Before Java 5.0, the Java memory model restricted synchronization removal. Each thread had a working memory, in which it kept copies of the values of variables from the main memory that was shared between all threads. According to the old Java memory model [40], locking and unlocking actions caused a thread to flush its working memory and empty all variables, which guaranteed that the shared values were reloaded from main memory afterwards. Synchronization on a thread-local object had the effect of a memory barrier [77]. Therefore, the old Java memory model did not allow useless synchronization to be completely removed, but the new one for Java 5.0 does [63].

5.2.3 Inlining of Synchronized Methods

The `synchronized` statement provides mutual exclusion for a single block within a method. As a convenience, a method can be declared as synchronized as well. Such a method behaves as if its body were contained in a `synchronized` statement. Nevertheless, there are certain differences:

- A synchronized block must specify the object to be locked. Thus it can synchronize on arbitrary objects. A synchronized method always locks the receiver of the method call.
- Synchronization of a block is explicitly represented in the bytecodes by a pair of a `monitorenter` and a `monitorexit` instruction, whereas the synchronization of a method is only visible in its signature.

Escape analysis must not remove synchronization from a method, because the method may also be called on an escaping receiver. The only chance for the JIT compiler to avoid synchronization is to inline the method. The inlining of synchronized methods is explained below, considering `StringBuffer.append` as an example.

In Java, strings are concatenated with a string buffer. The class `StringBuffer` is synchronized for the rare case that an instance is shared among multiple threads. JDK 5.0 adds the class `StringBuilder` for use by a single thread. It supports the same operations but is faster as it performs no synchronization. Both classes are derived from `AbstractStringBuilder`, which provides unsynchronized versions of the common methods. `StringBuffer` overrides the methods and adds synchronization according to this pattern:

```
class StringBuffer extends AbstractStringBuilder {
  public synchronized StringBuffer append(String str) {
    super.append(str);
    return this;
  }
}
```

The class `StringBuffer` is retained mainly for compatibility reasons. Its methods are small and can normally be inlined by the JIT compiler. Consider this piece of Java code:

```
StringBuffer buf = new StringBuffer();
buf.append("some text");
```

Although `append` is a virtual method, it can be bound statically in this example because the compiler knows from the preceding allocation site that the type of the receiver is `StringBuffer`. When the compiler inlines the method, it inserts an `enter monitor` and an `exit monitor` instruction into the HIR to preserve synchronization. The negative BCI indicates that the two instructions have no equivalents in the bytecodes:

```
B0 [0, 15]
__bci__use__tid____instr________________________________
. 0    8    a1      new instance StringBuffer
// inlined constructor of StringBuffer
  9    2    a14     <string "some text">
. -1   0     18     enter monitor(a1)
. 2    0    a19     a1.invokespecial(a14)
                    AbstractStringBuilder.append
. -1   0     20     exit monitor(a1)
```

A locked object must reliably be unlocked again, even when an exception is thrown within the synchronized code. For synchronized methods, this is guaranteed by the JVM. For synchronized blocks, the `javac` compiler generates an exception handler that unlocks the object and afterwards rethrows the exception. If a synchronized method is inlined, such an exception handler does not exist and must be generated on-the-fly by the JIT compiler:

```
B1 [-1, -1]
__bci__use__tid____instr________________________________
. -1   1    a22     incoming exception
. -1   0     23     exit monitor(a1)
. -1   0     24     throw a22
```

The method `AbstractStringBuilder.append` is too large to be inlined, but interprocedural escape information states that the receiver does not escape. If `a1` does not escape otherwise, synchronization and the exception handler can finally be removed. The result looks as if the programmer had used the unsynchronized `StringBuilder` of the JDK 5.0:

```
B0 [0, 15]
__bci__use__tid____instr_______________________________
. 0    8    a1     new instance StringBuffer
// inlined constructor of StringBuffer
  9    2    a14    <string "some text">
. 2    0    a19    a1.invokespecial(a17)
                   AbstractStringBuilder.append
```

6 Run-Time Support

The Java HotSpot VM interprets a method several times before the method is compiled. This way, machine code is generated only for the most frequently called methods. In certain situations, the Java HotSpot VM is forced to stop executing a method's machine code and transfer control back to the interpreter. Any optimizations performed by the compiler must be undone. This is called *deoptimization* [49].

Deoptimization and garbage collection require the compiler to supply *debugging information*, which describes the current program state as seen by the bytecodes, respectively the interpreter. It specifies where the machine code stores the values of local variables, which variables contain pointers into the heap, and which objects are locked.

In the context of escape analysis, the debugging information must also consider optimized objects. Eliminated objects must be reallocated during deoptimization, and objects for which synchronization was removed must be relocked. During garbage collection, pointers in stack objects must be treated as root pointers.

Scalar replacement, stack allocation and synchronization removal thus do not only affect the compiler, but also require an adaptation of the debugging information, the garbage collector and the deoptimization framework. This chapter deals with the modifications that were applied to the Java HotSpot VM in order to allow a safe execution of optimized methods.

6.1 Debugging Information

The aim of debugging is to provide the programmer with a snapshot of the current program state at the source code level. Various optimizations in the compiler, however, complicate the mapping between the machine code and the original source code. Inlining, for example, combines multiple methods into one piece of machine code and makes it difficult to identify the currently executed source method. Besides, the values of local variables may be stored in registers.

For this reason, the compiler provides additional information about the optimized code. Although this data is historically called debugging information, it is also used for the identification of live objects during garbage collection and for deoptimization. Debugging information is not available for every single machine instruction, but only for some discrete points in the program at which method execution can safely be suspended. These points are called *safepoints*.

The compiler generates debugging information during register allocation, because before register allocation, the locations of local variables are not known yet, and afterwards they are no longer available. When the machine code is installed into the VM, the generated information is serialized and stored with the code in a compressed form.

6.1.1 Scope Entries

The interpreter stores values on the operand stack or in local variables, but when the machine code of a method is executed, the values are spread across registers and spill slots. To allow source-level debugging, the JVM must know the exact locations of all values.

In the debugging information, each value is represented by a *scope entry*, i.e. an instance of one of the classes shown in Figure 6.1. A `LocationEntry` describes which register or spill slot contains the value. If the value is known at compile time, a `ConstantIntEntry` or `ConstantOopEntry` directly stores the value. An object eliminated by scalar replacement is represented by an `ObjectEntry`. No entries are created for dead values.

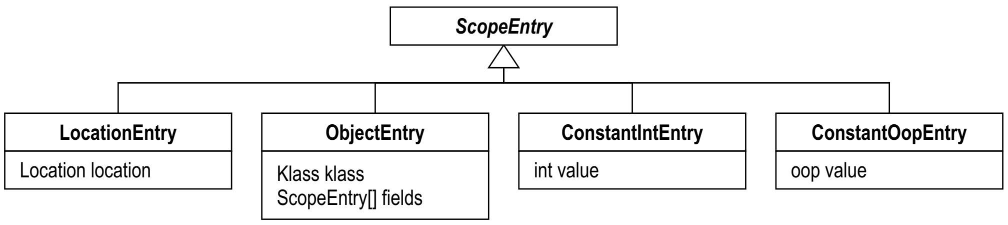


Figure 6.1: Class hierarchy for scope entries

The compiler generates two separate lists of scope entries: one for the operand stack and one for the local variables. Each scope entry describes one value, and its position within the list denotes which stack slot or local variable it refers to. Figure 6.2 presents a piece of source code, the generated machine code, and the scope entries for a particular safepoint.

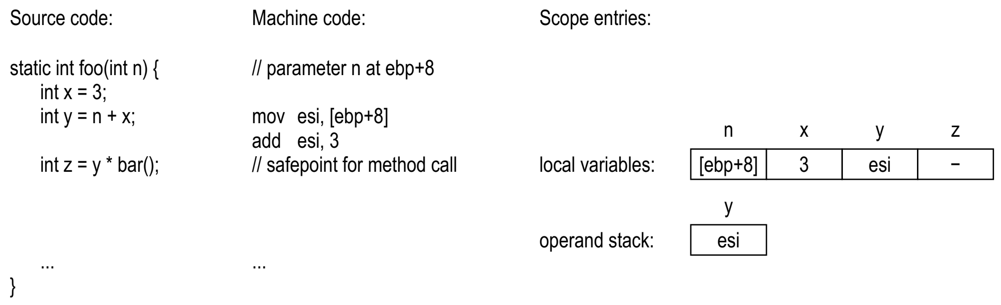


Figure 6.2: Example for scope entries

The lists of scope entries for the operand stack and the local variables are packed into a *scope descriptor*, which represents a single method invocation. Within an inlined method, the callee's values and the caller's values are described by different scope descriptors. The scope descriptors of inlined methods are linked in a stack-like manner, i.e. each descriptor stores a pointer to that of the surrounding method.

The complete debugging information provides a stack of scope descriptors for each safepoint in the machine code. When deoptimization is required, the descriptor stack for the current program counter is retrieved. Since the interpreter does not perform method inlining, each scope descriptor is used to reconstruct one interpreter frame. The slots of the new frame are initialized according to the scope entries.

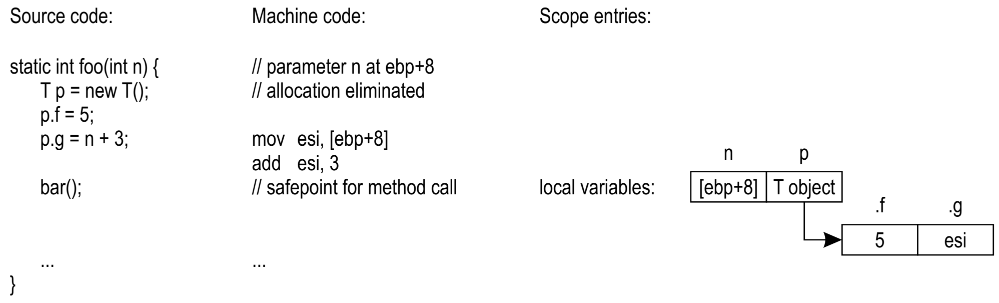
6.1.2 Object Entries

If a non-escaping object is replaced by scalar variables, its allocation is eliminated and its fields are directly stored in registers or spill slots. During deoptimization, the object must be reallocated for the interpreter and initialized with the current field values.

As we saw, the compiler creates scope entries for all local variables. If such a variable referenced an object that was eliminated by scalar replacement, the scope entry for this variable is an *object entry* describing the contents of the object instead of a reference to the object.

The object entry must specify the type of the eliminated object, because otherwise the deoptimization framework would not know which class to instantiate. The consequence is that an object represented by a phi function can only be eliminated if all operands have the same dynamic type.

In order to describe the values respectively locations of the object's fields, the object entry again holds a list of scope entries. The position of a scope entry within this list corresponds to the position of the described field within the object. An example is given in Figure 6.3.



Source code:

```
static int foo(int n) {
    T p = new T();
    p.f = 5;
    p.g = n + 3;

    bar();


    ...
}
```

Machine code:

```
// parameter n at ebp+8
// allocation eliminated

mov  esi, [ebp+8]
add  esi, 3
// safepoint for method call


...
```

Figure 6.3: Example for an object entry

Scalar replacement is not only applied to non-escaping objects, but also to non-escaping fixed-sized arrays that are exclusively accessed with constant indices (see Section 4.3). The object entry for an eliminated array is created analogous to that for an eliminated object. Its list of scope entries describes the locations of the array elements.

Serialization of debugging information and deoptimization require that the same non-escaping object in different local variables is represented by the same object entry. For this reason, each object entry is stored in a cache after creation. If the compiler is about to describe a local variable that contains a non-escaping object, the cache is searched for an object entry that represents the same object and can be reused.

6.1.3 Monitor Entries

The interpreter locks objects in a different way than the machine code, so the deoptimization framework must know which objects are currently locked. For this reason, the debugging information provides a list of *monitor entries*. A monitor entry encapsulates a scope entry that describes the location of a pointer for the locked object and the location of the displaced header.

Since objects are unlocked in reverse locking order, the compiler maintains a stack of locked objects when it translates the bytecodes into the HIR. An object is pushed onto the stack when it is locked and popped from the stack again when it is unlocked. During the generation of debugging information, a snapshot of the current monitor stack is converted to a list of monitor entries.

When an object is locked, a pointer to the object is saved on the method stack (see Section 5.2.1). Thus, the stack frame contains pointers to all currently locked objects. This simplifies the creation of a monitor entry, because the appropriate stack slot can be described, even if a pointer to the locked object is no longer present in the local variables.

If synchronization on a thread-local object was eliminated, it must be relocked during deoptimization, because the interpreter will later try to unlock it again. The method stack, however, does not contain the object's address in this case. Register allocation has to ensure that a pointer to the object is preserved in a register or in memory until the point where the object would have been unlocked. Otherwise, the object could not be described in the debugging information.

The corresponding monitor entry specifies the location of the pointer to the thread-local object. In addition, a flag is set to let the deoptimization framework know that the object must be relocked. In case of a method-local object whose allocation has been eliminated, the monitor entry encapsulates an object entry instead of a location entry.

6.1.4 Oop Map

Run-time methods of the Java HotSpot VM refer to objects in the Java heap via *handles*. A handle is a data structure that encapsulates a pointer to a Java object and resides in a special memory area where the garbage collector visits the object and updates the pointer if the object is moved. Machine code generated for a Java method, however, refers to objects via direct pointers for efficiency reasons. To contrast direct pointers with handles, they are called *ordinary object pointers* (oop).

In order to mark live objects, the garbage collector needs to know where the machine code stores pointers to heap objects. Therefore, the debugging information includes an *oop map* for each safepoint. It specifies all registers and stack slots that contain a root pointer.

As long as fields of a stack object can be accessed relative to the base of the stack frame, the address of the object is not loaded. Therefore, pointers in the stack object are root pointers and must be considered in the oop map. However, the oop map contains only one entry for the complete stack object instead of one per field that holds a pointer. This enables the garbage collector to mark the object after visiting it, so that the pointer fields are processed only once (see Section 6.3.2).

6.1.5 Serialization

After compilation, the generated machine code and meta information must be registered in the virtual machine. For this purpose, the compiler creates an *nmethod* (native method) data structure consisting of

- a header that specifies entry points into the machine code and various flags,
- relocation information for addresses of classes and methods in the machine code that must be updated if the referenced objects are moved in memory,
- machine code for the method,
- debugging information and method dependencies,
- an exception handler table, whose elements map a range of the machine code and an exception type to the entry of an exception handler, and
- an implicit exception table that specifies where execution continues when a machine instruction dereferences null or divides by zero.

Debugging information is split into three parts. The first part encodes all oop maps for the method in a compressed form. Each oop map is represented by a list of integers specifying the locations that contain pointers. Lower numbers correspond to CPU registers and higher ones to stack slots.

The second part stores scope entries and monitor entries for each safepoint. Due to inlining, it may be necessary to describe values of different methods. Each inlined method represents a separate scope. Figure 6.4 shows the data layout for one safepoint.

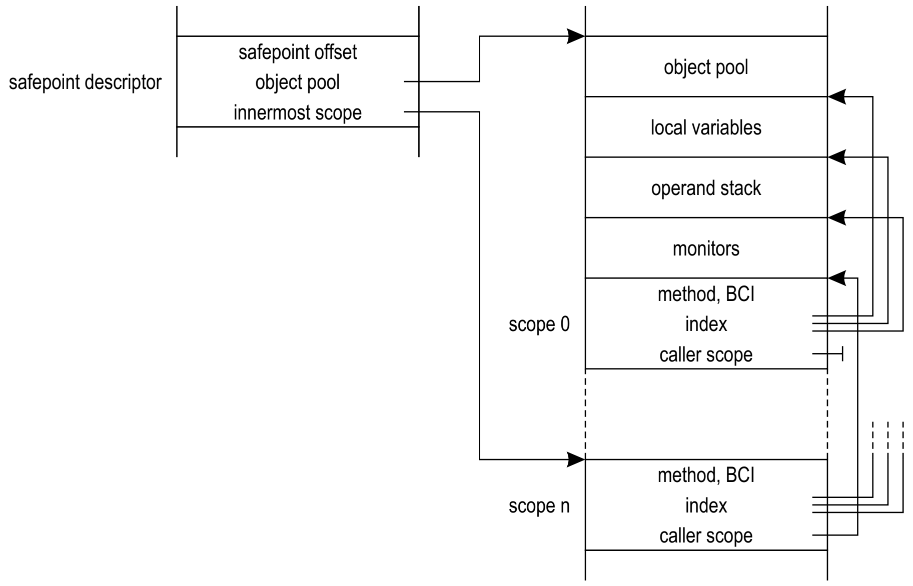


Figure 6.4: Internal representation of debugging information

The *object pool* stores object entries for eliminated allocations. Even if there are multiple scopes due to inlining, the object pool is written only once at the beginning. Each entry specifies a unique identification number, the type of the object and a list of scope entries for its fields. Since eliminated objects may reference each other, the transitive closure must be serialized.

Debugging information for a single scope consists of scope entries for local variables, scope entries for the operand stack, and monitor entries. If an entry refers to an eliminated object, the identification number of the appropriate object entry in the object pool is emitted.

Scopes are serialized sequentially from the outermost to the innermost inlined method. Each scope points to its description of local variables, the operand stack and locked objects, as well as to the next outer scope. It stores a reference to the corresponding method and the BCI of the current instruction in this method. Except for the innermost scope, all bytecode indices refer to call sites of inlined methods.

The third part of the debugging information is an array of safepoint descriptors with one element per safepoint. Each descriptor specifies the offset of the safepoint in the machine code and points to the object pool and to the innermost scope. Upon deoptimization, the VM looks up the safepoint descriptor whose offset matches the current program counter.

6.2 Deoptimization

As discussed at the beginning of this chapter, debugging information is accessed in various places throughout the virtual machine. This section deals with its use for deoptimization. We provide an example to motivate deoptimization, describe the basic framework and explain how it was extended for escape analysis.

6.2.1 Motivation

Although inlining is an important optimization and promotes scalar replacement of fields (see Section 4.4.3), it has traditionally been very difficult to perform for dynamic object-oriented languages like Java. It is not sufficient to examine call sites and inline the methods they invoke, because Java programs can dynamically load new code into a running program [84].

A method can only be inlined if the compiler identifies the called method statically despite polymorphism and dynamic method binding. Apart from static and final callees, this is possible if class hierarchy analysis [27] finds out that only one suitable method exists. If, however, a class is loaded later that provides another suitable method, maybe the wrong implementation was inlined.

If the compiler inlines virtual calls only when the receiver is *preexisting*, i.e. allocated before execution of the caller begins, class loading will not invalidate running code [28]. For general inlining, however, the Java HotSpot VM must be able to dynamically deoptimize a previously optimized method, even while executing machine code for it.

Assume that class `B` was not loaded yet when machine code for the method `calc` in our example below is generated. The compiler optimistically assumes that there is only one implementation of the method `foo` and inlines `A.foo` into `calc`. If the method `create` loads class `B` and returns an instance of it, the inlining decision turns out to be wrong and the machine code for `calc` is invalidated.

```
class A {
  void foo(Point p) { ... }
}

class B extends A {
  void foo(Point p) { ... }
}

static int calc(int x, int y) {
  Point p = new Point(x, y);
  A q = create();
  q.foo(p);
  return p.x * p.y;
}
```

Even if the method `foo` cannot be inlined, for example because of its size, it is statically bound to save the dispatching overhead. When the class `B` is loaded in the `create` method, the machine code of `calc` is invalidated because it jumps to the wrong `foo` method.

6.2.2 Method Dependencies

When a subclass of `A` is loaded which overrides `foo`, the VM must deoptimize the method `calc`. In other words, a dependency is introduced between `calc` and `A.foo`. It is recorded during the compilation of `calc` and stored both in the nmethod for `calc` and the class descriptor for `A`.

Assume that `A.foo` is not inlined, but statically bound based on class hierarchy analysis. The compiler uses interprocedural escape information to find out that the parameter `p` does not escape and can be allocated on the stack frame of `calc`. When class `B` is loaded, `calc` must be deoptimized not only because `A.foo` was statically bound, but also because the `Point` object may escape in `B.foo`.

However, it is not sufficient to record a dependency between `calc` and `foo`. Assume that the parameter `p` does not escape `A.foo` just because the compiler inlines the virtual method `bar` in `foo`. When another class is loaded that overrides `bar`, the method `foo` is deoptimized due to its dependency on `bar`. The machine code for `calc` must be invalidated as well, because `p` may escape in the newly loaded `bar` method.

If `calc` depended only on `foo`, it would not be deoptimized because method dependencies are not processed transitively. In other words, the compiler must record a dependency between two methods m and m' if

- m inlines m' or calls it with static binding, as well as if
- a direct or indirect callee of m inlines m' or calls it with static binding, and m passes at least one stack-allocated object to this callee.

Every native method stores a list of its dependencies. So when the compiler parses the method call of `foo` in `calc` and uses interprocedural escape information of `foo` to allocate some parameters on the stack, it not only records a dependency between `calc` and `foo`, but also inherits all dependencies between `foo` and its callees.

6.2.3 Basic Deoptimization Process

Deoptimization may be necessary when a new class is added to the class hierarchy. The VM examines the descriptors of the new class and its superclasses and marks dependent methods for deoptimization. It also iterates over the interfaces implemented by the new class and looks for methods depending on the fact that an interface had only one implementor.

Then the VM traverses the stacks of all threads. A frame that belongs to a marked method is not immediately deoptimized. Instead, the machine instruction at the program counter for this frame is patched to invoke a run-time stub. The actual deoptimization takes place when the frame is reactivated after all callees have returned. This is called *lazy deoptimization*. The nmethod is marked as non-entrant, so that the VM interprets new invocations instead of executing the patched machine code.

Upon deoptimization, all live registers are saved in a *register map*. Then the deoptimization framework creates an array of *virtual stack frames*, one for the method to be deoptimized and one for each inlined callee. A virtual frame does not exist on the stack, but stores the local variables, operand stack and monitors of a particular method. Debugging information is used to fill the virtual frame with the correct values from the register map and the memory.

In the next phase of deoptimization, the method stack is adjusted as shown in Figure 6.5. The frames of the run-time stub and the method to be deoptimized are removed and the virtual frames are unpacked onto the stack. Finally, a frame for the continuation of the run-time stub is pushed back onto the stack.

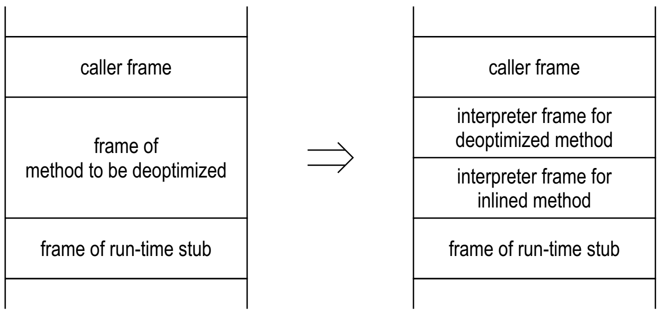


Figure 6.5: Adjustment of the stack during deoptimization

During unpacking of a virtual frame, the BCI is retrieved from the corresponding scope descriptor in the debugging information. It identifies the bytecode in the method that has to be executed next. The address of the interpreter code that handles the bytecode is pushed onto the stack as the return address. When the run-time stub returns, execution automatically continues in the interpreter.

6.2.4 Reallocation and Relocking

Due to escape analysis, deoptimization has to deal with objects eliminated by scalar replacement and objects for which synchronization was removed. The appropriate extension of the debugging information was discussed in Section 6.1. Based on this information, objects are reallocated and relocked before the virtual frames are created.

The reallocation code iterates over all object entries in the object pool. For each object entry, a new instance is allocated on the heap according to the specified type. A handle is created and stored in the object entry. If garbage collection is needed in the middle of reallocation, pointers in handles are treated as root pointers. Newly allocated objects are kept alive and pointers are updated if the objects move due to garbage collection.

Then the fields are initialized. A reallocated object may be referenced by multiple fields of other reallocated objects. Deserialization of debugging information takes care that fields referring to the same eliminated object are mapped to identical object entries. The pointer is retrieved from the handle in the object entry and stored in the heap objects.

Finally, all objects for which synchronization was eliminated are relocked. The representations of locked objects differ between interpreted and compiled code. We lock the objects as though the compiled code performed the locking and rely on the existing deoptimization code to convert the locks into the interpreter's representation. Debugging information also considers the level of synchronization, which allows the deoptimization framework to restore recursive locking.

When the virtual frames are filled with values, pointers are copied from object entries into local variables and the operand stack. Afterwards, the handles for reallocated objects that were stored in the object entries are released. Henceforth the garbage collector must not run until deoptimization has finished, because it does not examine the virtual frames and would free the newly allocated objects again.

6.2.5 Deoptimization of Stack Objects

Escape analysis uses interprocedural escape information to decide if actual parameters can be allocated in the stack frame of the caller. If the callee is not a static or final method but was identified as the only possible callee via class hierarchy analysis, a method dependency between the caller and the callee is recorded. The stack objects must be moved to the heap if

- the allocating method is deoptimized, because the interpreter frame created by deoptimization cannot contain stack objects, or if
- a new version of a direct or indirect callee is dynamically loaded, because the new version may let the objects escape.

A caller m that relies on interprocedural escape information of a callee m' inherits all dependencies of m'. Therefore, m is marked for deoptimization even if dynamic class loading overrides a callee of m'. Eliminated objects are reallocated when the frame of m is deoptimized after all callees have returned, but this is too late for stack objects.

Assume that a stack-allocated parameter escapes in the newly loaded callee via an assignment to a static field. Before the allocating frame is deoptimized, the garbage collector may run. It does not expect a static field to reference a stack object, so the object must immediately be moved to the heap. Besides, it would be difficult to retroactively adjust the static field to point to the heap object later.

Stack objects are moved to the heap as soon as all methods have been marked for deoptimization. When a heap object is allocated, a forwarding pointer to it is installed into the corresponding stack object. Objects for which synchronization was eliminated are relocked. Afterwards, the VM iterates over all stack frames and replaces pointers to deoptimized objects with their forwarding pointers (see Figure 6.6). This also affects interpreter frames because bytecode analysis allows stack objects to be passed to methods that have not been compiled yet.

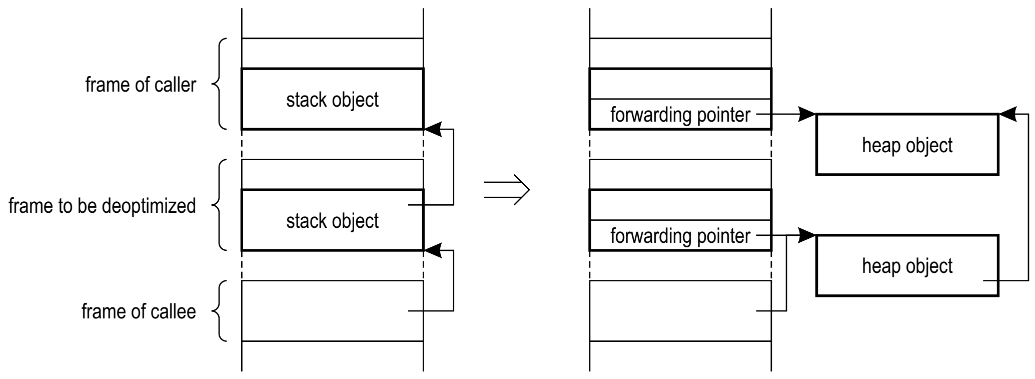


Figure 6.6: Deoptimization of stack objects

When the run-time stub deoptimizes a frame, all its stack objects have already been deoptimized and specify a forwarding pointer. Debugging information is used to fill the virtual frame with values. If a scope entry refers to a stack object, the object's forwarding pointer is stored in the virtual frame. The unpacked frame contains only heap pointers.

6.3 Garbage Collection

Although stack objects are implicitly deallocated when the stack frame is removed at the end of a method, the garbage collector cannot ignore them. Their fields may reference heap objects that need to be visited and marked as alive. However, stack objects must not move in memory, whereas heap objects may move due to heap compaction.

The Java HotSpot VM implements different garbage collection strategies that can be selected on the command line. By default, a generational, non-parallel garbage collector is used [54]. The heap is split into a young, an old, and a permanent generation.

The young generation consists of three spaces: *eden*, *from-space* and *to-space*. New objects are allocated in the eden. If it fills up, the young generation is collected by a *stop-and-copy* algorithm. The garbage collector first iterates over all root pointers and evacuates referenced objects of the eden and the from-space into the to-space. Then it scans the objects in the to-space and copies referenced, but not yet visited objects. Finally, the roles of the from-space and the to-space are swapped.

After a certain number of collection cycles, long-lived objects are moved to the old generation. The permanent generation stores internal data structures of the VM. These two generations are collected by a *mark-and-compact* algorithm, which requires four phases:

1. Live objects are marked by recursively traversing all reachable objects starting from the set of root pointers.
2. For each marked object, the garbage collector computes a new address. It walks linearly through the space and annotates live objects with a forwarding pointer to their new location.
3. All references are updated to point to the new addresses specified by the forwarding pointers.
4. The final phase walks the space linearly, moves objects to their new locations and clears the mark bits.

The garbage collector implements an operation to be performed for all objects as a so-called *oop closure* following the *Visitor* design pattern [36]. Starting with the root pointers, an oop closure is recursively applied to all reachable oops. Oop closures are used for the stop-and-copy collection, as well as in phases 1 and 3 of the mark-and-compact algorithm.

6.3.1 Wrapper Closure

Due to escape analysis, root pointers may reference stack objects. Stack objects need to be traversed but must not be copied to a new memory location. Therefore, the iteration of root pointers was modified to use a wrapper which abstracts from stack objects and presents their object fields as root pointers to the underlying closure (see Listing 6.1).

```
void do_oop(oop obj) {
  if (is_in_heap(obj)) {
    wrapped_closure.do_oop(obj);
  } else if (!obj.has_been_scanned()) {
    obj.set_has_been_scanned();
    obj.iterate_oop_fields(this);
  }
}
```

Listing 6.1: Wrapper closure that deals with stack objects

The heap is a continuous area in memory, so the `is_in_heap` test can be implemented as a comparison of the object address against the start and the end of the heap. If the test succeeds, the wrapped closure is invoked on the oop. Otherwise, the wrapper is applied to the oop fields of the stack object, which may again point to stack objects.

A stack object can directly or indirectly reference itself, so the garbage collector must remember which stack objects have already been scanned. We could set the mark bit to identify scanned objects, but this would require an extra pass to reset them.

Instead, we use two bits in the header word of stack objects to encode three values. The bits are initialized with 0, which means that the object was not yet scanned. At the beginning of every iteration over the root pointers, we toggle a global value between 1 and 2. The `has_been_scanned` test compares the header field with the global value for equality, and `set_has_been_scanned` copies the global value into the header of the stack object.

6.3.2 Root Pointers for Stack Objects

The garbage collector also influences how stack objects are represented in the debugging information. When a variable refers to a stack object directly and not through a phi function, the fields can be accessed relative to the base of the stack frame (see Section 5.1.3). It is not necessary that a pointer to the object exists.

In order to prevent heap objects referenced by stack objects from being deallocated, we could register the reference fields of stack objects as root pointers in the oop map. However, this would cause problems when the location of such a field is not statically known. Consider the following example:

```
p = new T();    // object 1
if (...) {
  q = p;
} else {
  q = new T();  // object 2
}
```

Since the variable q may point to one of two different stack locations, a phi function is created. When the condition of the if-branch is true, the address of object 1 is loaded into a register to resolve the phi function. The garbage collector regards this register as a root pointer and consequently visits all reference fields of object 1.

The variable p unambiguously refers to object 1 all the time. If its fields were registered in the oop map, the garbage collector would visit them a second time without checking whether the object was already marked as visited. This is not allowed. The mark-and-compact algorithm updates all pointers before the objects are actually moved to their new locations. Thus, after the first visit, a field does not point to an object any more. When the stop-and-copy algorithm visits a field twice, it duplicates the referenced object in the to-space.

Therefore, we create a special entry in the oop map to describe the stack object as a whole instead of its individual fields. When the garbage collector reads the oop map and encounters such an entry, it creates a temporary root pointer and marks the stack object as scanned before it visits the reference fields. Even if there is another root pointer to the same object, the fields will not be processed a second time.

7 Evaluation

This chapter evaluates our optimizations using a variety of benchmarks and applications. It considers quality and performance of the generated machine code as well as the impact of escape analysis on the compilation speed. Besides, we try to explain why a speedup is achieved and how certain design decisions affect the gain of escape analysis.

All tests were executed with Sun Microsystems' Java HotSpot client VM of the JDK 5.0 with our research compiler. Consequently, also the run-time libraries of the JDK 5.0 were used. The test machine was powered by an Intel Pentium 4 processor 540 with 3.2 GHz and 1 GB of main memory. The operating system was Microsoft Windows XP Professional (SP2).

7.1 CompileTheWorld

CompileTheWorld (CTW) is a compiler stress test directly integrated into the Java HotSpot VM. It can be activated via a VM flag and sequentially compiles all classes from the run-time libraries. No machine code is executed. The test is used to verify that the compiler does not crash and to measure compilation speed.

In the JDK 5.0, the workload for the compiler consists of 106,904 methods in 13,646 classes. Without escape analysis, 9.53 MB of bytecodes are compiled in 66.3 seconds, which results in an average compilation speed of 150,701 bytes/s. The generated machine code has a size of 52.55 MB.

Escape analysis identifies 1,331 of 114,584 object allocation sites as method-local. Table 7.1 shows which types of objects are eliminated most frequently. It reveals that instances of wrapper classes and some AWT classes can often be replaced by scalar variables. `Vector$1` is the anonymous class of enumeration objects returned by `Vector.elements`. Since the generated machine code is never executed, no information is available on how many object allocations are saved at run time.

class	sites
java.awt.Dimension	147
java.lang.StringBuilder	146
java.util.Vector$1	121
java.util.AbstractList$Itr	107
com.sun.org.apache.bcel.internal.generic.PUSH	94
java.awt.Rectangle	71
java.awt.Point	67
java.lang.Integer	61
java.lang.Long	50
java.lang.StringBuffer	49
java.lang.Double	46
java.util.HashSet	31
sun.security.jca.GetInstance$Instance	29
java.lang.Boolean	24
java.lang.Float	23
other classes	< 20
total number of eliminated allocation sites	1,331

Table 7.1: Number of eliminated allocation sites per class

11,249 sites allocate objects on the stack. 9,058 of them create a `StringBuilder` object and 707 a `StringBuffer` object. Moreover, 4,339 synchronization sites can be removed, and 3,718 of them synchronize an instance of `StringBuffer`.

The numbers mentioned so far do not include array allocation sites. 120 of 56,825 array allocation sites are eliminated and 430 allocate an array on the stack. In both cases, the arrays must have a constant length, which is 4.56 on average. Table 7.2 summarizes the results of escape analysis.

	sites	
object allocations eliminated	1,331	(1.16%)
object allocations on the stack	11,249	(9.82%)
object allocations on the heap	102,004	(89.02%)
array allocations eliminated	120	(0.21%)
array allocations on the stack	430	(0.76%)
array allocations on the heap	56,275	(99.03%)
synchronizations removed	4,339	(18.13%)
synchronizations retained	16,755	(70.03%)
synchronized methods	2,832	(11.84%)

Table 7.2: Static escape analysis results for CTW

In addition to 420,383 inlined methods, the compiler inlines another 19,947 methods whose size exceeds the maximum inline size. It does so, because it expects to increase the number of method-local allocation sites. 14,906 of the 440,330 inlined methods were synchronized.

The more aggressive inlining increases the workload from 9.53 MB to 10.78 MB of bytecodes. They are compiled to 56.48 MB of machine code in 86.3 seconds. In comparison to CTW without escape analysis, the compilation speed drops by 13.06% to 131,025 bytes/s.

Normally, callees of a method are likely to be compiled before the method because they are executed more frequently. This is not true for CTW. Methods are compiled in an arbitrary order, so precise interprocedural information for callees is often not available yet. The following benchmarks execute programs under realistic conditions and thus provide more meaningful statistics.

7.2 SPECjvm98

In assessing the performance of Java runtime environments, the SPECjvm98 benchmark suite [82] has become quite popular. It does not only provide information on the speed and quality of the just-in-time compiler, but rather measures the overall performance of a JVM including input and output, class loading and garbage collection. It is commonly used to compare virtual machines in different versions or by different vendors, as well as to evaluate the impact of new optimizations.

SPECjvm98 consists of seven benchmark programs with real-life relevance, which cover a broad range of scenarios and vary in their characteristics and system requirements. Each of the seven programs is executed several times and measured for the slowest and the fastest run:

- The slowest run is typically the first run of a program, because classes must be loaded and methods are either still interpreted or are just getting compiled. All in all, the slowest run is an indication of the *startup speed* of a virtual machine.
- The fastest run is usually the last run of a program, when most methods have already been compiled and the speed of the program does not significantly improve anymore. Most of the run time is spent in machine code. The fastest run thus indicates the *quality of the machine code* generated by the JIT compiler.

Execution of a program is repeated until no major change in run time is encountered. For the slowest and the fastest run, a score is computed as a ratio of the measured time and a reference time. The higher the score, the better is the performance of the JVM under examination.

SPEC defines very strict rules for how to run the benchmark in order to obtain official scores. The rules were marginally violated by the configuration used in this chapter, e.g. because the benchmark was not run from a web server but as a stand-alone application. Although the absolute numbers must not be compared with other SPECjvm98 metrics, the relative numbers of this chapter provide a meaningful evaluation of escape analysis and related optimizations.

The following sections consider some programs individually and examine how they benefit from our optimizations. The closing section summarizes the results and presents the overall performance of SPECjvm98 with and without escape analysis.

7.2.1 mtrt

The `mtrt` benchmark is the only multi-threaded program in the SPECjvm98 suite. It implements a ray tracing algorithm for two threads that render a 3D scene depicting a dinosaur on a 2D canvas by tracing light rays. Each thread is responsible for a certain area of the canvas.

The ray tracing algorithm performs a lot of floating-point computations and allocates short-lived objects for intermediate points and vectors. Therefore, the benchmark significantly benefits from good inlining decisions and scalar replacement of fields. The method `OctNode.Intersect`, for example, has roughly the following structure:

```
OctNode Intersect(Ray ray, Point intersect, ...) {
  Vector delta = new Vector(0.0f, 0.0f, 0.0f);
  int[] facehits = {-1, -1, -1};
  // set coordinates of vector delta
  // and update elements of facehits
  ...
  intersect.Add(delta);
  ...
}
```

It allocates a three-dimensional vector `delta` and sets the coordinates via the accessor methods `SetX`, `SetY` and `SetZ`. Finally, the vector is added to the point `intersect` that is passed as a parameter. The constructor, the accessor methods and the method `Add` are inlined, so that the `delta` object does not escape the method above. Its allocation is eliminated and its fields are replaced by local floating-point variables. The allocation of `facehits` can be eliminated as well, because the array has a fixed length of 3 and is accessed with constant indices only.

`OctNode.Intersect` is called 1,020,854 times in a single run of the benchmark, but the first 1,500 invocations are executed in the interpreter. Scalar replacement eliminates 1,019,354 objects and the same number of arrays, which saves a total of 46.66 MB on the heap. The two methods `PolyTypeObj.Intersect` and `TriangleObj.Check` also contain an eliminable vector allocation each, which saves 2,736,890 objects or 62.64 MB from being allocated on the heap.

`Scene.ReadPoly` offers another opportunity for scalar replacement. It contains the following lines of Java code:

```
String temp = infile.readLine();
... = Float.valueOf(temp).floatValue();
```

The method `Float.valueOf` parses the specified string and converts it into a floating-point number. It returns a `Float` object that is used only to retrieve the actual value. After inlining `valueOf` and `floatValue`, the allocation of the `Float` object is eliminated. The method is called only 6 times but contains a long-running loop. It is OSR compiled during the first invocation and fully compiled at the next one. Scalar replacement eliminates a total of 12,740 object allocations.

`Scene.Shade` creates an instance of `IntersectPt` and invokes `FindNearestIsect` on it. `FindNearestIsect` does not let the object escape, but it has a size of 330 bytes and is thus too large to be inlined. The object cannot be replaced by scalars, but it remains stack-allocatable. `Scene.Shade` is called 62,417 times, of which 1,500 invocations are interpreted, so 60,917 objects are allocated on the stack.

Table 7.3 shows the complete statistical data for an individual run of the `mtrt` benchmark. Static numbers refer to sites in the machine code, whereas dynamic numbers indicate how often these sites are executed.

	static numbers	dynamic numbers
object allocations eliminated	13 (6.47%)	3,784,525 (72.09%)
object allocations on the stack	8 (3.98%)	150,942 (2.87%)
object allocations on the heap	180 (89.55%)	1,314,370 (25.04%)
array allocations eliminated	1 (10.00%)	1,019,354 (77.62%)
array allocations on the stack	2 (20.00%)	2,417 (0.18%)
array allocations on the heap	7 (70.00%)	291,569 (22.20%)

Table 7.3: Statistics for one run of the mtrt benchmark

In order to prevent objects from escaping, the compiler decides to inline 53 methods whose size exceeds the usual threshold. 13 allocation sites of `Vector`, `Point` and `Float` objects and 1 array allocation can be eliminated. This saves 4.8 million allocations or 110 MB of memory at run time. 150,942 objects and 2,417 arrays with a total size of 3.97 MB are allocated on the stack.

When the benchmark is compiled without escape analysis, the fastest run requires 1.094 seconds. With escape analysis and scalar replacement enabled, this time is reduced by 32.1% to 0.828 seconds. When the server compiler is used, the fastest run takes 1.062 seconds.

7.2.2 db

The db benchmark performs multiple operations on a memory-resident database. Each record consists of a name, an address and a phone number. The benchmark reads a set of records from a file into memory and then adds, deletes, searches and sorts addresses. The results are printed on the console.

A database record is represented by an instance of the class Entry, which stores a list of strings. The method Entry.equals checks if the receiver and the entry parameter are equal. For this purpose, it performs a pairwise comparison of the corresponding string items:

```
Enumeration e1 = items.elements();
Enumeration e2 = entry.items.elements();
while (e1.hasMoreElements()) {
  o1 = e1.nextElement();
  o2 = e2.nextElement();
  // compare o1 and o2
}
```

An analysis of the bytecodes (see Section 4.4.4) reveals that the local variables e1 and e2 are not modified within the loop. Therefore, no phi functions hiding the exact type of the objects are created at the loop header. All methods invoked on e1 and e2 can be bound statically.

The method nextElement is called frequently and thus gets compiled before Entry.equals. Its interprocedural escape information states that the receiver does not escape. Therefore, nextElement is inlined although its size of 63 bytes exceeds the usual threshold.

Both elements and hasMoreElements are small enough to be inlined by default. The enumeration objects do not escape the method and can be replaced by scalars. Since Entry.equals is called 1,451,782 times, escape analysis eliminates the allocation of nearly 3 million objects with a total size of 44.29 MB.

7.2.3 jack

Jack is a Java parser generator based on the Purdue Compiler Construction Tool Set (PCCTS). It takes a set of combined grammatical and lexing rules in form of a text file and produces a Java class that parses an input file according to the grammar. The product name Jack was later changed to JavaCC [65].

The workload for the `jack` benchmark consists of a file that contains instructions for the generation of Jack. This file is repeatedly fed to Jack, so that the parser generates itself 16 times and produces an output of 1.56 MB.

Similar to the example discussed in the previous chapter, escape analysis eliminates 15 sites that create an enumeration object, e.g. for the iteration of BNF productions. This saves a total of 6,399 object allocations.

Apart from that, the benchmark mostly benefits from stack allocation of string buffers. Based on the input grammar, Jack creates a parser by assembling small snippets of Java code. 82 sites allocate a string buffer, 49 of them on the stack. The compiler inlines the `append` method and removes synchronization. This produces 340,894 stack objects at run time and eliminates 1,031,843 locks.

Table 7.4 provides the statistical data for the `jack` benchmark. The compiler inlines 395 synchronized and 110 large methods. 102,384 bytes of objects are replaced by scalars, and 5.22 MB are allocated on the stack. No arrays are optimized.

	static numbers	dynamic numbers
object allocations eliminated	15 (3.11%)	6,399 (0.20%)
object allocations on the stack	54 (11.18%)	342,023 (10.92%)
object allocations on the heap	414 (85.71%)	2,784,568 (88.88%)
synchronizations removed	185 (42.92%)	1,031,843 (6.66%)
synchronizations retained	233 (54.06%)	6,976,473 (45.02%)
synchronized methods	13 (3.02%)	7,486,681 (48.32%)

Table 7.4: Statistics for one run of the jack benchmark

A detailed inspection of the benchmark demonstrates how the implementation of the Java system classes influences the impact of escape analysis. The method `RunTimeNfaState.Move` looks up a character `c` in the hash table `charMoves`:

```
... = (RunTimeNfaState) charMoves.get(String.valueOf(c));
```

The `get` method expects the key to be an object, so the character is converted into a string via `String.valueOf`. This method is implemented as shown below. It creates a new string from a character array with `c` as the only element. Both the `valueOf` method and the `String` constructor can be inlined, so that the string object and the fixed-sized array do not escape so far.

```
public static String valueOf(char c) {
  char data[] = {c};
  return new String(0, 1, data);
}
```

`Hashtable.get` compares the newly created string with the keys of all entries `e` in the hash table. Since the compiler does not know the dynamic type of `e.key`, it fails to bind the `equals` method statically and consequently treats the `key` parameter as escaping globally.

```
if ((e.hash == hash) && e.key.equals(key)) {
  return e.value;
}
```

In contrast to `e.key`, the dynamic type of `key` is known to be `String`. When we rewrite the second part of the above condition to `key.equals(e.key)` and adjust the maximum inline size, the compiler inlines `String.equals`. Although `key` cannot be replaced by scalars because it is compared with `e.key` for identity, it can be allocated on the stack.

The small modification of `Hashtable.get` would increase the numbers of stack-allocated objects and arrays in the jack benchmark by 1.34 million each. However, it might break existing classes with an asymmetric implementation of the `equals` method.

7.2.4 Overall Performance

The SPECjvm98 suite provides several optimization opportunities for escape analysis. As the benchmark programs have different characteristics, they either benefit more from scalar replacement of fields, stack allocation or synchronization removal.

Table 7.5 summarizes the results of escape analysis for the complete SPECjvm98 suite. Note that in contrast to the time measurements below, these numbers were counted by executing the benchmarks only once and not until stability in run time is reached.

	static numbers	dynamic numbers
object allocations eliminated	45 (2.77%)	6,744,517 (32.87%)
object allocations on the stack	146 (8.98%)	855,193 (4.17%)
object allocations on the heap	1435 (88.25%)	12,917,314 (62.96%)
array allocations eliminated	3 (0.72%)	1,019,487 (11.13%)
array allocations on the stack	3 (0.72%)	628,031 (6.85%)
array allocations on the heap	413 (98.56%)	7,515,276 (82.02%)
synchronizations removed	432 (46.75%)	2,002,726 (2.16%)
synchronizations retained	454 (49.14%)	18,989,868 (20.49%)
synchronized methods	38 (4.11%)	71,674,222 (77.35%)

Table 7.5: Statistics for the complete SPECjvm98 suite

Scalar replacement eliminates the allocation of 6.7 million objects with a total size of 155 MB. 855,193 objects or 25.22 MB are allocated on the stack. The compiler inlines 373 methods whose size exceeds the usual threshold. 844 of the 8,447 inlined methods were synchronized.

Table 7.6 presents the elapsed times for the benchmarks when they are compiled by the client compiler with escape analysis disabled or enabled, or by the server compiler, which does not use escape analysis. The ratio represents the speedup achieved by escape analysis.

	without EA		with EA		ratio		server compiler	
	slowest	fastest	slowest	fastest	slowest	fastest	slowest	fastest
mtrt	1.391	1.094	1.187	0.828	1.172	1.321	2.000	1.062
jess	1.922	1.656	1.938	1.656	0.992	1.000	2.421	1.485
compress	6.000	5.922	6.000	5.922	1.000	1.000	5.641	5.516
db	11.922	11.766	11.813	11.672	1.009	1.008	11.422	10.906
mpegaudio	2.984	2.703	2.984	2.703	1.000	1.000	3.656	2.250
jack	3.266	2.969	3.219	2.875	1.015	1.033	7.625	2.828
javac	5.657	4.500	5.562	4.360	1.017	1.032	12.641	3.969
Mean	3.756	3.332	3.655	3.169	1.028	1.051	5.250	3.039

Table 7.6: Elapsed times for SPECjvm98 benchmarks (in seconds)

The bar chart in Figure 7.1 illustrates the scores that were computed for the elapsed times above. Light bars refer to slowest runs and dark bars to fastest runs. Higher means better.

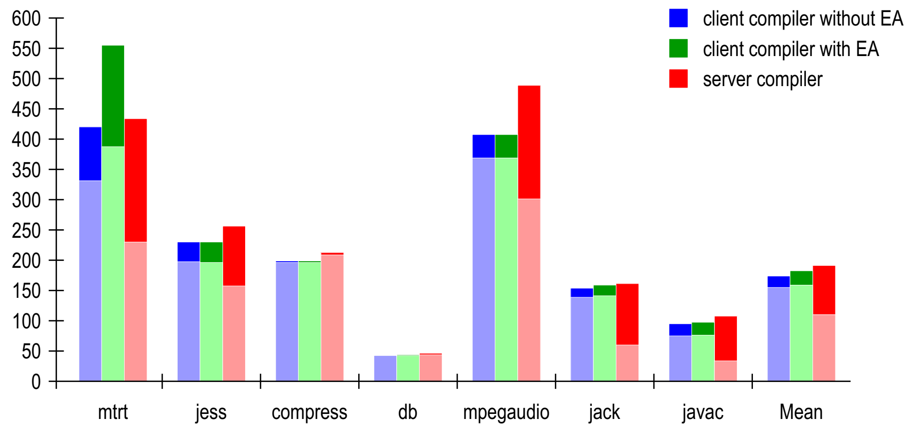


Figure 7.1: Comparison of SPECjvm98 scores

The `mtrt` benchmark shows the highest performance gain. It uses a lot of short-lived data structures, such as points and vectors, whose allocation can be eliminated by scalar replacement. We do not only achieve a speedup of 32.1% compared to the client compiler without escape analysis, but even outperform the server compiler both in the slowest and the fastest run.

The `jack` benchmark primarily benefits from stack allocation of string buffers and synchronization removal. Although more than 1 million object locks are removed per run, the speedup of 3.3% is smaller than the one achieved for `mtrt`, because the string buffers cannot be eliminated by scalar replacement.

Escape analysis also improves the performance of `javac`. The server compiler still achieves a higher score for the fastest run, but also a lower score for the slowest run. Escape analysis reduces the gap between client compiler and server compiler, at a higher compilation speed.

For `compress` and `mpegaudio`, fast memory allocation is not as crucial as for the other benchmarks. They create an insignificant number of objects and arrays and thus do not provide any optimization opportunities for scalar replacement or stack allocation. As far as the `db` benchmark is concerned, all three compilers yield similar scores.

Although the compilation speed for SPECjvm98 decreases from 214,714 bytes/s by 11.9% to 189,160 bytes/s when escape analysis is enabled, the optimizations yield an average speedup of 2.8% for the slowest runs and 5.1% for the fastest runs. This shows that the additional optimizations outweigh the increase in compilation time. For comparison, the speed of the server compiler for SPECjvm98 is 22,332 bytes/s.

7.3 SciMark 2.0

SciMark 2.0 [75] was developed by Roldan Pozo and Bruce Miller at the National Institute of Standards and Technology (NIST). The goal was to better understand the JVM and JIT behavior of various Java platforms. Similar to the Java Linpack benchmark, it measures the performance of numerical computations occurring in scientific and engineering applications.

SciMark is a composite benchmark. It consists of five kernels that deal with fast Fourier transform, Jacobi successive over-relaxation, Monte Carlo integration, sparse matrix multiplication and LU matrix factorization. Each kernel reports an individual score in approximate Mflops (millions of floating-point operations per second).

The five kernels primarily evaluate the performance of floating-point computations and allocate none or few objects. Only the Monte Carlo integration provides an opportunity for escape analysis and scalar replacement of fields. It estimates the value of π by approximating the area of a circle. The core method of this benchmark is `MonteCarlo.integrate`:

```
static double integrate(int Num_samples) {
  Random R = new Random(SEED);
  int under_curve = 0;
  for (int count = 0; count < Num_samples; count++) {
    double x = R.nextDouble();
    double y = R.nextDouble();
    if (x * x + y * y <= 1) {
      under_curve++;
    }
  }
  return ((double) under_curve / Num_samples) * 4.0;
}
```

It contains a loop which selects arbitrary points in the unit square and counts how many of them are within a radius of 1 or less. Each iteration generates two random numbers for the coordinates. The call tree looks as follows:

```
double MonteCarlo.integrate (66 bytes)
    Random.<init> (76 bytes)
        void Random.initialize (125 bytes)
    synchronized double Random.nextDouble (124 bytes)
    synchronized double Random.nextDouble (124 bytes)
```

By default, the maximum inline size is 35 bytes. The constructor of `Random` has a size of 76 bytes and is thus too large to be inlined. An analysis of its bytecodes, which recursively examines `Random.initialize`, produces interprocedural escape information. No bytecode analysis is required for the method `Random.nextDouble` because it is compiled before `MonteCarlo.integrate`. The compiler reveals that the `Random` object does not escape and can be allocated on the stack.

`MonteCarlo.integrate` calls the constructor of `Random`, which in turn calls the method `initialize`. Since the maximum inline size is cut by 10% with every level, the original threshold must be at least 125 / 0.9 ≈ 139 bytes to force all methods to be inlined. Then the allocation of the `Random` object is eliminated and synchronization is removed from `nextDouble`.

In the method `initialize`, an array with the constant length 17 is created and assigned to a field of the `Random` object. The array cannot be eliminated because it is not accessed with constant indices only, but it can be allocated on the stack. Due to the elimination of the `Random` object, the array is used directly instead of being loaded from a field. The compiler can generate array bounds checks which compare the index with the constant 17 instead of retrieving the array length from the array header.

The benchmark repeatedly calls `MonteCarlo.integrate` and doubles the parameter `Num_samples` every time, until at least 2 seconds are spent in one invocation. Provided that the maximum inline size is chosen large enough, the method is called 26 times if escape analysis is disabled, and 27 times if it is enabled. The first 11 invocations are interpreted, then the method is compiled due to the large amount of taken backward branches. Scalar replacement eliminates the allocation of 16 objects with a total size of 320 bytes. Besides, 268,431,360 object locks are saved, because the body of `nextDouble` is executed that often in machine code.

Only the final method invocation decides on the overall score, which is computed from the number of iterations and the elapsed time. Without escape analysis, 33,554,432 iterations are executed in 2.00 seconds, which results in a score of 67.11 Mflops. With escape analysis, a score of 88.10 Mflops is reported after twice as many iterations have been executed in 3.05 seconds.

8 Related Work

Lifetime analysis of dynamically allocated objects has traditionally been used for compile-time storage management. Over the years, a lot of algorithms with different characteristics and goals evolved. Although escape analysis focused on functional languages in the beginning, it revived with the spread of Java programs. We give an overview of various approaches to escape analysis, similar optimizations and other related work.

8.1 Overview

This section starts with a short outline of the historical background and the first algorithms associated with escape analysis, before it puts escape analysis in relation with pointer analysis. Finally, it deals with the most popular implementations of escape analysis for Java programs.

8.1.1 History of Escape Analysis

Cristina Ruggieri and Thomas P. Murtagh developed an interprocedural analysis that aimed at determining the lifetime of all objects created by a certain expression [79]. Influenced by the concept of a generational heap, they partition the heap into sub-heaps, each of which stores the objects whose lifetime is guaranteed to be contained in the lifetime of a particular procedure. When the procedure terminates, the space occupied by the objects in the corresponding sub-heap can immediately be reclaimed.

Young Gil Park and Benjamin Goldberg introduced the term *escape analysis* for determining which parts of a list do not escape a function call in a higher-order functional program [72]. The results are used for stack allocation and in-place reuse, an optimization that reuses heap space of dead objects without invoking the garbage collector. If none of these two optimizations is applicable, objects are grouped into a contiguous block of memory that can be reclaimed at once. This allows the deallocation of larger memory segments and reduces run-time overhead by avoiding the traversal of individual objects.

Mads Tofte and Jean-Pierre Talpin proposed a store which consists of a stack of regions [86]. Their optimization determines the lexically scoped lifetime for all run-time values in a functional language, including function closures, and puts them into regions. Region inference and effect inference are used to reveal where regions can be allocated and deallocated. The analysis can both distinguish the lifetimes of different invocations of the same function and handle recursive data types.

Alain Deutsch examined and improved the complexity of escape analysis [30]. For first-order programs, which do not store functions in data structures, pass them as parameters or return them as values, he proposed an almost linear analysis which computes the same information as the analysis by Park and Goldberg. A prototype was implemented for ML and integrated in the CSL compiler. He also showed formally that escape analysis for second-order programs, which allow first-order functions as parameters, has an exponential time complexity.

8.1.2 Pointer Analysis

Escape analysis and pointer analysis are closely related and overlapping fields of research. Pointer analysis makes assumptions about the possible run-time values of a pointer. It determines to which locations a variable may point (*points-to analysis*) or whether two pointers refer to the same location (*alias analysis*). This information is used both for optimizations and for debugging.

In the past years, several analysis algorithms and compiler optimizations that rely on points-to information were published. Additionally, points-to analysis was formalized in the frameworks of iterative data flow analysis, set constraints, graph reachability, and logic programming [18]. The algorithms differ in the way they use control flow information, consider the calling context, distinguish elements of an aggregate, or require a view of the whole program.

Efficient algorithms for pointer analysis give approximate results and probably report too large points-to sets. Constraints on the precision depend on the application of the analysis [45]. Optimizations usually establish an upper bound on precision, because more precision involves additional cost that does not necessarily pay off. If the analysis is used for error detection and program understanding, there is a lower bound on precision below which pointer information is useless. Bjarne Steensgaard developed a widespread algorithm with an almost linear time complexity [83]. Lars O. Andersen's analysis [6] is less efficient but more precise.

Although a great deal of research effort is spent on the analysis of C programs [39], pointer analysis is equally important for the Java language. All references are heap-directed, so alias information can help avoid reaccessing the same memory location. Unfortunately, some Java features, such as dynamic class loading, reflection, and native methods, make pointer analyses difficult to develop. Martin Hirzel et al. examined Andersen's analysis and showed how it can be enhanced to analyze the full Java programming language [46].

8.1.3 Escape Analysis for Java

Jong-Deok Choi et al. from the IBM T. J. Watson Research Center created a framework for escape analysis that can be applied to Java programs both in a flow-sensitive and a flow-insensitive manner [22]. The analysis is based on a program abstraction called *connection graph*, which captures the relationship between objects and object references (see Section 8.2.1). It allows summarizing the effects of a method on the escape states of its parameters. The summary information obtained for a callee is independent of the calling context and can be used to update the connection graph of each caller. A reachability analysis on the connection graph reveals if an object is method-local or thread-local. The analysis was implemented in a static Java compiler to allocate objects on the stack and remove unnecessary synchronization operations.

Bruno Blanchet extended the Java-to-C compiler turboJ by an escape analysis written in Java [12]. He also gives a correctness proof. The analysis transforms Java code into SSA form, builds equations and solves them with an iterative fixpoint solver. Escaping parts of data structures are represented via integers (see Section 8.2.3). The results are used for stack allocation and synchronization removal. Regarding dynamic class loading, the implementation either assumes that all suitable classes found on the class path may be loaded or it relies on information provided by the user.

Jeff Bogda and Urs Hölzle developed a flow-insensitive whole-program analysis to remove unnecessary synchronization in Java [13]. An object is regarded as thread-local if it is reachable only from local variables and fields of other thread-local objects. Objects reachable via more than one level of field access are not optimized, which allows the analysis to ignore recursive data structures. At a call site, the statements within the callee are examined if they have not been analyzed yet. In case of a recursive call, the entire process iterates until no change occurs. Each optimizable allocation site is finally transformed to create instances of a new class, which extends the original class and overrides synchronized methods with unsynchronized copies.

The optimization by Erik Ruf from Microsoft Research even eliminates synchronization on objects that are reachable from static fields, but accessed only from a single thread [78]. It relies on an equivalence-based representation similar to our equi-escape sets, in which potentially aliased values are forced to share common representative nodes. This eliminates the need for fixpoint operations during the analysis. The optimization is applied to statically compiled programs and was implemented in the Marmot native compilation system for Java which does not support dynamic class loading.

David Gay and Bjarne Steensgaard implemented another escape analysis algorithm for Marmot [38]. The analysis computes two properties for each local variable. The first one specifies if the variable holds a reference that escapes due to an assignment or a throw statement, and the second one if the reference escapes by being returned from the method. Each statement of a program may impose constraints on these properties that must be solved. References are not tracked through fields, so any reference assigned to a field is assumed to possibly escape from the method in which the assignment occurs. Non-escaping objects are eliminated or allocated on the stack. However, no attempt is made to allocate arrays on the stack.

Jeff Bogda and Ambuj Singh adapted Ruf's escape analysis to operate incrementally and on-the-fly [14]. The analysis is able to work with an incomplete call graph and to modify previous results as the call graph expands. The authors evaluated three strategies of when to initiate the analysis, and whether to make optimistic or pessimistic assumptions for optimization. The first approach performs an interprocedural analysis as soon as the program starts to run, whereas the second one delays the analysis until the run-time system has seen a portion of the program's execution. The third approach reuses analysis results from previous executions in order to avoid the repeated analysis of a method binding. The measurements suggest to reuse analysis results and to delay the initial analysis until the end of the first execution.

Jonathan Aldrich et al. compared static whole-program analyses that detect and remove the following kinds of locks [5]: *thread-local locks* (accessible by a single thread), *enclosed locks* (protected by the lock of another object), and *reentrant locks* (protected by the lock of the same object). The analyses compute the set of unnecessary synchronization operations using the Vortex research compiler, and then optimize Java class files directly via a binary rewriter.

John Whaley and Martin Rinard developed a combined pointer and escape analysis algorithm for Java programs [93]. It operates on *points-to escape graphs* that characterize how local variables and fields in objects refer to other objects. The graphs distinguish between objects and references created within an analyzed region and those created in the rest of the program. Therefore, they enable a flexible analysis of arbitrary regions of complete or incomplete programs, obtaining complete information for objects that do not escape the analyzed regions. The results are used for stack allocation and synchronization removal.

Frédéric Vivien and Martin Rinard observed that almost all of the objects are allocated at a small number of allocation sites [89]. For this reason, they present an algorithm which incrementally analyzes only those parts of a program that may deliver useful results. Their algorithm performs an incremental analysis of the neighborhood of the program surrounding selected allocation sites. It first skips the analysis of all potentially invoked methods, but maintains enough information to incorporate the results of analyzing the methods when they turn out to be useful. The modified MIT Flex compiler generates methods that allocate objects in the current stack frame or in the frame of a direct or indirect caller.

Together with Alexandru Sălcianu, Martin Rinard extended the points-to escape graphs to *parallel interaction graphs*, which maintain precise points-to, escape, and action ordering information for objects accessed by multiple threads [80]. Based on the results, the MIT Flex compiler is able to statically verify that multithreaded programs use region-based allocation correctly. Region-based allocation allows a program to allocate all objects created by a computation in a specific region and deallocate them when the computation finishes.

Matthew Q. Beers, Christian H. Stork and Michael Franz were concerned about the costs of escape analysis during just-in-time compilation [10]. In contrast to our approach, they suggest to perform the analysis ahead of time and ship its results as code annotations. The analysis is less precise than traditional escape analyses, but its results can efficiently be verified by the run-time system to ensure that the annotations have not been altered. In an environment that supports dynamic class loading, parameters of methods that cannot be bound statically are assumed to escape. The annotations are added to standard Java class files, but the authors refrained from modifying a VM to actually allocate objects on the stack, because they do not guarantee that all objects fit in a fixed-sized frame.

Diego Garbervetsky and his colleagues proposed an instrumentation of Java programs for scoped memory management [37]. The idea is to allocate objects in regions, which are created at the beginning of a method and freed when the method returns. A pointer and escape analysis is used to associate memory regions with methods in such a way that no dangling references occur. The program instrumentation does not specify a unique region where an object is allocated, but rather a set of regions corresponding to methods on the call stack. This allows to control at run time where the object is actually allocated without changing the source-level instrumentation.

Most existing approaches to escape analysis are implemented in static compilers or disregard dynamic class loading. The primary goals are stack allocation of objects and synchronization removal, whereas scalar replacement of fields is done rarely. To the best of our knowledge, our approach is the first that performs all three optimizations in a dynamic compiler and allows to reallocate and relock objects when dynamic class loading invalidates the optimized machine code.

8.2 Program Abstraction

Escape analysis is rarely performed directly on Java source code or on the bytecodes. Instead, a program abstraction is built which is capable of associating properties with references and objects. The analysis is then applied to this abstraction.

The program abstraction we chose for our analysis is influenced by the SSA form and the layout of the HIR. An object is represented by the HIR instruction which allocates the object. The set of referenced objects is stored as a list of instructions, and dependencies between objects are modeled via equi-escape sets. This chapter deals with program abstractions used in other implementations.

8.2.1 Connection Graph

Jong-Deok Choi et al. suggest a simple program abstraction called *connection graph*, a directed graph which captures the relationship between object references and objects [22]. The nodes represent either an object or a reference variable. They are connected by different kinds of edges:

- A *points-to edge* connects a reference node `p` with an object node to indicate that the variable `p` may point to the object.
- A *field edge* connects an object node with a reference node `f` if `f` represents a field of the object.
- A *deferred edge* is an edge between two reference nodes `q` and `p`. It denotes that the variable `q` may reference any object that is pointed to by `p`.

The connection graph is built via an abstract interpretation of the bytecodes. Storing an object reference into a variable introduces a new points-to edge. If a reference variable is assigned to another one, the two corresponding reference nodes are connected with a deferred edge.

Figure 8.1 shows an example for a simple connection graph. Object nodes are drawn as boxes and reference nodes as circles. A solid edge from a circle to a box is a points-to edge, and a dashed arrow between two circles is a deferred edge.

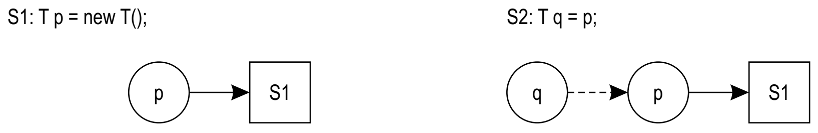


Figure 8.1: Example for a simple connection graph

Reference nodes for a field are created lazily when a variable `x` is assigned to a field `p.f`. For each object node pointed to by `p`, a field reference node is created and connected to the reference node `x` by a deferred edge (see Figure 8.2). If `p` does not point to any object node, a *phantom node* is added which represents an object created outside the current method.

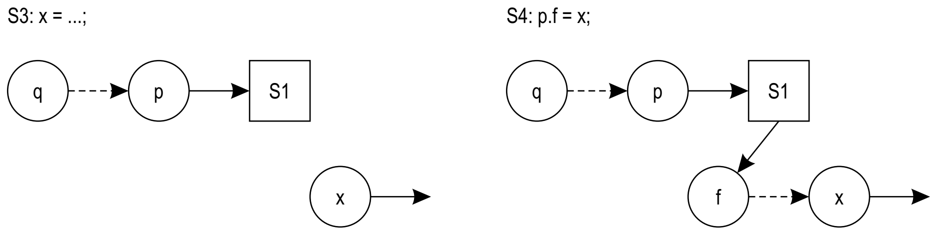


Figure 8.2: Creation of a field node in the connection graph

The connection graphs for alternative control flow paths are modified independently from each other. Deferred edges improve the efficiency of the analysis by delaying and thus reducing the number of graph updates. At control flow join points, the incoming connection graphs are merged. The result of the merge is the union of nodes and edges of the original graphs.

Escape analysis is reduced to a reachability problem over the connection graph. Each node in the graph has an escape state associated with it. Static field nodes, for example, are initialized as escaping globally. An allocation site can create objects on the stack if the corresponding object node is not reachable from an escaping node. When the graph is modified or merged with another graph, the escape states are adjusted. Loops are handled by iterating over the data flow solution until it converges.

8.2.2 Points-To Escape Graph

The analysis by John Whaley and Martin Rinard is based on a program abstraction called *points-to escape graph* [93]. Similar to a connection graph, it models how local variables and fields in objects refer to other objects. It also stores information about which objects allocated in one region of the program escape and can be accessed by another region. A points-to escape graph consists of

- *inside edges* for references created inside the currently analyzed region,
- *outside edges* for references created outside the current region,
- *inside nodes* for objects allocated inside the current region and accessed only via references created inside that region, and
- *outside nodes* for objects allocated outside the current region or accessed via references created outside the region.

There is one inside node for each allocation site within the current region. It represents all objects created at this site. The set of outside nodes includes nodes for objects that are referenced by a formal parameter, stored in a static field, loaded from a field or returned from the method. The node for an array has a single field that represents all of the array's elements, so no distinction is made between different elements of the same array.

Figure 8.3 shows the points-to escape graph at the end of the `insert` method. Solid circles and arrows represent inside nodes and inside edges, whereas dashed circles and arrows represent outside nodes and outside edges.

```
Elem insert(Object obj) {
    Elem m = this;
    while (m != null) {
        if (...) return this;
        m = m.next;
    }
    return new Elem(obj, this);
}

Elem(Object obj, Elem next) {
    this.val = obj;
    this.next = next;
}
```

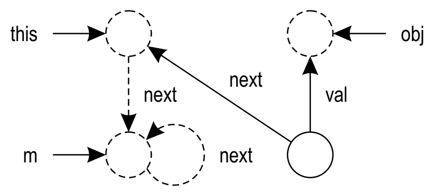


Figure 8.3: Example for a points-to escape graph

The variable `this` points to the receiver node, while `obj` points to the parameter node. Both nodes are outside nodes. The `next` edge from the receiver node points to a node that represents all objects referenced by the `m.next` field. As the references are traversed in a loop, there is also an edge from this node to itself. The only inside node represents the `Elem` object allocated within the `insert` method. It has an edge from the `next` field to the receiver node and from the `val` field to the parameter node. These are inside edges because the references are created during the execution of `insert`.

The algorithm uses a data flow analysis to generate a points-to escape graph at each point in a method. It constructs an initial points-to graph and then propagates it through the statements of the method. Most statements kill a set of inside edges before they add new inside and outside edges to the graph (see Figure 8.4 for an example).

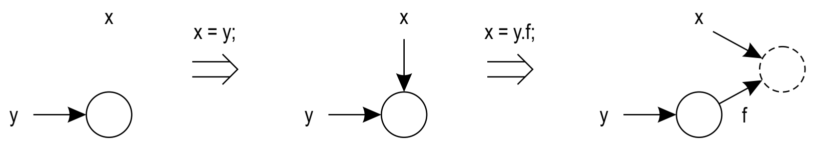


Figure 8.4: Effect of statements on the escape graph

An object is said to escape *directly* if it is passed as a parameter into the current method, written into a static field, passed as a parameter to a callee that has not been analyzed yet, or if it is a thread object. If an object is reachable from a directly escaping object via a sequence of references, it escapes. Otherwise it is said to be *captured* and can be allocated on the stack.

8.2.3 Escape Context

It would be too coarse to consider data structures as a whole, because sometimes only a part of a data structure escapes. For example, if `obj.elem` escapes, this does not mean that `obj` escapes. Bruno Blanchet uses integers in his analysis to represent the escaping part of an object. This integer is said to be the *escape context* of the object [12].

Traditionally, the escaping part of an object is expressed via a set of access paths from the object to the fields or elements that escape. The replacement of such a set by an integer leads to a more efficient analysis, because integers can be manipulated faster than graphs. Let the height of a type be the smallest integer such that

- the height is 0 for primitive types and greater than 0 for reference types,
- the height of a type is greater or equal the height of its subtypes,
- greater or equal the height of each element type, and
- strictly greater than the height of each element type if the type definition is acyclic (i.e. no object may contain an object of the same type).

The escape context of an object is computed as the height of its escaping part. If an object escapes, all contained elements are seen to escape as well, because the context of the object is greater or equal the height of each element type. If the escape context for an object is smaller than the height of its type, the object is known to not escape and can be allocated on the stack.

Figure 8.5 shows an example for the height of types. Assume that `list` denotes a `NodeList` and that `list.elem.info` escapes. The escape context for `list` equals the height of the type `Entry`, which is 2. It is greater than the height of `Object`,

```
class NodeList {
    NodeList next;
    Node     elem;
}

class Node {
    NodeList sons;
    Entry    info;
}

class Entry {
    Object   elem;
    Object   key;
}
```

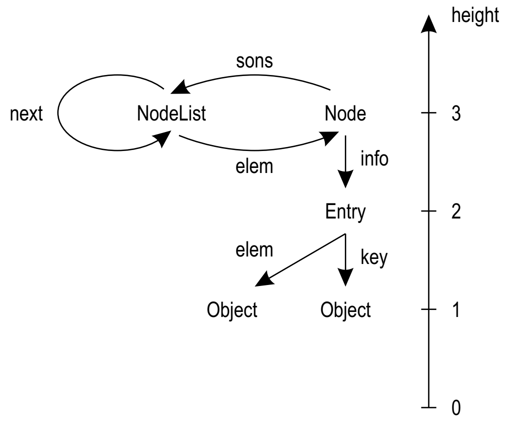


Figure 8.5: Example for the height of types

but smaller than the height of `NodeList`. All instances of `Entry` and `Object` in the list are regarded as escaping, but the `list` object itself can be allocated on the stack.

This representation is in general not able to distinguish different fields of the same object. However, the information for the top of data structures is precisely represented. Since the top of a data structure is the part that can most often be allocated on the stack, the representation is well adapted for this field of application. Inside complex data structures, precision is lost, but also stack allocation is less probable.

The computation of escape information relies on a *bidirectional* propagation. A backward analysis first marks the method result as escaping and then propagates the escape state to what is read to build the result. However, it cannot take into account that a value escapes because it is stored into an object. At the point of the assignment, it is not known for example whether the object is a parameter or not. Therefore, a forward analysis is performed to deal with values that are stored in static fields, in parameters or in the result of the method.

8.2.4 Alias Set

The analysis by Erik Ruf [78] takes three phases. At first, it identifies thread allocation sites and determines which methods may be executed by the threads. The second phase computes which threads synchronize on a value. Finally, the results are used to remove or simplify synchronization operations.

Erik Ruf models aliasing in a flow-insensitive manner by grouping potentially aliased expressions into equivalence classes. He stores synchronization behavior as attributes of these classes. The optimization represents run-time values as *alias sets*. Non-reference values are associated with the special set $\perp$. The alias set for a reference value is a tuple that consists of

- a *field map* which maps fully qualified instance field names to alias sets for the corresponding field values,
- a property *synchronized* to remember if the value may be the target of a synchronization operation,
- a set *syncThreads* of threads that may synchronize on the value, and
- a property *global* that specifies if the value escapes, i.e. is reachable from a reference constant or a static field.

In order to create and update alias sets, statements that modify reference variables or values are considered. If a reference variable is assigned to another one, the corresponding alias sets are merged. The attributes of the resulting alias set are the union of the input attributes. Joining two field maps causes alias sets for field names present in both maps to be merged.

The assignment of field `x.f` to a variable `y` is treated in a similar way. The alias set for `f` is retrieved from the field map for `x` and merged with the alias set for `y`. If a variable represents the result of a phi function, its alias set is merged with the alias set for each operand of the phi function.

A synchronization operation sets the *synchronized* property of the alias set for the object to be locked. If the alias set is *global*, the allocation sites of all threads that probably call the current method are added to the *syncThreads* property. Merging a non-global, synchronized alias set with a global one causes the *syncThreads* attribute of the result to be augmented with the set of threads associated with the current method.

Synchronization on an object is eliminated if the *syncThreads* set of the corresponding alias set is empty or contains a single thread that is executed at most once. So even if an object escapes, synchronization on it is removed unless it may be locked by more than one thread.

8.3 Interprocedural Analysis

Interprocedural analysis is only reasonable if it avoids reanalyzing a method at each of its call sites. On the one hand, this means that the analysis of a method must produce escape information that is independent of the method's calling context. On the other hand, it does not make sense to retain information about a method unless it concerns parameters or return values.

The following sections deal with different approaches to interprocedural escape analysis. Each of them refers to one of the program abstractions discussed in the previous chapter.

8.3.1 Phantom Nodes

The connection graph (see Section 8.2.1) was especially designed to support interprocedural escape analysis. It summarizes the effects of a method independent of the calling context, so that the same summary information can be used to update the connection graphs of different callers.

At the entry of a method, the connection graph is extended by a node for each formal parameter of reference type. The receiver of an instance method appears as the first parameter. From each of these nodes, a deferred edge points to a separate *phantom node* which represents the actual parameter created outside of the current method. The phantom nodes serve as anchors for the summary information that is generated when the analysis of the method has finished. If a new object is assigned to a formal parameter, the deferred edge to the phantom node is deleted. Future operations on the formal parameter do not affect the escape state of the actual one. This is the reason why summary information is built from phantom nodes instead of nodes for formal parameters.

The return of a reference to the caller is modeled as an assignment to a special phantom variable similar to a formal parameter. Multiple return statements are handled by merging the corresponding values. Any object thrown as an exception is conservatively treated as escaping globally.

A reachability analysis on the connection graph finally yields the subgraph of globally escaping nodes, which can be collapsed into a single *bottom node*, and the subgraph of nodes that escape via a parameter. The union of the two subgraphs is called *non-local subgraph* and represents the summary information of the analyzed method. It does not consider non-escaping nodes. These are used to decide whether an object created by the current method is stack-allocatable or not.

At a method invocation site, each actual parameter is matched to the corresponding phantom node of the callee. After the method invocation, the callee's summary information is mapped back to the caller's connection graph. This includes the adjustment of escape states and the creation of new nodes and edges. If the callee is a virtual method, the caller's connection graph is updated with the summary information from each possible target at that site, effectively merging the corresponding graphs.

The analysis represents the relationship between callers and callees in a *program call graph*. In order to handle cycles due to recursion, the algorithm iterates over the nodes in strongly connected components of the call graph until the data flow solution converges. If no convergence is reached within a maximum number of iterations, all nodes for actual parameters and return values of methods involved are marked as escaping globally.

An object that escapes via a parameter or return value is not stack-allocatable, but it may be thread-local depending on what the caller does with it. To identify such thread-local objects, information needs to be propagated from the caller to its callees in a separate top-down pass over the program call graph. Afterwards, an object in a method is marked as escaping its thread of creation if it escapes the thread in any caller of that method. If this step is omitted, all objects that escape the method will have to be conservatively regarded as escaping the thread.

Since Java allows a native method to perform arbitrary operations on a Java object accessible through the *Java Native Interface* (JNI), any object passed to a native method is considered as escaping the current thread. More generally, objects passed to methods whose body cannot be analyzed are marked as escaping globally. This also includes methods of dynamically loaded classes because they probably make an object reachable for other threads.

8.3.2 Incremental Escape Analysis

John Whaley and Martin Rinard developed a partial program analysis for stack allocation and synchronization removal. It does not only analyze a method independently of its callers, but is also capable of analyzing a method independently of methods that it may invoke. At each method invocation site, the analysis has the option of skipping or analyzing the site. If it skips the site, it marks all of the parameters as escaping.

Frédéric Vivien and Martin Rinard extended the base algorithm to an *incremental analysis* [89]. In addition to points-to information, the analysis records how objects escape the currently analyzed region of the program. If the analysis of a method is skipped, escape information can be supplemented later when it becomes desirable. The goal is to analyze just enough of the program to capture objects of interest.

When the algorithm analyzes a method, it records call sites and actual parameters at each site, but ignores the invoked methods. At the end, a particular allocation site is chosen in an attempt to capture objects allocated at this site. The algorithm repeatedly selects a skipped call site through which the objects escape and analyzes the methods potentially invoked at this site. In order to avoid reanalyzing the same method, cached data is reused if available. The resulting points-to escape graphs are integrated into the graph for the current method. As soon as the chosen allocation site is captured, the algorithm moves on to the next site until a configurable amount of time has expired.

The incremental analysis is guided by a policy to find and analyze allocation sites that can be captured quickly and promise a large optimization payoff. At each step, the policy can invest analysis resources in one of several allocations sites. Its decision is based on empirical data, the current analysis result and the number of objects allocated at each site in a previous profiling run. As the analysis proceeds and more information about the allocation sites is available, the quality of the decisions improves.

If an object escapes the allocating method but is recaptured within a direct or indirect caller, the object may be allocated in the stack frame of the method that captures it. The compiler can either inline the allocating method into the caller or generate a specialized version of the method, which takes a pointer to preallocated space in the caller frame and initializes the object at the given location instead of allocating it on the heap. Of course, this requires that the allocation site is executed at most once per call path that leads from the capturing to the allocating method.

8.3.3 Context Transformers

Bruno Blanchet uses integer contexts (see Section 8.2.3) to determine if objects can be allocated on the stack. During the analysis of a method m, the reference scope is m. The parameters and the result of the method must be regarded as escaping, because they can be accessed outside the current scope.

Interprocedural analysis decides if an object can be allocated on the stack in a method m' that calls m. Now the reference scope is m'. The parameters and the result of m do not necessarily escape from the caller m', depending on what m' does with the objects.

So the method m needs to be analyzed in several calling contexts. To avoid reanalyzing the method, the analysis is modeled as a function of the calling context, which is represented by the escape contexts of the parameters and the result of m. The escape analysis is said to be context-sensitive.

Object names are *context transformers*, i.e. functions from contexts to contexts. They take the escape contexts of the parameters and of the method result and yield the escape context associated with a concrete object. The receiver is treated as any other parameter of the method.

8.4 Results

Jong-Deok Choi et al. evaluated their escape analysis on a set of 10 benchmarks. In the `toba` and `wingdis` benchmark, more than 70% of the objects are allocated on the stack, and a high percentage of synchronization operations is removed. This reduces execution time by about 15%. The `jgl` benchmark even yields a speedup of 23%. All other programs show execution time reductions between 2% and 7%. The measurements refer only to objects created in the user code, because the optimization of predefined classes would require a recompilation of the library code. The flow-sensitive and flow-insensitive variants do not differ significantly [22].

Bruno Blanchet implemented his analysis in a Java-to-C compiler. Experiments with seven benchmarks showed that 13% to 95% of the data are allocated on the stack and more than 20% of the synchronization is removed in most programs. The run time decreases by 43% for `dhry` thanks to stack allocation, 40% for `JLex` thanks to synchronization removal, and about 10% for `javac`, `turboJ` and `javacc`. The speedup comes more from the decrease of the garbage collection and allocation times than from improvements on data locality. The average overhead generated by the optimizations is 29% of the compilation time without stack allocation [12].

Erik Ruf tested his algorithm on five single-threaded and five multi-threaded programs. When the single-threaded programs are optimized, no synchronization operations are executed at run time. The speedup varies from 7% to 26%, only `jlex100` achieves a speedup of 159%. In `javac`, some operations are only partially removed to preserve notification semantics. Multi-threaded benchmarks do not become faster as a result of synchronization removal, except for `multimarmot`, which improves by 10%. Between 2.75% and 6.75% of the compilation time is spent on the analysis [78].

Jonathan Aldrich et al. compared the gains of removing thread-local, reentrant and enclosed locks. The thread-local analysis eliminates 64% to 99% of the synchronization operations in the single-threaded, and 0% to 89% in the multi-threaded benchmarks. The reentrant lock analysis tends to eliminate operations that are also removed by the thread-local analysis and thus has a small impact on most benchmarks. An exception is the `proxy` benchmark, which benefits from the removal of reentrant locks that are not thread-local. Similarly, 12% of the dynamic synchronization operations in `jlogo` are eliminated by enclosed lock analysis but not recognized by other algorithms. The overall speedups range from 0% to 53% [5].

Frédéric Vivien and Martin Rinard present statistics for the incremental pointer and escape analysis. It captures virtually the same number of objects as the whole-program analysis, but requires less time to do so. The two scientific programs `Barnes` and `Water` interact poorly with the conservative garbage collector, so stack allocation provides a speedup of 32% and 40%. Execution time for `JLex` is reduced by 9%, whereas the SPEC benchmarks hardly benefit from the optimizations [89].

8.5 Write Barrier Removal

Advanced garbage collection strategies usually expect the user program (often referred to as *mutator*) to interact with the collector. The compiler generates special machine instructions to inform the garbage collector that a pointer field is read or written. These machine instructions are called *read barriers* or *write barriers*, respectively.

Under certain conditions, e.g. if a field points into a certain area of the heap or was null before its modification, no barrier is required. These conditions depend on the implementation of garbage collection. If the compiler can prove statically that they always hold for a certain point in the machine code, it can eliminate the barrier. The resulting machine code is smaller and executes faster.

The optimizations described in this chapter neither perform scalar replacement nor stack allocation, but they incorporate a form of pointer or escape analysis to make assumptions about the current values of fields. While we eliminate write barriers as a side effect of stack allocation, the optimizations described in this chapter explicitly aim at barrier removal. The sections below give an overview of the different kinds of barriers and describe how they can be removed.

8.5.1 Kinds of Barriers

Generational garbage collection concentrates its efforts on the part of the heap where memory is most likely to be reclaimed. Young generations, which mainly contain temporary short-lived objects, are collected more frequently than older generations. If the garbage collector moves an object into an older generation, it keeps track of all pointers to younger objects in a remembered set. By treating them as root pointers, a young generation can be collected without having to traverse older ones.

When a pointer into a young generation is stored in an object after the object was moved into an older generation, the mutator must take care that the pointer is added to the remembered set. Therefore, pointer writes are protected with a *write barrier* (see Section 4.2). The barrier marks the modified heap area (*card*) as dirty and thus causes the garbage collector to examine it and update the remembered set. If the compiler can guarantee that a field always points to the same or an older generation, the write barrier may be omitted.

More sophisticated barriers are used in the context of an incremental garbage collector, which runs in parallel to the mutator to avoid the pauses caused by traditional *stop-and-collect* strategies. Incremental garbage collection has a different goal than generational collection. It demands guarantees for the worst-case performance, whereas generational collection attempts to improve the average pause time without considering the worst case [54].

Incremental garbage collection is usually described by means of a tricolor abstraction [32]. All objects start as unvisited, represented by the color *white*. When the garbage collector traces a pointer to a white object, the referenced object is marked as *grey* to indicate that it needs to be visited. When all immediate descendents of an object were processed, the object is colored *black* and will not be visited again. Thus objects become darker during the marking phase. The current marking cycle terminates as soon as no more grey objects exist. All reachable objects are black at this point. Any object left white is garbage and can be reclaimed.

The parallel execution of the mutator and the collector introduces a consistency problem: The mutator may unlink a white object from the object graph and write a pointer to it into a black object. In this case, the white object would never be visited and thus reclaimed at the end of the collection cycle although it is referenced. To avoid this problem, it is sufficient to ensure one of the following conditions:

- The mutator never sees a white object.
- A pointer to a white object is never written into a black object.
- The original reference to a white object is not destroyed.

The first condition involves a *read barrier*. As soon as the mutator attempts to access a white object, the object is marked as grey and visited by the collector. Since the mutator cannot read pointers to white objects, it cannot write them into black objects.

The second and the third condition can be ensured via *write barriers*, which are cheaper than read barriers because heap writes are less common than heap reads. Methods using write barriers are classified as either *incremental-update* or *snapshot-at-the-beginning* (SATB), depending on whether they prevent the creation of a pointer to a white object or the loss of the original reference [94].

Incremental-update methods record changes made by the mutator to the shape of the object graph. When the mutator attempts to install a pointer to a white object into a black object, one of the two objects is shaded grey. No action is required when a pointer to a white object is destroyed. Any white object that becomes garbage during the marking phase is reclaimed by the sweep phase within the same cycle.

SATB algorithms mark all objects reachable in a logical snapshot taken at the beginning of each collection cycle. Whenever a pointer is overwritten, the original reference is shaded grey. This prevents the object from being reclaimed for the case that a pointer to it is installed into a black object. As a result, objects that become garbage must wait until the next cycle to be reclaimed. The barrier can be eliminated if the compiler guarantees for a certain write that the original reference is null.

8.5.2 Removal of Generational Write Barriers

In the context of generational garbage collection, write barriers are used to keep track of pointers from old into young generations. This overhead has traditionally been accepted as part of the cost of a generational collector. Karen Zee and Martin Rinard implemented a flow-sensitive pointer analysis to identify field assignments that always create a reference from a young object to an older one [98]. Write barriers associated with such assignments can safely be removed by the compiler.

The intraprocedural analysis computes for each program position the set of variables that point to the most recently allocated (*method-youngest*) object. Statements that use one of the variables to write a field of this object do not require a write barrier. At the method entry and after each method invocation site, the analysis conservatively assumes that no variable points to the method-youngest object.

Two extensions augment the analysis with interprocedural information. The first extension records the types of objects that are allocated by callees invoked since the method-youngest object was created. If an object `obj`, whose type is not contained in the set, is stored in a field of the method-youngest object, the write barrier can be removed because `obj` was not created in one of the callees and is thus older than the method-youngest object. The second extension examines all call sites to find out if the callee is invoked only on method-youngest objects. In this case, the algorithm can assume at the callee's method entry that the `this` variable refers to the most recently allocated object.

To increase the effectiveness of the write barrier removal, the object allocation order is changed to match the direction of references between the objects. If an object is allocated in a constructor, the allocation is moved out of the constructor so that it occurs before the object under construction is allocated. Besides, single-linked lists and similar data structures are built bottom-up instead of top-down.

The reservation of a bit in the object header enables an optimistic write barrier removal. The bit is set when an object is created and cleared after completion of all statements that write a field of the object and for which the write barrier was removed. If the collector promotes an object into an older generation while the bit is set, it adds the object to a remembered set which is scanned for references into younger generations at the beginning of each collection. This way, write barriers can be removed even if objects are promoted out of order or allocated directly in an old generation.

8.5.3 Removal of SATB Write Barriers

The advantage of SATB over incremental-update marking is that new objects are allocated black and thus need not be examined by the current marking phase. However, the interaction between mutator and collector involves higher costs. While card-marking write barriers for incremental-update get away with two instructions per pointer write, SATB write barriers are more expensive. They need to read the pre-write value of a pointer, check that it is non-null and then add it to a buffer that is processed by the garbage collector [29].

If a field is null before it is assigned a new reference, no write barrier is needed because no object is unlinked from the object graph. V. Krishna Nandivada and David Detlefs from Sun Microsystems Laboratories developed two analyses to identify such assignments in a just-in-time compiler [69]. The first one does so for fields of objects, and the second one for elements of object reference arrays. In the majority of cases, the field or array element is null because it was cleared during object allocation and not modified since then. The first assignment is an *initializing write*, which does not require a write barrier.

If an object is not thread-local, the compiler cannot guarantee that one of its fields is null, because the object may be modified asynchronously by multiple threads. An escape analysis tracks the *thread-locality* of each reference. It computes two abstract object references for each allocation site *id*: $R_{id/A}$ represents a reference to the object most recently allocated at *id*, and $R_{id/B}$ summarizes all references to objects previously allocated at *id*. At each allocation site, attributes previously associated with $R_{id/A}$ are merged into $R_{id/B}$, before $R_{id/A}$ is associated with the newly allocated object.

The use of two reference values per allocation site enables a precise analysis. It computes the possible reference values that might flow into fields of objects before those fields are modified. If an instruction updates a field of a single abstract reference value that is unique, i.e. denotes a single run-time value, the old field value is replaced by the new one. Otherwise, the new value is merged into the previous contents via set union.

After a basic block was processed, the final program state of the block is merged into the initial state of each of the block's successors. A reference value may escape by being assigned to a static field, passed as a parameter to a method, or by being stored into the field of an object that is possibly non-thread-local. The analysis tracks the escape state of objects for each point in the program. Even if an object escapes, a write barrier can be eliminated as long as the write occurs before the object escapes.

8.6 Debugging and Deoptimization

The Java HotSpot client compiler performs optimizations based on assumptions about the class hierarchy. Therefore, dynamic class loading may invalidate the machine code of a compiled method later. If the method is currently being executed, it must be deoptimized before its execution can continue in the interpreter.

For this purpose, the state of the compiled method must be mapped to a state that the same method had produced if it had been executed in interpreted mode. Such a mapping is commonly used for debugging optimized machine code at the source code level, which has been a topic of interest for more than two decades [16]. This chapter describes related work for debugging optimized code and for deoptimization, because the basic concepts are similar.

8.6.1 Debugging Information

Zellweger defines two levels of debugger behavior in her thesis [99]: A debugger is said to provide *truthful behavior* if it detects that the executing program differs from the source program and admits that the exact answer to a debugging query cannot be given. It is said to provide *expected behavior* if it hides the effects of optimizations from the user. Debugging with expected behavior reduces debugging time and programmer confusion, but relies on information collected by the compiler.

Hennessy was one of the first to deal with symbolic debugging of optimized code [43]. He suggests that the compiler produces a set of labeled data flow graphs that represent code dependencies in both the optimized code and the original unoptimized code. The graphs are used by the debugger to identify variables whose values do not match those in the unoptimized code and to recover their real values. Wall et al. later point out errors in Hennessy's approach and give corrected algorithms [90].

Coutant et al. concentrate on optimizations that make symbolic source-level debugging most difficult [24], namely register promotion and assignment, loop variable elimination, and instruction scheduling. Similar to the debugging information of the Java HotSpot VM, they track a variable's values from memory through registers by describing the locations of the variable for certain ranges in the generated code.

Chambers et al. propose a layout for compiled SELF methods [21], akin to our nmethod data structure. In addition to the machine code, each method specifies the positions of references in the machine code that must be updated by the scavenger, a list of methods that the compiled code depends on, descriptions of inlined method scopes used to display source-level call stacks, and a mapping between bytecodes and program counter values.

8.6.2 Deoptimization

As we have seen, debugging of optimized code requires the compiler to communicate information to the debugger, but not all optimizations can be expressed this way. The compiler may limit optimizations to those supported by the debugger, but this is often close to debugging an unoptimized version of the program. Therefore, Zurawski restricts debugging to discrete *inspection points* [100], between which the compiler can perform extensive optimizations without affecting debuggability. The concept is similar to our safepoints.

Hölzle et al. present a conversion of optimized code into unoptimized code and call it *dynamic deoptimization* [49]. Dynamic deoptimization enables the system to debug individual methods at the source code level while executing others at full speed, as well as to change a running program and immediately observe the effects of the change. The authors also introduce *lazy deoptimization*, which defers deoptimization until control returns to the method to be deoptimized.

Chambers and Ungar describe how their SELF compiler propagates type binding information through the control flow graph in order to replace dynamically bound messages with statically bound procedure calls [20]. Dean et al. gain information about the possible classes of a receiver via a static class hierarchy analysis [27]. A dependency framework [19] is used to identify those parts of a program that must be recompiled when the class hierarchy or the set of methods change.

The Java HotSpot VM combines several of the above concepts [33]. The JIT compiler inlines or statically binds virtual methods based on class hierarchy analysis and records method dependencies. It generates debugging information and stores it with the machine code. A compiled and currently executing method is lazily deoptimized if the user sets a breakpoint in the method or if dynamic class loading invalidates the method's machine code.

Deoptimization can also be used to support *partial redundancy elimination*, which eliminates redundancies by moving instructions out of a loop or into another basic block [55]. The removal of instructions that may throw an exception is usually restricted because their order must be preserved. Odaira and Hiraki overcome this restriction by deoptimizing the code when an exception actually occurs at a reordered instruction [70].

Ishizaki et al. use code patching instead of completely deoptimizing or recompiling a method [52]. The compiler analyzes the current class hierarchy to inline or directly jump to a virtual method, but also generates *backup code* which performs the original dynamic call. When the assumption about the class hierarchy becomes invalid, the machine code is patched to henceforth execute the backup code. This approach involves less run-time overhead than deoptimization, but it is also less flexible and e.g. not suitable for undoing optimizations like scalar replacement or stack allocation.

9 Summary

The previous chapters described our algorithm for escape analysis. They dealt with the computation of escape states and their use for various optimizations, with run-time support, benchmark results and related work. The following sections list the main contributions of our work, summarize the core parts of the algorithm, and give an outlook on planned enhancements.

9.1 Contributions

Today's applications are large and target a variety of platforms, so portability and robustness gain in importance. This needs not come at cost of speed and efficiency. A lot of research effort has been spent on just-in-time compilation and optimizations such as escape analysis. This work contributes the following:

- It presents a new intraprocedural and interprocedural approach to escape analysis for Java.
- It describes an algorithm especially tailored to the needs of a dynamic compiler which lacks a view of the complete program.
- It introduces equi-escape sets for the representation of dependencies between objects and the efficient propagation of escape states.
- It estimates the escape states for parameters of methods that have not been compiled yet.
- It performs scalar replacement of fields, stack allocation of objects and fixed-sized arrays, as well as synchronization removal.
- It uses interprocedural escape information to support the compiler in inlining decisions and to allocate actual parameters on the stack.
- It implements and evaluates the analysis and related optimizations in the just-in-time compiler of a production system.
- It adapts garbage collection and write barriers for card marking to deal with stack objects.
- It extends the deoptimization framework to reallocate and relock objects when execution needs to be continued in the interpreter.

This project is a result of the collaboration between Sun Microsystems and the Institute for System Software at the Johannes Kepler University Linz. Therefore, design decisions were influenced by the architecture of Sun's Java HotSpot VM and its client compiler.

9.2 The Big Picture

This chapter recapitulates the core parts of our escape analysis. It gives a coarse overview and describes the individual steps sorted by their occurrence during the compilation process. We consider the compilation of the following method, which allocates and prints a person with a name and a date of birth specified in milliseconds since January 1, 1970:

```
static void printPerson(String name, long date) {
  Person p = new Person(name, date);
  p.print();
}
```

The constructor for `Person` stores the parameters in instance fields after encapsulating the birthday in a `Date` object. By convention, the current time is used as the birthday if the specified `date` parameter is 0:

```
public Person(String name, long date) {
  this.name = name;
  if (date == 0) {
    this.bday = new Date();
  } else {
    this.bday = new Date(date);
  }
}
```

The `print` method of the `Person` class is synchronized. It displays the person's name on the standard output and delegates printing the birthday to a date formatter, which is retrieved via the static method `getDateFormatter`:

```
public synchronized void print() {
  System.out.println(name);
  DateFormatter f = getDateFormatter();
  f.format(bday);
}
```

9.2.1 Intraprocedural Analysis

We start our observations at the time when the VM initiates the compilation of `printPerson`. The client compiler parses the method and translates the bytecodes into the HIR. The constructor of `Person` and the `print` method are inlined. Figure 9.1 shows the resulting control flow graph together with the locals array and the field map.

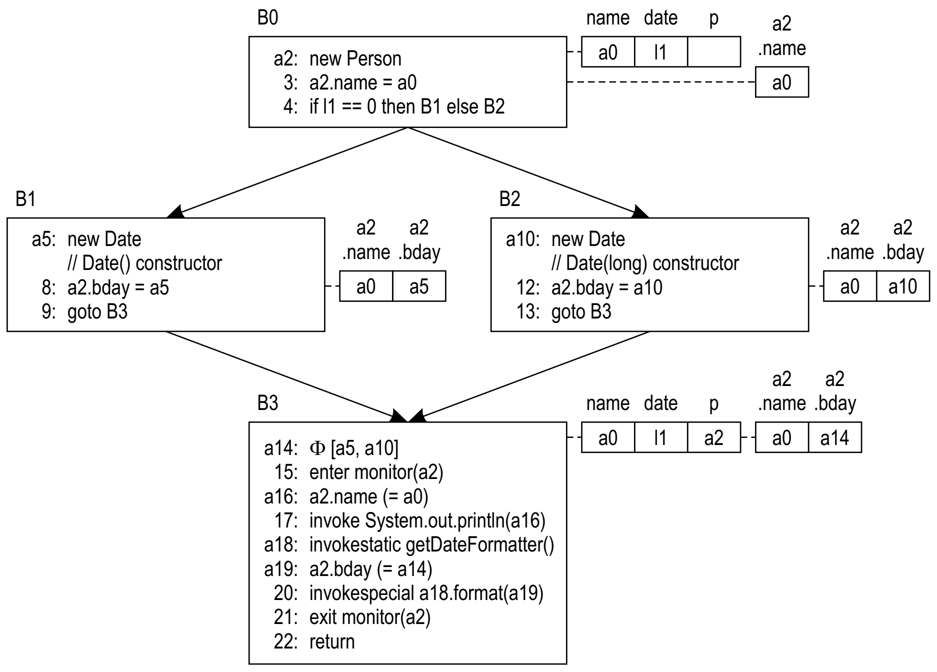


Figure 9.1: HIR of the sample method

At the beginning of the method, the locals array only contains the values `a0` and `l1`, representing the formal parameters `name` and `date`. The first HIR instruction allocates memory for the `Person` object. Assume that the class `Person` is loaded and does not define a finalizer. Therefore, the object starts as non-escaping. When the inlined constructor assigns the string parameter `a0` to the field `a2.name`, our algorithm records the value in a new slot of the field map.

Depending on the comparison of the second parameter with 0, either block `B1` or block `B2` is executed. Both branches create a new `Date` object and assign it to the `bday` field. Assume that the constructors are inlined, so that the objects remain non-escaping. A reference to `a5` respectively `a10` is stored in the field map.

At the subsequent join point, a phi function `a14` for the field `bday` is created. Its operands are `a5` and `a10`. The field map is adjusted to map `a2.bday` to the phi function at the beginning of basic block B3. No phi function is created for the field name, because it contains a0 regardless of control flow.

If the phi function or one of its operands turn out as escaping in the course of parsing, none of the objects can be eliminated any longer. Therefore, the phi function and both operands are inserted into an equi-escape set (see Figure 9.2). The phi function is selected as the root of the instruction tree and thus acts as the representative. In the future, its escape state is used for all elements.

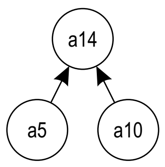


Figure 9.2: Equi-escape set of date objects

After execution of the constructor, the newly created Person object is assigned to the local variable p. The compiler does not generate an HIR instruction for the assignment, but stores a reference to a2 in the locals array.

9.2.2 Inlining of Synchronized Methods

Although print is a virtual method, it can be inlined. The compiler knows that the type of the receiver is Person, because the method is invoked on an object allocated within the current method.

During inlining, synchronization at method-level is converted into a synchronized block. The enter monitor instruction locks the object a2 before the inlined code is executed, and exit monitor unlocks it at the end.

Within the method, the fields name and bday are loaded. We look up their current values in the field map and associate them with the load instructions. No replacement is performed at this point because a2 may still escape below.

9.2.3 Interprocedural Analysis

Assume that the method format is too large to be inlined, but interprocedural escape information tells the compiler that the parameter a19 does not escape from format. The date object cannot be eliminated, but it may be allocated in the stack frame of the printPerson method.

The current value of a19 is a14. Escape analysis adjusts the escape state of a14 accordingly. Since a5 and a10 are elements of the same equi-escape set, their escape state implicitly changes to stack-allocatable as well.

Class hierarchy analysis determines that currently no subclass of DateFormatter is loaded. Although format cannot be inlined, it is bound statically to eliminate the dispatching overhead. This is indicated via the invokespecial instruction.

9.2.4 Optimizations

Our algorithm for escape analysis is performed in parallel to the construction of the intermediate representation. This saves the compiler from an extra pass over the control flow graph. At the end, when we know which objects escape and which do not, we iterate over the instructions for the purpose of scalar replacement and synchronization removal.

The allocation of the non-escaping Person object and assignments to its fields are eliminated. When a field is loaded and used as an operand in another instruction, the operand is substituted with the current value of the field that is stored in the load instruction.

Instructions that lock or unlock a non-escaping or stack-allocatable object are eliminated because the object will never be accessed by more than one thread. The optimized intermediate representation for our sample method is shown in Figure 9.3. The back end generates smaller and faster executing machine code.

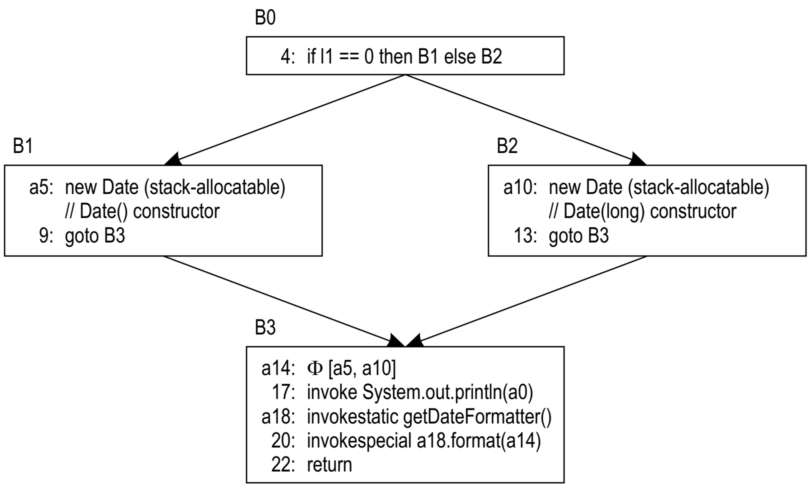

B0
4: if l1 == 0 then B1 else B2
B1
a5: new Date (stack-allocatable)
// Date() constructor
9: goto B3
B2
a10: new Date (stack-allocatable)
// Date(long) constructor
13: goto B3
B3
a14: Φ [a5, a10]
17: invoke System.out.println(a0)
a18: invokestatic getDateFormatter()
20: invokespecial a18.format(a14)
22: return

Figure 9.3: Optimized intermediate representation

9.2.5 Debugging Information

The static binding of the `format` method improves performance, but can cause problems at run time. If `getDateFormatter` dynamically loads a subclass of `DateFormatter` and returns an instance of it, the generated machine code calls the wrong method.

In this case, the machine code is invalidated before the `format` method is called. Execution of `printPerson` is suspended and continued in the interpreter. Since the interpreter does not know about scalar replacement, stack allocation and synchronization removal, the optimizations must be undone. This is called *deoptimization*.

The compiler uses the locals array and the field map to consider optimized objects in the debugging information. It describes type and field values of eliminated objects and records eliminated locks. The deoptimization framework uses the debugging information to reallocate eliminated objects and relock objects if synchronization on them was removed.

9.3 Future Work

We have implemented a reliable, sometimes conservative algorithm for escape analysis. Future work will focus on more aggressive optimizations. Based on the results and experience gained from our implementation, Sun Microsystems plans to add escape analysis also to the Java HotSpot server compiler. Besides, we are going to investigate and implement new optimizations for the client compiler.

9.3.1 More Aggressive Optimizations

According to the Java language specification, fields of an object are always initialized with default values. This is an expensive part of object allocation. We think about eliminating the initialization of fields that are explicitly assigned a value by the constructor. To prevent other threads from seeing uninitialized fields, the object must not escape before all fields have been assigned.

Elimination of write barriers for the snapshot-at-the-beginning style of concurrent garbage collection requires an escape analysis. The compiler must prove that a field was null before it is assigned a pointer, which implies that no other thread has modified the field concurrently. Such an analysis has already been implemented in a version of the client compiler without an SSA-based intermediate representation [69]. We are going to investigate how our algorithm can be modified to produce similar results.

Finally, we are concerned about synchronized methods that are too large to be inlined. Without inlining, synchronization cannot be removed because the methods may be invoked on shared objects. However, it is possible to generate unsynchronized versions of hot methods and invoke them if the compiler guarantees that the receiver does not escape at a certain call site. Methods have to be selected carefully to avoid wasting memory with rarely called duplicates.

9.3.2 Escape Analysis in the Server Compiler

Sun Microsystems and Sun Microsystems Laboratories work on an escape analysis for the Java HotSpot server compiler (see Section 1.2.3). They are going to implement both a flow-sensitive and a flow-insensitive analysis. Although the short-term goal is only synchronization removal, current thoughts also consider scalar replacement and stack allocation.

The implementation of a flow-sensitive escape analysis is complicated by the fact that the server compiler represents a Java method as a sea of instruction nodes. There is no explicit control flow graph, but only a data-dependency graph with control flow constraints. SSA form is used even for modifications of the memory. The field assignment

```
T p;
...
p.f = x;
```

is modeled by nodes representing the state of the *memory slice* of all fields `f` in objects of type `T`. The above assignment updates the current state of the field slice. It converts the input state of `T.f` to a new state in which `T.f[p]` is `x`. The output value flows to downstream uses, as usual in a data-dependency graph.

This approach works fine for analysis attributes that are modeled as values of these nodes, but some attributes are global attributes of the analysis state. The set of escaped nodes, for example, is a property of the whole program state at an execution point rather than of some node.

For this reason, the details of the analysis will differ from our implementation in the client compiler. However, the server compiler will probably adopt our approach to stack allocation, the enhancements of the debugging information and the deoptimization framework, the conservative bytecode analysis of uncompiled methods, and our treatment of write barriers for card marking.

9.3.3 Automatic Object Inlining

In C++, an object may not only point to, but actually contain other objects. The programmer can choose between the two alternatives at the point of field declaration. In Java, fields always store references to other objects, not the objects themselves.

Automatic object inlining is an optimization that shares some ideas and goals with escape analysis. The compiler identifies sets of objects that can efficiently be fused into larger objects. New classes are created and used to allocate compound objects. This reduces the number of object allocations and pointer dereferences [61].

Figure 9.4 illustrates how `Point` objects can be inlined into a `Rectangle` object. It contrasts two possible memory layouts: When objects are inlined with their headers, their structure and pointers to them are preserved. Alternatively, only the fields of objects may be inlined. This saves memory space, but requires a modified field access also in interpreted code.

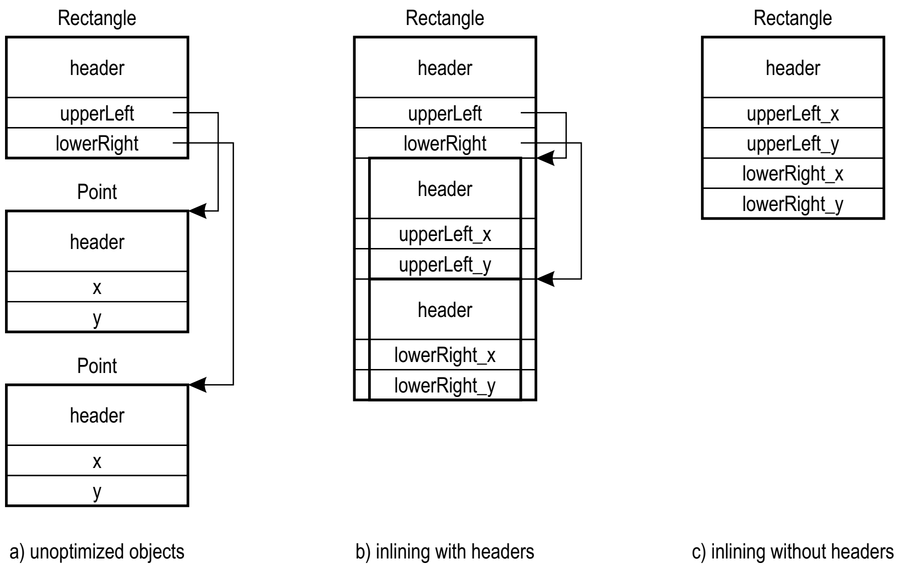


Figure 9.4: Example for object inlining

Several algorithms for automatic object inlining were implemented in static compilers [34, 60]. In the context of our collaboration with Sun Microsystems, we will investigate object inlining for a dynamic compiler. We are going to implement a prototype for the Java HotSpot client compiler and adjust the garbage collector and the deoptimization framework accordingly [96].

9.4 Conclusions

Since its introduction, the Java programming language has attracted the attention of a broad audience of programmers, software engineers and web masters. It provides built-in support for exception handling, multi-threading and synchronization, and is often viewed as the modern alternative to C++.

Java has been considered as execution-inefficient for a long time, despite the fact that inefficiency is a property of language implementations and not of the language itself. A great deal of the last years' research effort on Java tried to reduce the gap of speed between Java applications and C programs [88].

Execution speed improved significantly with the development of just-in-time compilers and advanced garbage collectors. Although garbage collection has not reached the efficiency of explicit memory management yet [44], modern approaches cause only minimal GC pauses.

The escape analysis described in this thesis both leads to faster machine code and reduces the burden of the garbage collector. As more and more compiler optimizations increase performance, software developers and users will decide for Java applications not only because of their safety and portability, but also because of their execution speed.

Bibliography

[1] Adl-Tabatabai, A.-R., M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. "Fast, Effective Code Generation in a Just-In-Time Java Compiler." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 280–290. Montreal, June 1998. doi:10.1145/277650.277740.

[2] Agesen, O. "GC Points in a Threaded Environment." Technical Report TR-98-70, Sun Microsystems Laboratories, December 1998.

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

[4] Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[5] Aldrich, J., E. G. Sirer, C. Chambers, and S. J. Eggers. "Comprehensive Synchronization Elimination for Java." *Science of Computer Programming*, 47(2–3):91–120, May 2003. doi:10.1016/S0167-6423(02)00129-6.

[6] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[7] Arnold, M. *Online Profiling and Feedback-Directed Optimization of Java*. PhD thesis, Rutgers, The State University of New Jersey, New Brunswick, October 2002.

[8] Aycock, J. "A Brief History of Just-In-Time." *ACM Computing Surveys*, 35(2):97–113, June 2003. doi:10.1145/857076.857077.

[9] Bacon, D. F., R. Konuru, C. Murthy, and M. Serrano. "Thin Locks: Featherweight Synchronization for Java." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 258–268. Montreal, June 1998. doi:10.1145/277650.277734.

[10] Beers, M. Q., C. H. Stork, and M. Franz. "Efficiently Verifiable Escape Analysis." In *Proceedings of the European Conference on Object-Oriented Programming*, 75–95. Oslo, June 2004.

[11] Bilardi, G., and K. Pingali. "Algorithms for Computing the Static Single Assignment Form." *Journal of the ACM*, 50(3):375–425, May 2003. doi: 10.1145/765568.765573.

[12] Blanchet, B. "Escape Analysis for Java^{TM}: Theory and Practice." *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003. doi:10.1145/945885.945886.

[13] Bogda, J., and U. Hölzle. "Removing Unnecessary Synchronization in Java." In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 35–46. Denver, November 1999. doi:10.1145/320384.320388.

[14] Bogda, J., and A. Singh. "Can a Shape Analysis Work at Run-time?" In *Proceedings of the Java Virtual Machine Research and Technology Symposium*. Monterey, April 2001.

[15] Bracha, G. *JSR 14: Add Generic Types to the Java^{TM} Programming Language*, September 2004. http://jcp.org/en/jsr/detail?id=14.

[16] Brender, R. F. "An Annotated Bibliography on Debugging Optimized Code." *Digital Technical Journal*, 10(1), 1998.

[17] Briggs, P., K. D. Cooper, and L. Torczon. "Improvements to Graph Coloring Register Allocation." *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994. doi:10.1145/177492.177575.

[18] Bruns, G., and S. Chandra. "Searching for Points-To Analysis." In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, 61–70. Charleston, November 2002. doi:10.1145/587051.587061.

[19] Chambers, C., J. Dean, and D. Grove. "A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies." In *Proceedings of the International Conference on Software Engineering*, 221–230. Seattle, April 1995. doi:10.1145/225014.225035.

[20] Chambers, C., and D. Ungar. "Making Pure Object-Oriented Languages Practical." In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1–15. Phoenix, October 1991. doi:10.1145/117954.117955.

[21] Chambers, C., D. Ungar, and E. Lee. "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes." In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 49–70. New Orleans, October 1989. doi:10.1145/74877.74884.

[22] Choi, J.-D., M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. "Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis." *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, November 2003. doi:10.1145/945885.945892.

[23] Click, C., and M. Paleczny. "A Simple Graph-Based Intermediate Representation." In *Papers from the ACM SIGPLAN Workshop on Intermediate Representations*, 35–49. San Francisco, January 1995. doi:10.1145/202529.202534.

[24] Coutant, D. S., S. Meloy, and M. Ruscetta. "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 125–134. Atlanta, June 1988. doi:10.1145/53990.54003.

[25] Cramer, T., R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. "Compiling Java Just in Time." *IEEE Micro*, 17(3):36–43, May 1997. doi:10.1109/40.591653.

[26] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. doi:10.1145/115372.115320.

[27] Dean, J., D. Grove, and C. Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis." In *Proceedings of the European Conference on Object-Oriented Programming*, 77–101. Århus, August 1995.

[28] Detlefs, D., and O. Agesen. "Inlining of Virtual Methods." In *Proceedings of the European Conference on Object-Oriented Programming*, 258–278. Lisbon, June 1999.

[29] Detlefs, D., C. Flood, S. Heller, and T. Printezis. "Garbage-First Garbage Collection." In *Proceedings of the International Symposium on Memory Management*, 37–48. Vancouver, October 2004. doi:10.1145/1029873.1029879.

[30] Deutsch, A. "On the Complexity of Escape Analysis." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 358–371. Paris, January 1997. doi:10.1145/263699.263750.

[31] Deutsch, L. P., and A. M. Schiffman. "Efficient Implementation of the Smalltalk-80 System." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 297–302. Salt Lake City, January 1984.

[32] Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. "On-the-Fly Garbage Collection: An Exercise in Cooperation." *Communications of the ACM*, 21(11):966–975, November 1978. doi:10.1145/359642.359655.

[33] Dmitriev, M. "Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications." In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution.* Tampa Bay, October 2001.

[34] Dolby, J., and A. A. Chien. "An Automatic Object Inlining Optimization and its Evaluation." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 345–357. Vancouver, June 2000. doi:10.1145/349299.349344.

[35] Fink, S. J., and F. Qian. "Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement." In *Proceedings of the International Symposium on Code Generation and Optimization*, 241–252. San Francisco, March 2003. doi:10.1109/CGO.2003.1191549.

[36] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[37] Garbervetsky, D., C. Nakhli, S. Yovine, and H. Zorgati. "Program Instrumentation and Run-Time Analysis of Scoped Memory in Java." In *Proceedings of the International Workshop on Runtime Verification*, 105–121. Barcelona, April 2004. doi:10.1016/j.entcs.2004.01.031.

[38] Gay, D., and B. Steensgaard. "Fast Escape Analysis and Stack Allocation for Object-Based Programs." In *Proceedings of the International Conference on Compiler Construction*, 82–93. Berlin, March 2000.

[39] Ghiya, R., D. Lavery, and D. Sehr. "On the Importance of Points-To Analysis and Other Memory Disambiguation Methods for C Programs." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 47–58. Snowbird, June 2001. doi:10.1145/378795.378806.

[40] Gosling, J., B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification.* 2nd edition. Addison-Wesley, 2000.

[41] Grarup, S., and J. Seligmann. *Incremental Mature Garbage Collection.* Master's thesis, Department of Computer Science, Aarhus University, Århus, August 1993.

[42] Griesemer, R., and S. Mitrovic. "A Compiler for the Java HotSpot$^{\text{TM}}$ Virtual Machine." In *The School of Niklaus Wirth: The Art of Simplicity*, edited by L. Böszörményi, J. Gutknecht, and G. Pomberger, 133–152. Heidelberg: dpunkt.verlag, October 2000.

[43] Hennessy, J. "Symbolic Debugging of Optimized Code." *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982. doi: 10.1145/357172.357173.

[44] Hertz, M., and E. D. Berger. "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management." In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications.* San Diego, October 2005.

[45] Hind, M. "Pointer Analysis: Haven't We Solved This Problem Yet?" In *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 54–61. Snowbird, June 2001. doi:10.1145/379605.379665.

[46] Hirzel, M., A. Diwan, and M. Hind. "Pointer Analysis in the Presence of Dynamic Class Loading." In *Proceedings of the European Conference on Object-Oriented Programming*, 96–122. Oslo, June 2004.

[47] Hölzle, U. "A Fast Write Barrier for Generational Garbage Collectors." In *Proceedings of the ACM OOPSLA Workshop on Garbage Collection and Memory Management.* Washington, D.C., October 1993.

[48] Hölzle, U., C. Chambers, and D. Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches." In *Proceedings of the European Conference on Object-Oriented Programming*, 21–38. Geneva, July 1991.

[49] Hölzle, U., C. Chambers, and D. Ungar. "Debugging Optimized Code with Dynamic Deoptimization." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 32–43. San Francisco, June 1992. doi:10.1145/143095.143114.

[50] Hosking, A. L., and R. L. Hudson. "Remembered Sets Can Also Play Cards." In *Proceedings of the ACM OOPSLA Workshop on Garbage Collection and Memory Management.* Washington, D.C., October 1993.

[51] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 2A & 2B: Instruction Set Reference*, 2004. Order numbers 253666 and 253667.

[52] Ishizaki, K., M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. "A Study of Devirtualization Techniques for a Java$^{\text{TM}}$ Just-In-Time Compiler." In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 294–310. Minneapolis, October 2000. doi:10.1145/353171.353191.

[53] Ishizaki, K., M. Kawahito, T. Yasue, et al. "Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler." In *Proceedings of the ACM Conference on Java Grande*, 119–128. San Francisco, June 1999. doi:10.1145/304065.304111.

[54] Jones, R., and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Chichester: John Wiley & Sons, 1996.

[55] Knoop, J., O. Rüthing, and B. Steffen. "Optimal Code Motion: Theory and Practice." *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994. doi:10.1145/183432.183443.

[56] Kotzmann, T. *Ein Just-in-Time-Compiler für Java.* Master's thesis, Institute for Practical Computer Science, Johannes Kepler University Linz, August 2002.

[57] Kotzmann, T., and H. Mössenböck. "Escape Analysis in the Context of Dynamic Compilation and Deoptimization." In *Proceedings of the International Conference on Virtual Execution Environments*, 111–120. Chicago, June 2005. doi:10.1145/1064979.1064996.

[58] Krall, A. "Efficient JavaVM Just-in-Time Compilation." In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 205–212. Paris, October 1998. doi:10.1109/PACT.1998.727250.

[59] Krall, A., and M. Probst. "Monitors and Exceptions: How to implement Java efficiently." In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, 15–24. Palo Alto, February 1998.

[60] Laud, P. "Analysis for Object Inlining in Java." In *Proceedings of the Workshop on Java Optimization Strategies for Embedded Systems.* Genova, April 2001.

[61] Lhoták, O., and L. Hendren. "Run-time Evaluation of Opportunities for Object Inlining in Java." In *Proceedings of the Joint ACM-ISCOPE Conference on Java Grande*, 175–184. Seattle, November 2002. doi:10.1145/583810.583830.

[62] Lindholm, T., and F. Yellin. *The Java^{TM} Virtual Machine Specification.* 2nd edition. Addison-Wesley, 1999.

[63] Manson, J., W. Pugh, and S. V. Adve. "The Java Memory Model." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 378–391. Long Beach, January 2005. doi:10.1145/1040305.1040336.

[64] McCarthy, J. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Communications of the ACM*, 3(4):184–195, April 1960. doi:10.1145/367177.367199.

[65] McManis, C. "Looking for lex and yacc for Java? You don't know Jack." *JavaWorld*, December 1996. http://www.javaworld.com/javaworld/jw-12-1996/jw-12-jack.html.

[66] Mössenböck, H. "Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java Hotspot$^{\text{TM}}$ Client Compiler." Technical Report 15, Institute for Practical Computer Science, Johannes Kepler University Linz, November 2000.

[67] Mössenböck, H., and M. Pfeiffer. "Linear Scan Register Allocation in the Context of SSA Form and Register Constraints." In *Proceedings of the International Conference on Compiler Construction*, 229–246. Grenoble, April 2002.

[68] Muchnick, S. S. *Advanced Compiler Design and Implementation.* San Francisco: Morgan Kaufmann, 1997.

[69] Nandivada, V. K., and D. Detlefs. "Compile-Time Concurrent Marking Write Barrier Removal." In *Proceedings of the International Symposium on Code Generation and Optimization*, 37–48. San Jose, March 2005. doi:10.1109/CGO.2005.12.

[70] Odaira, R., and K. Hiraki. "Sentinel PRE: Hoisting beyond Exception Dependency with Dynamic Deoptimization." In *Proceedings of the International Symposium on Code Generation and Optimization*, 328–338. San Jose, March 2005. doi:10.1109/CGO.2005.32.

[71] Paleczny, M., C. Vick, and C. Click. "The Java HotSpot$^{\text{TM}}$ Server Compiler." In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 1–12. Monterey, April 2001.

[72] Park, Y. G., and B. Goldberg. "Escape Analysis on Lists." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 116–127. San Francisco, June 1992. doi:10.1145/143095.143125.

[73] Pelegrí-Llopart, E., and S. L. Graham. "Optimal Code Generation for Expression Trees: An Application of BURS Theory." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 294–308. San Diego, January 1988. doi:10.1145/73560.73586.

[74] Poletto, M., and V. Sarkar. "Linear Scan Register Allocation." *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999. doi:10.1145/330249.330250.

[75] Pozo, R., and B. Miller. *Java SciMark 2.0.* http://math.nist.gov/scimark2/.

[76] Printezis, T., and D. Detlefs. "A Generational Mostly-concurrent Garbage Collector." Technical Report TR-2000-88, Sun Microsystems Laboratories, June 2000.

[77] Pugh, W. "Fixing the Java Memory Model." In *Proceedings of the ACM Conference on Java Grande*, 89–98. San Francisco, June 1999. doi:10.1145/304065.304106.

[78] Ruf, E. "Effective Synchronization Removal for Java." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 208–218. Vancouver, June 2000. doi:10.1145/349299.349327.

[79] Ruggieri, C., and T. P. Murtagh. "Lifetime Analysis of Dynamically Allocated Objects." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 285–293. San Diego, January 1988. doi:10.1145/73560.73585.

[80] Sălcianu, A., and M. Rinard. "Pointer and Escape Analysis for Multithreaded Programs." In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 12–23. Snowbird, June 2001. doi:10.1145/379539.379553.

[81] Sedgewick, R. *Algorithms.* 2nd edition. Addison-Wesley, 1988.

[82] Standard Performance Evaluation Corporation. *The SPEC JVM98 Benchmarks.* http://www.spec.org/jvm98/.

[83] Steensgaard, B. "Points-to Analysis in Almost Linear Time." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 32–41. St. Petersburg Beach, January 1996. doi:10.1145/237721.237727.

[84] Sun Microsystems, Inc. *The Java HotSpot Virtual Machine, v1.4.1*, September 2002. http://java.sun.com/products/hotspot/.

[85] Sun Microsystems, Inc. *Java^{TM} 2 Platform Standard Edition 5.0 API Specification*, 2004. http://java.sun.com/j2se/1.5.0/docs/api/.

[86] Tofte, M., and J.-P. Talpin. "Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions." In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 188–201. Portland, January 1994.

[87] Traub, O., G. Holloway, and M. D. Smith. "Quality and Speed in Linear-scan Register Allocation." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 142–151. Montreal, June 1998. doi:10.1145/277650.277714.

[88] Veldema, R. S., T. Kielmann, and H. E. Bal. "Optimizing Java-specific Overheads: Java at the Speed of C?" In *Proceedings of the International Conference on High Performance Computing and Networking*, 685–692. Amsterdam, June 2001.

[89] Vivien, F., and M. Rinard. "Incrementalized Pointer and Escape Analysis." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 35–46. Snowbird, June 2001. doi:10.1145/378795.378804.

[90] Wall, D., A. Srivastava, and F. Templin. "A Note on Hennessy's 'Symbolic Debugging of Optimized Code'." *ACM Transactions on Programming Languages and Systems*, 7(1):176–181, January 1985. doi:10.1145/2363.215005.

[91] Weaver, D. L., and T. Germond, eds. *The SPARC Architecture Manual, Version 9*. New Jersey: Prentice Hall, 1994.

[92] Whaley, J. "Partial Method Compilation using Dynamic Profile Information." In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 166–179. Tampa Bay, October 2001. doi:10.1145/504282.504295.

[93] Whaley, J., and M. Rinard. "Compositional Pointer and Escape Analysis for Java Programs." In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 187–206. Denver, November 1999. doi:10.1145/320384.320400.

[94] Wilson, P. R. "Uniprocessor Garbage Collection Techniques." In *Proceedings of the International Workshop on Memory Management*, 1–42. St. Malo, September 1992.

[95] Wimmer, C. *Linear Scan Register Allocation for the Java HotSpot^{TM} Client Compiler*. Master's thesis, Institute for System Software, Johannes Kepler University Linz, August 2004.

[96] Wimmer, C. “Automatic Object Inlining.” Project proposal, Institute for System Software, Johannes Kepler University Linz, January 2005.

[97] Wimmer, C., and H. Mössenböck. “Optimized Interval Splitting in a Linear Scan Register Allocator.” In *Proceedings of the International Conference on Virtual Execution Environments*, 132–141. Chicago, June 2005. doi: 10.1145/1064979.1064998.

[98] Zee, K., and M. Rinard. “Write Barrier Removal by Static Analysis.” In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 191–210. Seattle, November 2002. doi:10.1145/582419.582439.

[99] Zellweger, P. T. *Interactive Source-Level Debugging of Optimized Programs.* PhD thesis, University of California, Berkeley, May 1984.

[100] Zurawski, L. W. *Interactive Source-Level Debugging of Globally Optimized Code with Expected Behavior.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.

Index