JOHANNES KEPLER
UNIVERSITY LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

Partial Escape Analysis and Scalar Replacement for Java

DISSERTATION

submitted in partial fulfillment of the requirements
for the academic degree

Doktor der technischen Wissenschaften

in the Doctoral Program in Engineering Sciences

Submitted by

Dipl.-Ing. Lukas Stadler

At the

Institute for System Software

Accepted on the recommendation of

Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

Univ.-Prof. Dipl.-Ing. Dr. Walter Binder

Linz, May 2014

Abstract

Escape Analysis allows a compiler to determine whether an object is accessible outside the allocating method or thread. This information is used to perform optimizations such as Scalar Replacement, Stack Allocation and Lock Elision, allowing modern dynamic compilers to remove some of the abstractions introduced by advanced programming models.

The all-or-nothing approach taken by most Escape Analysis algorithms prevents all these optimizations as soon as there is one branch where the object escapes, no matter how unlikely this branch is at runtime.

This thesis presents a new, practical algorithm that performs control flow sensitive Partial Escape Analysis in a dynamic Java compiler. It allows Escape Analysis and Scalar Replacement to be applied on individual branches. We implemented the algorithm on top of an open-source Java just-in-time compiler, and it performs well on a diverse set of benchmarks.

In this thesis, we evaluate the effect of Partial Escape Analysis on the DaCapo, ScalaDaCapo and SPECjbb2005 benchmarks, in terms of run-time, number and size of allocations and number of locking operations. It performs particularly well in situations with additional levels of abstraction, such as code generated by the Scala compiler. It reduces the amount of allocated memory by up to 58.5%, and improves performance by up to 33%.

Kurzfassung

Escape Analysis ermöglicht es Compilern festzustellen, ob auf ein Objekt von außerhalb der allokierenden Methode oder des allokierenden Threads zugegriffen werden kann. Basierend darauf können Optimierungen wie Scalar Replacement, Stack Allocation und Lock Elision durchgeführt werden, die es modernen Compilern erlauben, die Abstraktionen moderner Programmiertechniken zu entfernen.

Der Alles-oder-Nichts - Ansatz der meisten Escape Analysis Algorithmen verhindert alle diese Optimierungen, sobald das Objekt auch nur in einem Pfad entkommt, egal wie unwahrscheinlich dieser Pfad zur Laufzeit ist.

Diese Arbeit stellt einen neuen, praktikablen Algorithmus vor, der kontrollflusssensitive Partial Escape Analysis in dynamischen Java Compilern durchführt. Dieser erlaubt es, Escape Analysis und Scalar Replacement auf einzelne Pfade anzuwenden. Die Implementierung dieses Algorithmus ist Teil eines Open-Source Java Just-in-Time Compilers, wobei dieses System eine Vielzahl unterschiedlicher Benchmarks effizient ausführt.

Diese Arbeit evaluiert den Effekt von Partial Escape Analysis basierend auf den DaCapo, ScalaDaCapo und SPECjbb2005 Benchmarks in Hinblick auf Laufzeit, Anzahl und Größe der Allokationen und Anzahl der Lock-Operationen. Partial Escape Analysis arbeitet besonders gut in Situationen mit zusätzlichen Abstraktionsebenen, wie sie z.B. der Scala Compiler erzeugt. Die Größe des allokierten Speichers wird um bis zu 58,5% verringert, die Laufzeit um bis zu 33% verbessert.

Contents

Acknowledgments

Many people have contributed directly or indirectly to the work presented in this thesis.

First and foremost, I thank my advisor Prof. Hanspeter Mössenböck for his valuable feedback, for his encouragement and for bringing me into the “SSW family”. I also thank my second advisor, Prof. Walter Binder from the Università della Svizzera Italiana in Lugano, for the time and effort spent examining my thesis.

I thank all the current and previous members of our project, especially Thomas Würthinger, Gilles Duboscq and Doug Simon, for letting me take part in this adventure called Graal. Seeing it grow from a small prototype to the great and diverse project it is today made me realize that even decisions that seem small at the time of taking them can have a big impact - never accept compromises that you don’t believe in.

Various projects, especially Truffle, use the algorithms and the implementation presented in this thesis. I thank all my colleagues in the VM Research Group of Oracle Labs and at the Institute for System Software for the discussions and the feedback that helped weeding out problems and bugs.

This work was performed in collaboration with Oracle Labs. The continuous support provided by Oracle, and formerly Sun Microsystems, is what allowed us to bridge the gap between academia and industry. I thank all the people inside Oracle that made this happen, especially Mario Wolczko, Eric Sedlar and Laura Hill. I also thank my Alma Mater Kepleriana for providing the framework for successful and mutually beneficial cooperations.

I thank my parents and my whole family for supporting and encouraging me throughout my studies. Most of the time I spent on writing up this thesis I stole from my significant other. Thank you, Lilli, for your patience and for your love and understanding during my late-night and all-weekend writing sessions.

Chapter 1

Motivation

1.1 Problem Statement

Application servers such as WebLogic, desktop applications such as NetBeans and Eclipse, libraries such as Hibernate and tools such as Maven have all reached sizes that are counted in millions of lines of code. Large applications such as these can only be developed and maintained by introducing interfaces and APIs, by generalizing and abstracting, by using object-oriented design and by applying design patterns such as iterators, factories, adapters and proxies.

All these techniques are intended to make it easier to do the right thing, and harder to do the wrong thing. However, they often introduce abstractions that manifest themselves in additional object allocations. For example, while more low-level systems will return a `null` value in case no result is available, more sophisticated systems might allocate and return a wrapper object that can be queried for the availability of a result[1]. Another example are data structures such as `Point` which encapsulate multiple primitive values in a heap object. The application could just keep track of the `x`, `y` and `z` coordinates instead of creating a `Point` object, but the overhead is in most cases accepted in return for the additional encapsulation and maintainability.

All these additional objects are allocated many times during the run time of an application. They put stress on the garbage collection subsystem which would not be strictly necessary. Modern generational garbage collectors are very efficient, especially for garbage that dies young. However, the time spent on allocating and reclaiming objects is still a limiting factor for many applications. Depending on the configuration, current systems can sustain allocation rates in the order of 5-10 GB per second.

[1]Scala's `Option` is an example of such a wrapper: it can either be `None` or `Some`.

Even applications that do not exhaust the garbage collection system are still impacted by the cost of unnecessary allocations. Encapsulating information in heap objects also has the additional disadvantage that it hides information from the compiler: it is much harder for the compiler to reason about a value that is coming from a field than about a value that is accessed more directly, e.g., as a local variable. Furthermore, the contents of the heap object could potentially be modified and observed by other threads at any point in time.

Compilers that aim for competitive peak performance therefore need to remove as many of these abstractions as possible.

1.1.1 Existing Solutions

Escape Analysis (see Chapter 4) is the canonical answer to this problem. It checks whether an allocated object escapes, i.e., can be used outside, the allocating method or thread. This happens, for example, if it is assigned to a global variable or a heap object, or if it is passed as a parameter to some other method. Compilers use Escape Analysis to determine the dynamic scope and the lifetime of allocated objects. The result of this analysis allows the compiler to perform numerous optimizations on operations such as object allocations, synchronization primitives and field accesses.

For example, Escape Analysis allows the compiler to perform Stack Allocation, i.e., replace the allocation of an object on the heap with an allocation on the stack. The compiler can also replace the allocation of an object with local variables for all its fields, a process called Scalar Replacement. If Escape Analysis can prove that an object will never leave the current thread, lock operations can be omitted (Lock Elision).

1.1.2 Remaining Challenges

Escape Analysis is an all-or-nothing analysis - either an object escapes, or it does not. Even if an allocated object escapes only in a single unlikely branch, the allocation will not be optimized. This is amplified by the viral nature of the "escaping" property: all objects referenced by the barely-escaping object will also be considered to escape.

Another problem that stems from the huge amount of objects allocated in modern software is that many allocations of small objects will be scattered throughout the application code. The allocations might not even be visible to the programmer due to language features such as auto-boxing or for-each loops, both of which incur object allocations. It would be much more efficient to allocate several objects in one batch, but combining allocations is a hard problem in dynamic systems such as a modern Java Virtual Machine.

Also, the contents of an object will be initialized to the default values (`0`, `null`, `false`) during allocation. Afterwards, the constructor usually initializes most of the fields to some other value. In most cases, however, only the final value is visible from outside the current scope, so that initializing fields to default values is unnecessary.

1.1.3 Novel Solution

This thesis suggest a flow-sensitive analysis and optimization algorithm which it calls *Partial Escape Analysis*. It aggressively performs optimizations such as Scalar Replacement and Lock Elision on all object allocations, and makes sure that objects exist in the unoptimized state in branches where they escape.

Both the analysis and the optimizations are performed during a reverse post order traversal of the compiler graph. The analysis might need to, but in practice rarely does, backtrack during loop processing. Additionally, our Partial Escape Analysis works on the compiler's intermediate representation, so that it can be applied (possibly multiple times) at any point during compilation.

Partial Escape Analysis performs the following optimizations in our system:

- It removes allocations of objects that are known to be non-escaping.
- If possible, it pushes allocations into infrequent branches, so that they will be executed less often at run time.
- All writes to an object up to the point where it escapes, and thus has to be allocated, are coalesced, so that the object can be initialized with the correct values during allocation.
- Partial Escape Analysis causes object allocations to concentrate on the escape points, so that usually multiple objects can be allocated at once.
- It defers writes to objects, even if the object will subsequently escape.
- Additionally, our implementation of Partial Escape Analysis performs read and write elimination during its iteration. This is not intrinsic to the Partial Escape Analysis algorithm, but read and write elimination fit well into the general structure of Partial Escape Analysis.

```java
public class ArrayMap {
  private static final int CAPACITY = 100;
  private final Object[] keys = new Object[CAPACITY];
  private final Object[] values = new Object[CAPACITY];
  private int size = 0;
  public int indexOf(Object key) {
    for (int i = 0; i < size; i++) {
      if (key.equals(keys[i])) {
        return i;
      }
    }
    return -1;
  }
  public void put(Object key, Object value) {
    int index = indexOf(key);
    if (index == -1) {
      keys[size] = key;
      values[size++] = value;
    } else {
      values[index] = value;
    }
  }

  public static void foo(ArrayMap map, int a, int b, Object value) {
    map.put(new Key(a, b), value);
  }
}
```

Listing 1.1: Example for motivating Partial Escape Analysis.

```java
public static void foo(ArrayMap map, int a, int b, Object value) {
  Key key = new Key(a, b);
  int index = -1;
  for (int i = 0; i < size; i++) {
    if ((keys[i] instanceof Key && key.a == ((Key) keys[i]).a &&
        key.b == ((Key) keys[i]).b)) {
      index = i;
      break;
    }
  }
  if (index == -1) {
    keys[size] = key;
    values[size++] = value;
  } else {
    values[index] = value;
  }
}
```

Listing 1.2: Example from Listing 1.1 after inlining.

1.1.4 Example

Listing 1.1 shows a simplified array-based map implementation in Java[2]. When compiling the code for the foo method, the calls to putIfAbsent, put, indexOf and equals will normally all be inlined. After inlining, the method might look like in Listing 1.2.

For traditional Escape Analysis, the newly allocated Key object clearly escapes, so no Stack Allocation or Scalar Replacement will be performed. Partial Escape Analysis recognizes that the object only escapes in the branch where the new entry is appended to the keys array. All accesses to the key object will be replaced with local variables, and the object will be created and initialized in the correct state right before the keys[size] = key; statement. The final result is shown in Listing 1.3.

```
public static void foo(ArrayMap map, int a, int b, Object value) {
  int key_a = a;
  int key_b = b;
  int index = -1;
  for (int i = 0; i < size; i++) {
    if ((keys[i] instanceof Key && key_a == ((Key) keys[i]).a &&
        key_b == ((Key) keys[i]).b)) {
      index = i;
      break;
    }
  }
  if (index == -1) {
    Key key = new Key(key_a, key_b);
    keys[size] = key;
    values[size++] = value;
  } else {
    values[index] = value;
  }
}
```

Listing 1.3: Example from Listing 1.2 after Partial Escape Analysis.

1.1.5 Applicability

The system presented in this thesis is implemented in the Graal compiler [17, 18, 41, 56, 57, 58], which is part of the OpenJDK project. It correctly and efficiently executes a wide range of benchmarks such as the DaCapo and Scala DaCapo benchmark suite, the SPECjvm2008 suite and the SPECjbb2005 and SPECjbb2013 benchmarks.

[2]Note that this example does not handle null keys.

Additionally, Graal's Partial Escape Analysis is used extensively as an integral part of the Truffle framework [76, 77]. It is also tested in this context, and benchmarks in languages such as JavaScript and Ruby are executed on a regular basis.

While this thesis describes the system in the context of dynamic Java compilers, the underlying algorithms are applicable to any compiler that deals with object allocations and accesses.

1.2 Scientific Contributions

This thesis contributes the following novel aspects, parts of which are published in [58]:

- A control-flow-sensitive Partial Escape Analysis algorithm that checks the escapability of objects for individual branches.
- An interface which allows compiler extensions to interact with Partial Escape Analysis.
- The integration of Partial Escape Analysis into a Java compiler based on SSA form and speculative optimizations as well as on a deoptimization mechanism.
- An evaluation of this algorithm on a set of current benchmarks, in terms of runtime, number and size of allocations and number of monitor operations, showing that the algorithm performs well on a variety of benchmarks.

Additionally, the author of this thesis (co-)designed and published numerous components of the Graal compiler. These aspects, some of which are not presented in detail in this thesis, include, but are not limited to, the following:

- The declarative intermediate representation, along with its implementation [17].
- The specific structure of the intermediate representation used in Graal, which is well suited for speculative optimizations [18].
- The caching of intermediate results within the compiler [56].
- Novel techniques for prioritizing compilation tasks in a compiler [56].
- The overall design of the compiler, along with the optimizations it includes [57].

1.3 Project Context

The work presented in this thesis was performed in the context of an ongoing collaboration between the Institute for System Software at the Johannes Kepler University Linz and Oracle Labs (formerly Sun Microsystems). This successful cooperation facilitated research on numerous aspects of Java Virtual Machines:

- Mössenböck [39] added a new SSA form intermediate representation to the HotSpot client compiler.
- Mössenböck and Pfeiffer [38] developed a linear scan register allocation algorithm.
- Wimmer and Mössenböck [64] expanded upon the previous work on linear scan register allocation and implemented it in the HotSpot client compiler.
- Kotzmann et al. [32, 34] added an Escape Analysis algorithm based on equi-escape sets to the HotSpot client compiler.
- Wimmer and Mössenböck explored automatic co-location of objects based on read barriers [65, 68], followed by automatic object and array inlining [66, 67].
- Würthinger et al. [70; 72] added an algorithm for array bounds check elimination to the HotSpot client compiler.
- Würthinger et al. [71] also worked on visualization of program dependence graphs.
- Würthinger et al. [78] implemented dynamic code evolution for the HotSpot VM which allows arbitrary changes to running applications.
- Würthinger et al. [73; 74; 75] extended the work on dynamic code evolution into the areas of aspect-oriented programming, graphical user interfaces and safe updates.
- Häubl et al. [24] worked on an improved representation of strings within the HotSpot VM.
- Schwaighofer [50] modified the HotSpot VM to allow for tail call optimizations.
- The author of this thesis worked on extending the HotSpot VM with continuations [54] and coroutines [55].
- Schatzl et al. [49] worked on various aspects of efficient garbage collection systems, including metadata management.

- Häubl et al. [23, 25, 26, 27] worked on numerous aspects of trace compilation, including heuristics and applications.

Most recent work in the context of the collaboration stems from the Graal project. This project started as an attempt to introduce a compiler written in Java into the HotSpot VM. After its inception in 2011, it quickly sparked interest both within the Oracle collaboration and with researchers from other universities. Numerous projects emerged from Graal, most notably the Truffle framework and the associated language implementations. Research and development on the Graal compiler and the Truffle framework is performed under the umbrella of the Graal OpenJDK project [41].

- The author of this thesis developed efficient methods for queuing of compilation tasks and caching of intermediate results within compilers [56].
- Würthinger et al. [77] worked on self-optimizing AST interpreters.
- Würthinger et al. [76] extended the work on AST interpreters by adding a mechanism to generate compiled code from the interpreter definition.
- The author of this thesis performed experimental studies on dynamic compiler optimizations [57].
- Grimmer et al. [22] introduced an efficient interface for native function calls.
- Duboscq et al. [17; 18] developed the design and implementation of Graal's intermediate representation.

Graal contains backends for GPGPU processing based on the PTX and HSAIL technologies. OpenCL backends are being developed by the Compiler and Architecture Design Group at the University of Edinburgh. A group of researchers at the University of Lugano, Switzerland, is working on extending Graal to allow for in-depth profiling of compiled code characteristics.

1.4 Structure of this Thesis

This thesis incorporates and extends upon the work published in [58] and [56]. The introduction contains sections taken from [53].

Chapter 2 sets the stage for the rest of this thesis: it introduces the Java language and the Java Virtual Machine. Certain aspects important to the topics of this thesis, such as the object system and the memory model, are explored in more detail. This chapter also provides an introduction to the Java HotSpot^{TM} Virtual Machine.

Based on this introduction, Chapter 3 presents the Graal compiler. It starts with an overview of Graal's characteristics in comparison to other compilers, and explains the different phases of the compilation process. The main part of Chapter 3 describes Graal IR, the intermediate representation used throughout the compiler. This includes frame states, which store the meta data used for debugging and deoptimization.

Chapter 4 introduces the concept of Escape Analysis and motivates it, with the help of several examples, in the context of Java. It provides an overview of the optimizations facilitated by Escape Analysis and classifies algorithms for Escape Analysis based on a number of criteria.

Chapter 5, which is the main part of this thesis, introduces our novel Partial Escape Analysis. Starting with an example, the algorithm and its implementation in Graal are explained in detail.

The interface used to extend Partial Escape Analysis for new node types is introduced in Chapter 6. Examples for non-trivial node types demonstrate the usage of this interface.

Several interesting case studies are presented in Chapter 7. This includes code snippets from Graal itself, excerpts from benchmarks, and synthetic examples that show certain characteristics of Partial Escape Analysis.

After a discussion of the effects of Partial Escape Analysis, Chapter 8 presents an evaluation of Partial Escape Analysis on a number of benchmarks. Apart from performance measurements, this includes criteria such as number of bytes allocated per benchmark iteration. Chapter 8 closes with an evaluation of the impact our optimization has on compile time.

Chapter 9 puts the topics presented in this thesis in context with related work. This includes two unpublished works similar to Partial Escape Analysis.

Chapter 10 then concludes this thesis with future work and conclusions.

Chapter 2

Introduction

This chapter introduces Java and the associated Java Virtual Machine execution model[1]. It starts with a description of the Java language and the implementation-independent Java Virtual Machine, and progresses towards the important parts of the actual system used in the implementation of the algorithms introduced in this thesis.

2.1 Java

Java[1] is a general-purpose object-oriented programming language developed by Sun Microsystems[2]. After its introduction in 1995 it quickly became one of the most successful and most widely-used programming languages, consistently ranking at the top of programming language rankings such as the TIOBE programming language index [59]. Listing 2.1 shows a simple “Hello World!” program implemented in Java.

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello␣World!");
  }
}
```

Listing 2.1: “Hello World!” in Java.

While it was originally designed to run on interactive TV appliances, Java’s success is tightly coupled to the success of the World Wide Web[3]. The ability to transfer

[1]The introduction to Java contains parts which are taken from [53].
[2]now a subsidiary of Oracle Corporation
[3]Java-enabled TV sets were, at the time, unviable due to the cost of hardware capable of running Java. Recently introduced Blue-ray TV appliances, however, use Java for their interactive components.

code over a network and to execute it in a controlled environment, without letting it compromise the surrounding system (also known as *sandboxing*), played a crucial role in the adoption of Java *applets* as interactive elements in HTML pages. New technologies such as powerful JavaScript virtual machines diminished the importance of Java applets, but the large number of Java installations facilitated by the applet technology jump-started the success of Java in many other areas.

Java gained widespread adoption in business applications because of a number of security-related features. Java pointers (called *references*) do not support pointer arithmetics, which is a major source of errors in lower-level languages. Java also does not allow explicit object deallocation, it instead provides a *garbage collector* that automatically frees unreachable objects. This avoids a large portion of common memory leaks and security problems. The memory model is sound enough to allow for *exact garbage collection*, which allows heap objects to be moved. This is a requirement for many high-performance garbage collection algorithms. Java is also largely platform independent, although in practice a programmer still needs to be careful to achieve full platform independence.

For programmers, Java is easier to learn and to maintain than many other languages. Java's object model is simple and easy to understand, but powerful enough for most use cases. For example, multiple inheritance is limited to interfaces, which is much simpler than multiple inheritance for classes. The Java programming language is accompanied by a large runtime library, which includes support for many advanced tasks such as XML-parsing, graphical user interfaces, etc. Java also has a concise, C++ - like syntax which provides familiarity for C and C++ programmers.

2.1.1 Java Objects

Java is an object-oriented language, albeit not "pure" since it employs two disjunct type hierarchies for primitive and object types. Nevertheless, code and data always live within a class or within an instance of a class. Static members (fields, methods and initializers) are associated directly with a class, while instance members (fields, methods and constructors) are associated with a specific instance of a class.

Ignoring the effect of access modifiers such as `private`, static members are accessible without the need for a specific context. Instance members, on the other hand, can only be accessed when the appropriate context, i.e., the class instance, is available.

Java also supports arrays of primitive and object types. While there is syntax for creating multi-dimensional arrays, all arrays with more than one dimension are implemented as arrays of arrays. Arrays are accessed using a zero-based index of `int` type, and their size is fixed when they are created. An attempt to access an element outside the bounds of an array will raise an `ArrayIndexOutOfBoundsException`.

All arrays and class instances need to be allocated explicitly and they are stored in the heap as separate objects. Java does not provide syntax for stack allocation or for declaring complex value types[4].

For the rest of this thesis, class instances are referred to as *instances*, and array instances will be referred to as *arrays*. The term *object* denotes either an instance or an array.

Every object has an object identity, so that simple equality comparisons between objects do not compare the contents of the objects, but only the object identities. Additionally, every object contains a monitor that can be locked and unlocked in order to serialize access to the object from multiple threads.

2.1.1.1 Object Creation and Modification

This section provides an overview over the different ways in which objects can be created and modified. While there are alternative ways to perform these actions, e.g., by using `sun.misc.Unsafe` or Java *reflection*, this list only contains the basic language primitives. Subsequent chapters will reference this list when describing how operations on objects are represented and processed.

Instance Allocation: `Instance i = new Instance();`
: The type of newly created instances is specified explicitly. Creating instances of types that are only available at runtime is possible using mechanisms such as reflection. New instances will be allocated from the garbage collected heap, and an allocation might raise an `OutOfMemoryError`.

Array Allocation: `int[] a = new int[size];`
: Allocating arrays is similar to creating instances, but the allocation needs to also specify the size of the newly created array.

Field Store: `i.field = value;`
: Sets the value of the field named “field”, or throws a `NullPointerException` if `i` is `null`.

Array Store: `a[index] = value;`
: Sets the value of the array element at index `index`. Throws a `NullPointerException` if `a` is `null` and an `ArrayIndexOutOfBoundsException` if `index` is out of bounds.

[4]Current plans for JDK9 contain extensive changes to the JVM to allow for value types.

Acquiring and Releasing Lock: `synchronized (o) { ... }`
The monitor of object o will be locked before the synchronized block is entered, and unlocked after the block is left. Throws a NullPointerException if o is null.

2.1.1.2 Accessing Object Properties

There is also a multitude of operations for accessing the contents and properties of objects. This includes getting the contents, getting a property and checking for a property.

Field Load: `var = i. field ;`
Retrieves the value of the field named “field”, or throws a NullPointerException if i is **null**.

Array Load: `var = a[index];`
Retrieves the value of the array element at index index. Throws a NullPointerException if a is null and an ArrayIndexOutOfBoundsException if index is out of bounds.

Array Length: `var = a.length;`
Retrieves the length of the array, or throws a NullPointerException if a is null.

Type Check: `o instanceof Type`
Evaluates to true if o is non-null and an object of the given (array or instance) type, and false otherwise.

Type Cast: `(Type) o`
Throws a ClassCastException if o is non-null and not an object of the given (array or instance) type.

Equality Comparison: `o1 == o2`
Checks whether the two objects refer to the same object instance, i.e., if they are associated with the same object identity.

2.1.1.3 Additional Operations on Objects

Inside a method, objects can be assigned to local variables. Objects can be passed out of, i.e., escape, the current method in a number of ways:

Static Field Store: `Type. field = o;`
Stores the reference o into the static field named “field” of class “Type”.

Method Call: Type.foo(o); or i.foo(o);

Calls the method “foo”, passing the reference o as a parameter. The former call is static, while the latter passes along the instance i as the receiver. Throws a NullPointerException if i is null.

Return Value: **return** o;

Returns the reference o from the current method.

Thrown Exception: **throw** o;

Returns the reference o from the current method by raising it as an exception. Throws a NullPointerException if o is null.

Additionally, an object can escape by storing it into a field or array element of a globally accessible object.

2.1.2 Java Virtual Machines

The Java Compiler (javac) compiles Java source code, contained in .java-files, into platform- and CPU-independent *bytecode* stored in one or more .class-files.

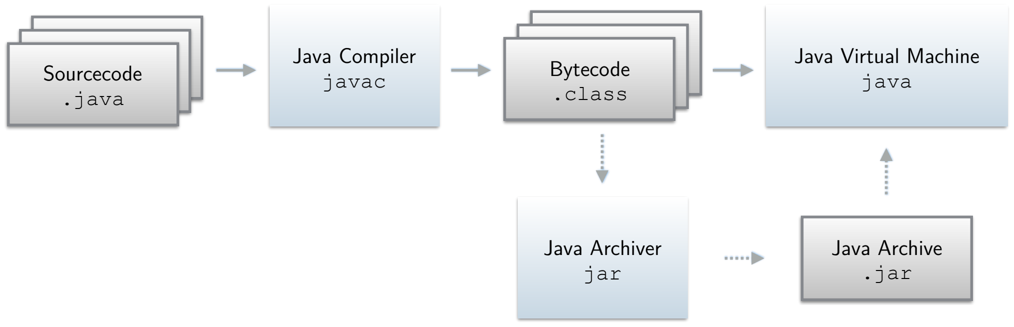


Figure 2.1: Overview of the artifacts generated by the Java system.

A *Java Virtual Machine* (JVM) then loads, verifies, and executes these bytecodes. It needs to adhere strictly to the semantics defined by the Java virtual machine specification [35]. It is important to note that all executable bytecodes are contained within methods, and in turn all methods are contained within classes. Although there can be more than one class within a single .java-file there is always exactly one class per .class-file. Multiple .class-files can be packaged into a compressed Java Archive (a so-called .jar-file), along with other resources needed by the application. Figure 2.1 provides an overview of these artifacts and the tools that generate them.

After the classes are loaded and verified for soundness the bytecodes are executed. They can be interpreted or compiled during execution (i.e., just in time); most JVMs employ a sophisticated combination of both. The JVM is a stack machine that uses an expression stack instead of registers, but in practice all JVMs compile bytecode into register-based, platform-specific code.

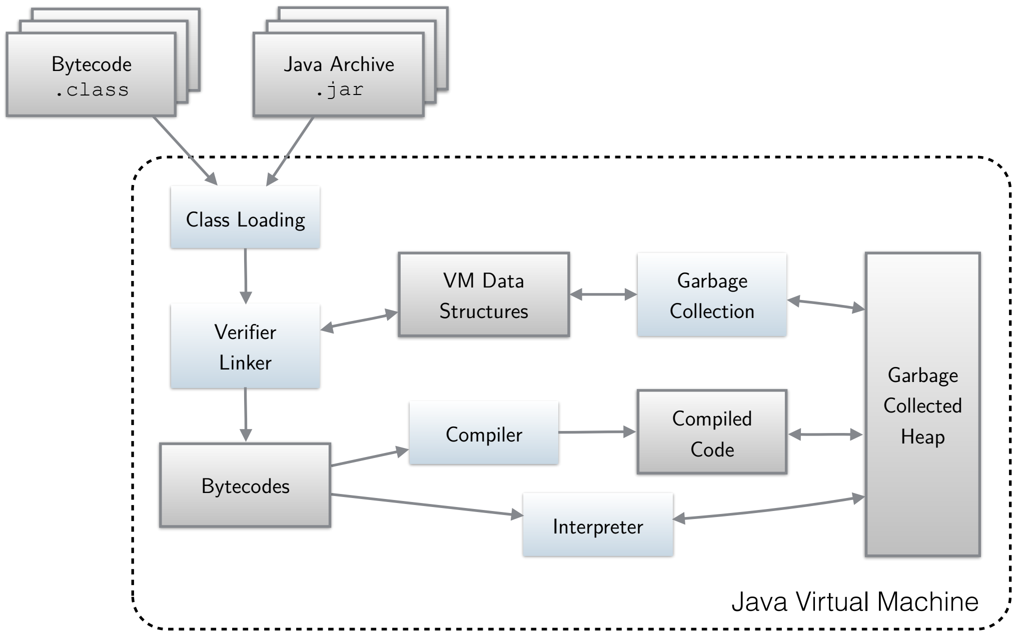


Figure 2.2: Java Virtual Machine overview.

Figure 2.2 provides a simplified overview of the most important subsystems of a Java Virtual Machine in the context of this thesis. It is important to note that compiled code, interpreted code and the garbage collector interact with the garbage collected heap.

2.1.3 Java Memory Subsystem

Java VMs are required to provide a garbage collected heap, into which all allocated objects will be placed. There is no deallocation primitive, because objects which are not referenced from a root (local and static variables) will be collected automatically. The fact that an object was reclaimed by garbage collection is not normally visible to running applications, although it can be observed through *finalizers* and *reference queues*.

The contents of a newly allocated object will be initialized with default values (0, false or null), so that an application can never inspect the uninitialized memory of a newly allocated object. There is also no way to convert the reference to an allocated object into a numeric type or to convert a numeric value into a reference. It is thus impossible in Java to perform pointer arithmetics.

All field accesses are statically verified, so that accessed fields are guaranteed to exist in the referenced object. For array accesses it is checked that the index is within the bounds of the bounds of the array at runtime, so that array accesses cannot be used to observe arbitrary memory locations.

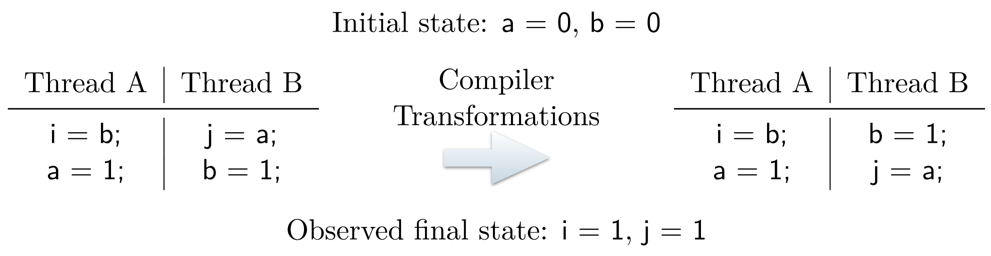
Due to these properties, Java is able to provide a sound security encapsulation for applications with limited privileges. Also, many common programming mistakes lead to more meaningful error messages or are prevented altogether. In the context of this thesis, however, the implications these properties have on compilation are more important: The compiler can safely assume that accesses to disjunct fields, accesses to instances or arrays with disjunct object identity, accesses to arrays of disjunct type and accesses to array elements with disjunct indexes can never influence each other.

For example, writing to field “a” can never influence a read from field “b”, which means that the compiler can reorder these field accesses. While this might seem trivial, properties such as this are not easy to prove within languages that support pointer arithmetics, such as C++.

2.1.3.1 Java Memory Model

For single-threaded applications, the above conditions for reordering are sufficient to determine the valid tranformations a compiler can perform. Such an application can always expect to observe a behavior that appears to be a sequential execution of all instructions.

For multi-threaded applications, however, there is no efficient implementation strategy for a sequentially interleaved execution model, especially on modern multi-core architectures. The Java Memory Model (JMM) [29, 36, 61] provides an execution model for multi-threaded applications, by defining a compromise between efficient VM implementations and useful application behavior.



Initial state: a = 0, b = 0

Thread A	Thread B
i = b;	j = a;
a = 1;	b = 1;

Compiler Transformations

Thread A	Thread B
i = b;	b = 1;
a = 1;	j = a;

Observed final state: i = 1, j = 1

Figure 2.3: Example showing the effect of code reordering on multi-threaded code.

The JMM allows most of the transformations that would be valid for single-threaded applications. This can have surprising results, as shown in Figure 2.3. The code on the left is the original code. Given the initial state a = 0 and b = 0, the result can either be i = 1 and j = 0 or i = 0 and j = 1, depending on the actual instruction interleaving. However, the compiler is free to transform the code into the version shown on the right-hand side. By executing Thread A in-between the two statements of Thread B, the originally impossible final state i = 1 and j = 1 can be reached.

While a more restrictive model might seem more useful, such a model would be hard to implement efficiently. Even if the compiler performs no transformations like in the example above, the limited cache coherency of multi-core CPUs and multi-processor systems can still have the effect of non-trivial code reordering, which can only be prevented by explicit and expensive memory barriers. The JMM does not force Java VMs to execute multi-threaded code in a way that behaves like a sequential interleaving of instructions.

The JMM does, however, specify a certain ordering of the observability of effects around synchronization primitives and volatile fields, which is usually implemented by inserting memory barriers. It is important to note that this is limited to effects that *can* be observed; if another thread cannot observe the effect for other reasons, e.g., because no reference to the object whose field is changed exists outside the current thread, no memory barrier is required.

2.2 Java HotSpot^{TM} Virtual Machine

The Java HotSpot^{TM} VM was introduced along with Java 1.3. It can either interpret Java bytecodes or use one of its two just-in-time (JIT) compilers, called *client compiler* [34] and *server compiler* [43], to compile bytecodes into optimized machine code. The name "hotspot" refers to the fact that it detects frequently executed methods, called *hot spots*, which are then targeted for compilation and further optimization.

HotSpot contains two different interpreters, the C++ interpreter and the template interpreter. While the C++ interpreter is intended to ease porting to new architectures by using as little architecture-dependent code as possible, the template interpreter is a highly-optimized interpreter assembled at startup from hand-written assembly snippets. The internal state of these interpreters corresponds closely to the state described by the Java virtual machine specification. The template interpreter can be executed in a mode in which it collects information about the behavior of the currently executing method. This method profile will include properties such as the probability of branching instructions and type profiles for calls.

Apart from an experimental llvm-based compiler ("Shark"), there are two main compilers in HotSpot, the client compiler and the server compiler. The client compiler uses a simple control flow graph, which is turned into compiled code using hand-crafted assembly snippets, in order to quickly generate compiled code with acceptable performance. The server compiler is based on a program dependence graph and uses a bottom-up tree rewriting component, derived from a description written in a domain specific language, to create highly optimized compiled code. It performs more sophisticated optimizations, so that compilation takes significantly longer.

HotSpot also contains a sophisticated memory management system that performs *precise garbage collection.* It requires the exact size and layout of an object and all object pointers within it to be known to the runtime system. Every object is preceded by a two-word header that contains a pointer to the class of the object and additional information, such as locking and garbage collection bits.

2.2.1 Deoptimization

Under some circumstances, the code compiled by the client and server compilers will be based on certain assumptions, e.g., the current hierarchy of loaded classes. When one of these assumptions is violated, the code will be invalidated and subsequently removed from the collection of compiled methods (which is referred to as *code cache*).

For a currently running execution of the method in question, however, this is not enough, because it is not safe to continue executing it. The execution needs to be transferred back to the interpreter, which does not take any assumptions and therefore can correctly handle all cases. This tranfer from compiled code back to the interpreter is called *deoptimization* and was pioneered in the SELF system by Hölzle et al. [28].

For each point in compiled code at which such a deoptimization can happen, the compiler stores information that allows the runtime system to reconstruct an interpreter stack frame from the native stack frame of an execution of the compiled code. For Java, this consists of the current *bytecode index* (bci), the current method, the number and values of the local variables and stack expressions, and the locks currently held by this frame.

2.2.2 Thread-local Allocation Buffers

In a system that supports multithreading, such as the HotSpot VM, allocating memory from a heap shared between all threads is a complex operation, since it usually requires some form of synchronization between threads. Memory allocation is a very frequent operation in most Java applications, therefore the HotSpot VM provides a thread-local allocation buffer (TLAB) for each thread.

Chunks of memory are allocated from the shared heap whenever the TLAB is full. The TLAB for each thread is sized dynamically based on the thread's previous behavior. Threads that frequently allocate large amounts of memory will acquire memory from the shared heap in larger pieces.

Within the TLAB, allocations are managed by thread-local “top” and “end” pointers. The former represents the end of the last object in the TLAB, while the latter points to the end of the TLAB. Whenever an object is allocated, its size is added to “top”. If the new “top” is beyond “end”, the TLAB does not have enough free space for the new object and needs to be emptied into the shared heap.

Chapter 3

Graal

This chapter provides an overview of the Graal compiler, starting with a characterisation of the compiler's most prominent features. It also explains the overall compilation process and gives a detailed introduction into Graal's intermediate representation, Graal IR. The Graal IR is explained in detail, because it motivates many of the choices taken in the algorithms presented in later chapters. After a section on the frame state concept, it concludes with a detailed list of how operations on objects are represented in Graal IR.

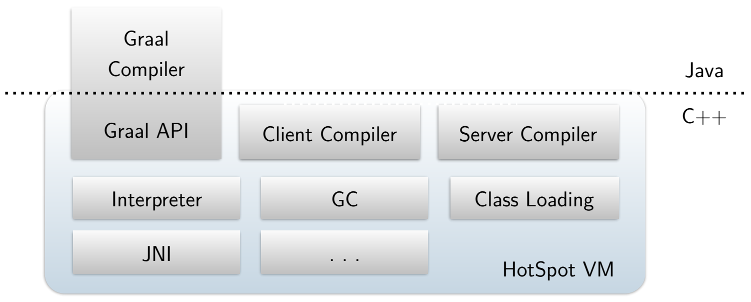


Figure 3.1: Overview of HotSpot and Graal components.

Graal [17, 18, 56, 57, 58] is a just-in-time compiler for Java which is written in Java and developed as part of the Graal OpenJDK project [41]. The Graal VM is a modification of the HotSpot VM which replaces the client and server compilers with the Graal compiler. It reuses all components of HotSpot that are not directly involved in compilation: interpreter, garbage collection, class loading, JNI, the class library, and many others, as shown in Figure 3.1.

Graal tries to leverage the approachability and malleability of Java code for compiler development. This has already manifested in numerous extensions to Graal: Graal has been ported to other VMs (Maxine [69] and Substrate VM), it received new CPU backends (SPARC and ARMv8), and there are multiple projects supplying GPU backends for Graal, e.g., Sumatra [42].

3.1 Compiler Characteristics

Graal differs heavily from its distant ancestry, the C1X compiler [60] and the HotSpot client compiler [34], with the most notable changes being:

Intermediate Representation Graal introduces a new intermediate representation called *Graal IR* [17]. For a detailed description see Section 3.2.

Late Inlining Many dynamic Java compilers perform inlining during the bytecode parsing step. This is not desirable, because all inlining decisions have to be taken very early, so they cannot benefit from the additional information later compiler phases and optimizations gather about the code. Some optimizations, such as constant folding and global value numbering, need to be performed during bytecode parsing to make reasonable inlining decisions possible at all, which unnecessarily complicates the parsing step.

Additionally, parsing the bytecode of a method using early inlining usually incorporates information such as the current inlining context or the remaining inlining budget. Thus, the result of of bytecode parsing cannot be cached in a useful way. There is significant potential for reuse of cached parsing results in typical Java applications [56].

Graal performs late inlining, i.e., it parses all methods independently of one another, and combines them only later in the compilation process, so that it can cache the result of bytecode parsing and can employ a more sophisticated inlining strategy.

Fine-grained Optimization Phases The Graal IR includes data structures that allow for efficient iteration of all instances of a specific node type in the graph. This, combined with the def-use information in the graph, means that optimization phases can look at, and operate on, specific nodes, without having to perform whole-graph iterations.

Aggressively Optimistic Graal relies heavily on the information collected by the profiling interpreter. Parts of methods that were never executed in the profiling interpreter are not even parsed, and neither are references to uninitialized classes and methods. Graal only creates exception-handling code in places where, according to profiling information, exceptions happened previously.

Assumptions Graal uses a mechanism called static and dynamic assumptions to ensure that it will still generate correct code in the presence of aggressively optimistic optimizations. Static assumptions are assumptions about the state of the loaded application, e.g., that there is only one implementation of a specific interface. If one of these static assumptions is invalidated, e.g., by further class loading, the compiled code will be invalidated and active executions of it will be *deoptimized* (see Section 2.2.1).

Dynamic assumptions are assumptions that cannot be statically proven, but are hinted at by profiling information. These assumptions still need to be checked at run time, e.g., a condition that was always false during profiling, and thus leads to a never executed branch, still needs to be checked (*guarded*) at runtime. If one of these guards fails, the compiled code will invalidate itself, again using deoptimization.

Snippets As an improvement over C1X's templating mechanism, *XIR* [60], Graal introduces the concept of Java snippets for simplifying ("lowering") complex operations in the IR. Java methods, written to adhere to certain rules, describe how operations such as monitorenter and new can be decomposed into lower-level operations.

3.1.1 Compilation Process

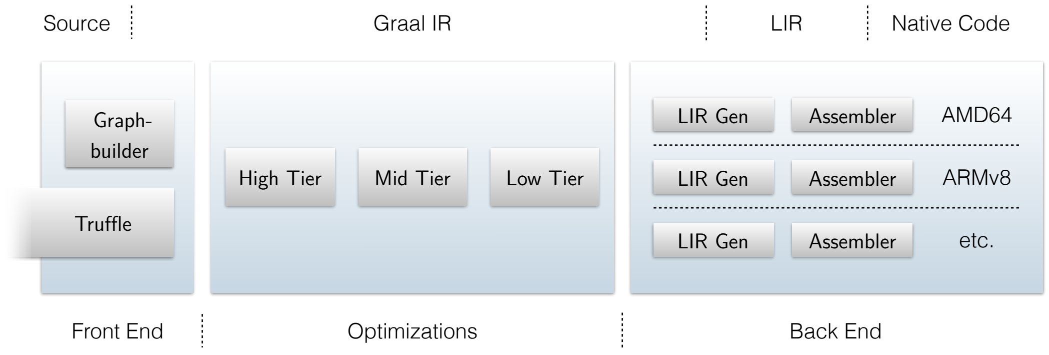


Figure 3.2: Graal compilation process.

Like most compilers, Graal is split into a source-specific part, a part that performs most optimizations, and a target-specific part. Figure 3.2 contains a schematic overview of Graal's compilation process:

- The *Front End* is responsible for turning the Source representation, e.g., bytecode, into Graal IR. The *Graphbuilder* parses bytecode and incorporates profiling feedback gathered by the interpreter. Another front end available within the Graal project is the *Truffle framework* [76, 77], which can be used to develop runtime environments for a wide variety of languages.

- The generic, platform-independent optimization phases are split into three tiers: The most high-level optimizations, such as inlining, loop optimizations and escape analysis, occur within the *High Tier*. The *Mid Tier* deals with memory, safepoint and guard optimizations. Finally, the *Low Tier* prepares the graph such that it can be converted into the low-level representation.

  Graal IR is designed to allow these tiers to be deconstructed into small, independent compiler phases that focus on a single task. A *lowering phase* at the end of each tier deconstructs a certain set of operations into more low-level operations. Constant folding, global value numbering, control-flow-sensitive value propagation and certain other optimizations happen throughout all optimization stages.

- The *Back End* translates the high-level Graal IR into a low-level IR, *LIR*, which is already target-specific. After register allocation and peephole optimizations, the actual native code for the LIR instructions is emitted.

3.2 Graal IR

Graal translates Java bytecode into a high-level intermediate representation called Graal IR [17], on which it performs all optimizations. This SSA-based intermediate representation models both control flow and data dependencies between nodes. Graal IR [17] is therefore a hybrid structure based on a combination of two directed and acyclic[1] graphs: a control flow graph and a data dependency graph. Each node in the graph produces zero or one value, and the structure of each node type is defined by a specially annotated Java class. Nodes can be connected by input dependencies ("inputs") and control flow anti-dependencies ("successors"), and the graph structure keeps def-use information for these edges, i.e., looking up the usages and predecessors of a node is an efficient operation.

3.2.1 Declarative Node Types

Node types used within Graal IR are described in a declarative syntax and with conventions about the structure of their implementation classes. Listing 3.1 shows an example

[1]Phi nodes and proxy nodes being the only exception.

```
@NodeInfo(shortName = "+")
public class IntegerAddNode extends FloatingNode {
  @Input private ValueNode x;
  @Input private ValueNode y;

  public IntegerAddNode(ValueNode x, ValueNode y) {
    this.x = x;
    this.y = y;
  }

  public ValueNode getX() {
    return x;
  }

  public void setX(ValueNode x) {
    updateUsages(this.x, x);
    this.x = x;
  }

  // ...
}
```

Listing 3.1: Example declaration of a node class.

of a simple node class representing the addition of two values. The properties of a node type are derived from a number of attributes of its class:

- In order for it to be recognized as describing a node type, the class directly or indirectly derives from Node.
- The optional @NodeInfo annotation provides a label for the node type. If this annotation is missing, the label is derived from the class name by removing the "Node" suffix (if present).
- Inputs and successors are marked as such with @Input and @Successor annotations. They can reference a Node directly, or use a NodeInputList or NodeSuccessorList if an input or successor list is needed.
- Unless a list is used, setters for inputs or successors need to call updateUsages or updatePredecessors to keep the graph's use-def edges in sync.

Deriving from the Node class provides node classes with efficient iterators for inputs, successors, usages and predecessors. These are used, in turn, to implement fast replacement of nodes at their usages and predecessors. It is important to note that, while inputs and successors are ordered, usages and predecessors are unordered collections.

Node classes can also be augmented with numerous additional capabilities by implementing predefined interfaces. The most important ones are:

ValueNumberable
By implementing this marker interface, a node type is automatically subject to global value numbering, i.e., two nodes with the same inputs, the same primitive field values and equal object fields will automatically be combined into a single node.

IterableNodeType
Marking a node class with this interface tells the system that iterating over all nodes of this type in a graph should be an efficient operation. This is implemented by keeping nodes of the same type in a linked list [17].

Canonicalizable
This interface lets node classes implement local optimizations that replace the current node, e.g., constant folding and local strength reduction.

Simplifiable
In constrast to canonicalization, the simplification interface allows for non-local changes, e.g., removing a branch of an if statement that has a constant condition.

3.2.2 Hierarchy of Node Types

Figure 3.3 contains an overview of Graal's node type hierarchy. Nodes can be either floating, or fixed within the control flow graph. The special VirtualState nodes are the only exception to this rule. Fixed nodes have predecessors and/or successors, so that they form a control flow graph, which is the "skeleton" of the graph. Floating nodes, on the other hand, are only connected to the graph by inputs and usages, and will automatically be deleted if they are not used (transitively) by a fixed node. Additionally, floating nodes will automatically be global value numbered upon insertion, because FloatingNode implements ValueNumberable.

The GraphBuilder turns the stack-based bytecode into static single assignment (SSA) form [14], which means that phi nodes are used to merge values at control flow merges. All values are also proxied by Proxy nodes at loop exits, so that the graph is in loop closed form [19]. This means that values flowing from the loop body to the successors of the loop are explicit in the graph, which greatly simplifies loop optimizations like peeling and unrolling. The loop closed form is maintained throughout high tier, but removed later on to allow for additional optimizations.

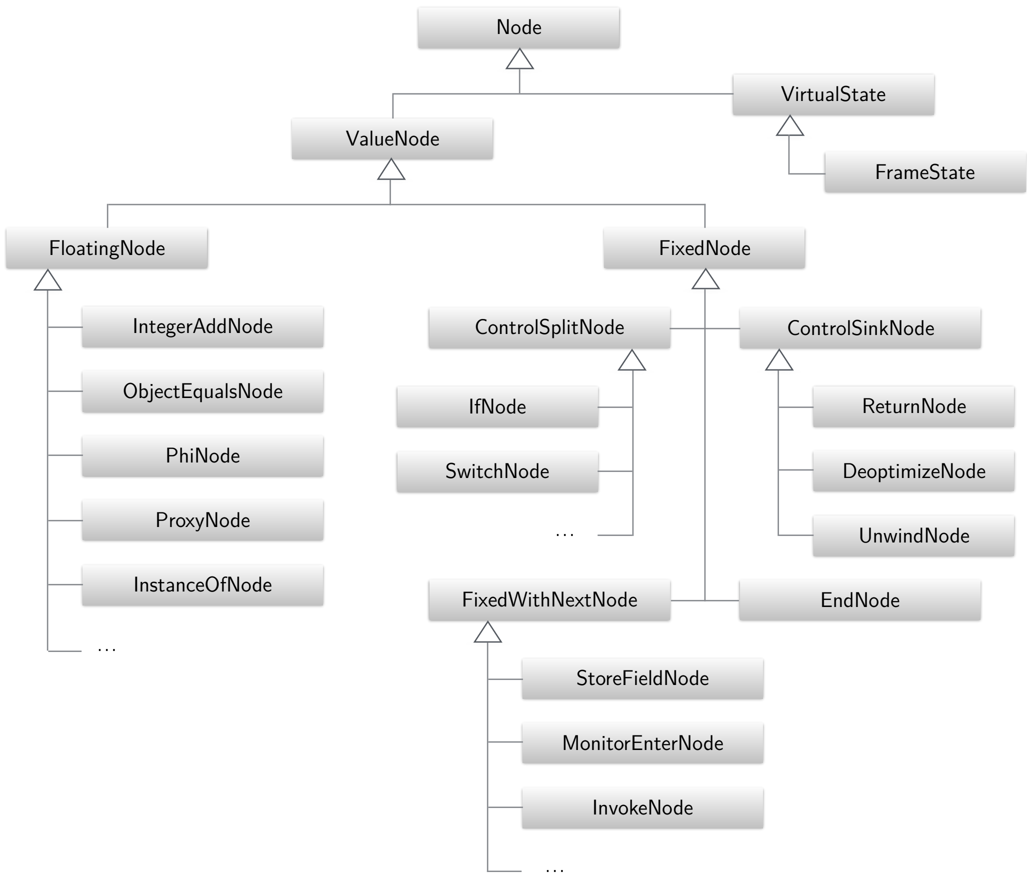
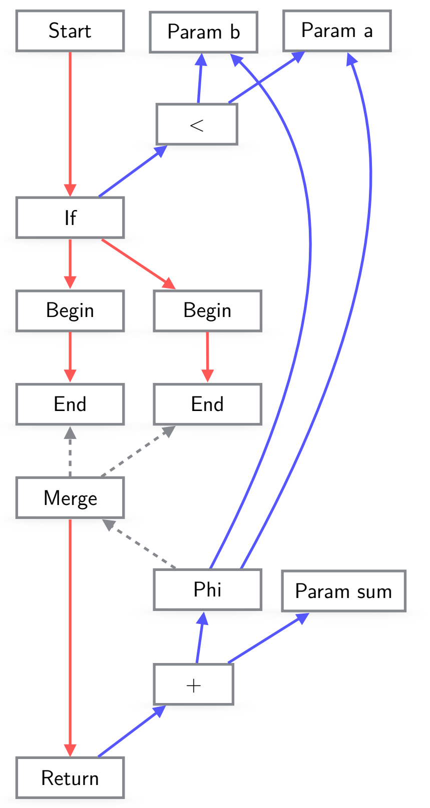


Figure 3.3: Hierarchy of Graal Node Classes



```
int addMin(int a, int b, int sum) {
  int min;
  if (a > b) {
    min = b;
  } else {
    min = a;
  }
  return min + sum;
}
```

Figure 3.4: Small example along with the Graal IR that will be generated for it.

Figure 3.4 shows the Graal IR generated for a small example method. Some specific properties of Graal IR are visible in this example:

- Successor edges between nodes create the control flow of the graph. In the example, these edges are represented by red downwards-pointing arrows.
- Input edges between nodes create the data flow of the graph. These edges are represented by upwards-pointing arrows. Solid blue arrows are real data flow, while dashed grey arrows are structural dependencies.
- Each graph starts with a special Start node.
- The targets of a control flow split are always Begin nodes, and the predecessors of a merge are always End nodes.
- The Merge node uses inputs to reference its predecessor End nodes. The predecessors of a node are an unordered collection in Graal IR. Therefore, connecting the End nodes to the Merge node via normal control flow would mean that it is not clear which End node a specific input of a Phi function refers to. By using

inputs, the Merge node maintains the order of its predecessors, and the ith input of the Phi function can be associated with the ith End node.

- The “greater than” comparison was immediately replaced with a “less than” comparison with switched inputs. Graal IR only supports “less than” and “equals” comparisons, because it tries to be as canonical as possible. All other comparisons can be transformed into one of these two options by switching input operands and/or switching the If node’s successors.

The graph in Figure 3.4 contains a control flow structure which consists of all nodes connected by successor edges (red arrows). These so-called fixed nodes will be executed in a specific, predefined order. Floating nodes, such as the addition and the comparison, are only connected to other floating and fixed nodes via data flow edges (blue arrows), and they need to be placed at a specific point in the control flow during the final stages of compilation. There are usually many positions where an operation could be placed, and it is up to the so-called *Scheduler* to determine a good one according to a chosen strategy (e.g., earliest possible, latest possible, latest possible but out of loops).

3.2.3 Frame States

Graal is an aggressively optimistic compiler, which means that, for many operations, it generates code which can only correctly handle the common case or the cases that were encountered during profiling. For example, it never expects a field access to throw a NullPointerException, unless the profiling information indicates otherwise. Graal will make sure that the target of the field access is tested for a null value in some way, but it will not create any code to raise and handle the exception.

Such an assumption, which still need to be checked when a method is executed, is considered to be a dynamic assumption. Graal can also take static assumptions on the current state of the Java VM, e.g., that only one implementation of a specific interface is currently loaded.

When one of these assumptions is invalidated, e.g., by loading a new class or by entering an unexpected branch, the execution needs to be transferred from compiled code back to the interpreter (which does not make any assumptions and can execute all code). This switch back to the interpreter is called *deoptimization* [28] (see Section 2.2.1) and requires a translation from the machine state (native stack frames) back to the Java VM state (interpreter stack frames).

Within Graal IR, it is possible to obtain a valid mapping to Java VM state for all positions that can cause a deoptimization to occur. This mapping is expressed as FrameState nodes and consists of the current position (bytecode index and method), the local variables, the contents of the expression stack and the locked objects.

After inlining, one position can map to multiple Java VM stack frames. A frame state thus contains a reference to an *outer frame state*, which is the caller's state. This reference is used to create chains of FrameState nodes that describe the state of all inlined methods at the current position.

The Graal IR keeps the frame states not at the points where the actual deoptimizations take place, but at so-called *state splits*. State splits are operations such as field stores and method calls which have side-effects and cannot be safely re-executed. Operations such as an integer addition do not have a side effect and re-executing them will lead to the same result. A memory allocation does have a side effect on the VM, but it is not visible to the application, therefore it is also not considered to be a state split.

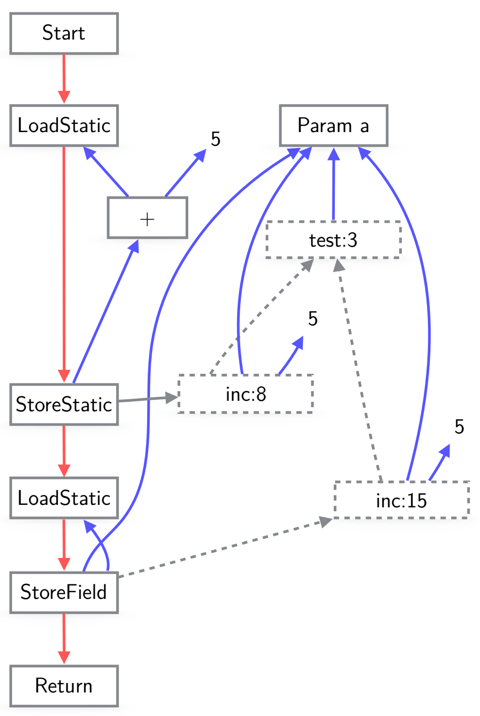
State splits are associated with a frame state that describes the state *after* the state split. A machine code instruction that causes or can cause a deoptimization will be associated with the after state of the last side-effecting instruction. All instructions that were executed in-between do not have side effects and can therefore be safely re-executed in the interpreter.

An example that includes inlined frame states is shown in Figure 3.5. The method test was compiled, and within it the call to inc was inlined. The compiler graph shown in the example therefore contains code from both the test and the inc method. It is important to note that the LoadStatic nodes are fixed nodes[2].

The StoreField and StateStatic nodes in this example are state splits and, therefore, need frame states. These frame states refer to the bytecode indexes 8 and 15 in the method inc, which are the instructions *after* putstatic and putfield. The inputs of the frame states point to the values of the local variables and the stack expressions at the frame state's position.

The frame state for test has one local variable, which contains the parameter a (represented by the Param a node). In addition to the parameter a, the frame states for inc contain the constant 5 as the current value of the parameter add.

[2]They will be transformed into floating nodes during later stages of compilation. This *lowering* mechanism is of no consequence to Escape Analysis, and is therefore not covered in this thesis.



```
class A {
  int value;
  static int staticValue;
}
static void inc(A a, int add) {
  A.staticValue += add;
  a.value = A.staticValue;
}
static void test(A a) {
  inc(a, 5);
}

static void inc(A, int);
   0: getstatic A.staticValue
   3: iload_1
   4: iadd
   5: putstatic A.staticValue
   8: aload_0
   9: getstatic A.staticValue
  12: putfield A.value
  15: return
static void test(A);
   0: aload_0
   1: bipush 5
   3: invokestatic inc(A, int)
   6: return
```

Figure 3.5: Example for frame states of inlined methods, along with the Java source and the bytecode of the methods.

Both inc frame states refer to an outer frame state within test at bytecode index 3, which is the location of the invokestatic instruction. In case one of the frame states of the stores is used for deoptimization, the reference to the outer frame state is used to reconstruct all frames at the store, i.e., the frames for both the test and the inc methods, in order to undo the inlining performed during compilation.

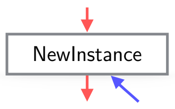
Such a deoptimization will occur, e.g., if null is passed as parameter a to the test method. The second field store instruction, which is the store to value, will fail, because it would have to store to a null instance. As this is an infrequent case, the Graal IR does not contain code to raise the NullPointerException. Deoptimization will occur, which creates interpreter frames for the test:3 and inc:8 frames.

It is important to note that the inc:15 frame state is not used in this case, because it is only valid *after* the store was executed. The interpreter will re-execute the bytecodes of inc starting at index 8. It will thus re-execute the aload_0 and getstatic instructions and raise the correct exception when it tries to execute putfield.

3.2.4 Object Creation and Modification

The following graph snippets show how object creation and modification is represented within the Graal IR:

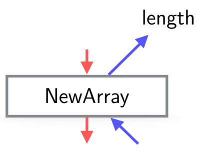
Instance Allocation:



NewInstance nodes are used to create new instance objects. It will allocate the object in the heap, set the object header and initialize all fields to their default values (0, false or null). The class of the object to be allocated needs to be known at compile time.

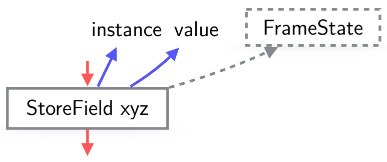
The blue data dependence arrow indicates usage of the created instance by later instructions.

Array Allocation:



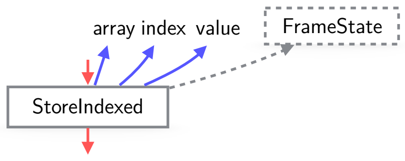
New arrays are allocated by NewArray nodes. The length of the array is a runtime input to the node, while the element class needs to be known at compile time. All entries in the array will be initialized to their default values.

Field Store:



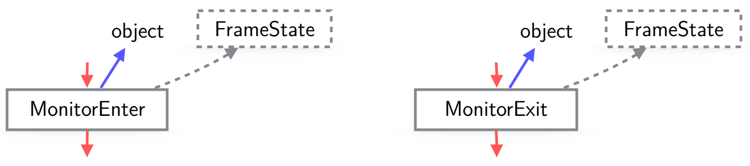
StoreField nodes set the value of a field in the given instance object. The field store has a side effect, so it cannot be re-executed after a deoptimization. Therefore, it is a state split node and has an associated FrameState.

Array Store:



StoreIndexed nodes are similar to StoreField nodes, except for the fact that they also require the array index to be stored to.

Acquiring and Releasing a Lock:

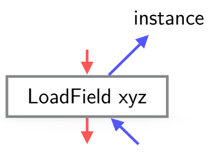


MonitorEnter nodes and MonitorExit nodes are used to acquire and release locks. These operations have a side effect, so they cannot be re-executed after a deoptimization. Therefore, they are state splits and have an associated FrameState.

3.2.5 Accessing Object Properties

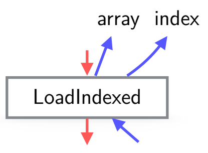
The following graph snippets show how access to object properties is represented within the Graal IR:

Field Load:



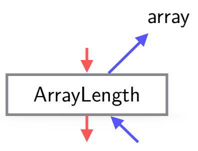
The LoadField node, which is used to load the value of an instance field, has no side effect, so it does not need a FrameState.

Array Load:



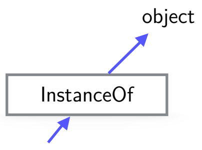
The LoadIndexed node is similar to the LoadField node, except for the fact that it also requires the array index to be loaded from.

Array Length:



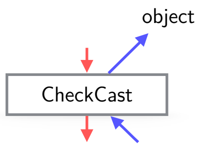
The ArrayLength node produces the length of the given array.

Type Check:



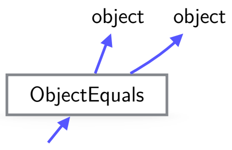
The InstanceOf node, which is used, e.g., as a condition for an if node, tests if the given object is an instance of a specified type.

Type Cast:



The CheckCast node will deoptimize if the given object is not either null or an object of a specified type. After deoptimization, the execution in the interpreter will continue at the closest previous point for which a FrameState is defined. It will re-execute the cast and thus throw a ClassCastException.

Equality Comparison:

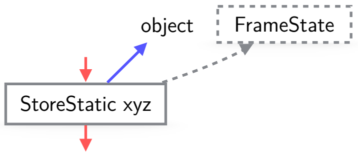


The ObjectEquals node checks whether two given objects are either both null or the same object, as determined by object identity. It can also be used as a condition, e.g., for If nodes.

3.2.6 Additional Operations on Objects

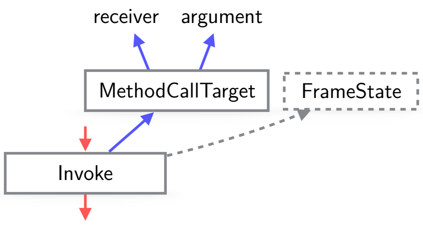
The following graph snippets show how additional operations on objects, such as passing them to method invocations, are represented within the Graal IR:

Static Field Store:



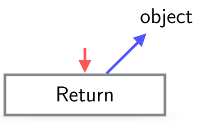
Objects can be stored into static fields by a StoreStatic node. In this case, no instance is necessary.

Method Call:



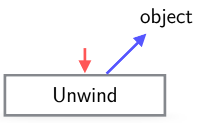
Invoke nodes, which represent method invocations, use a MethodCallTarget node for representing the target of the invoke. This node references both the receiver (for non-static calls) and the parameters of the method call. The MethodCallTarget node serves as an abstraction that encapsulates the common parts of different platform-specific Invoke node types.

Return Value:



The Return node can return a value if the current method is not of type void.

Thrown Exception:



If an exception is thrown and caught within the same compilation scope, this will be represented as ordinary control and data flow. Only if the exception is not handled in the same compilation scope, an Unwind node is used to pass the exception to the next outer frame at run time.

Chapter 4

Escape Analysis

This chapter introduces the basic concepts of Escape Analysis. It elaborates on the optimizations enabled by Escape Analysis and why they are beneficial, and explains characteristics of different algorithms for Escape Analysis that put the new techniques introduced in later chapters into context.

Escape Analysis checks whether an allocated object escapes, i.e., can be used outside, the allocating method or thread. This happens if it is assigned to a global variable or to a heap object, if it is passed as a parameter to some other method, or if it is returned from the current method. Compilers use Escape Analysis to determine the dynamic scope and the lifetime of allocated objects. The result of this analysis allows the compiler to perform numerous optimizations on operations such as object allocations, synchronization primitives and field accesses.

Frameworks such as Java EE [30] and languages such as Scala [40] confront Java VMs with additional levels of abstractions, which usually consist of small and short-lived objects. Escape Analysis can help in removing these abstractions, thereby offsetting the costs these new technologies incur in exchange for their advantages. Java itself also introduces additional layers of abstraction, such as in the following examples:

```
ArrayList<Integer> list = ...
int s = 0;
for (int v : list) {
    s += v;
}
return "sum: " + s;
```

```
StringBuffer buffer =
        new StringBuffer();
buffer.append(...);
buffer.append(":");
buffer.append(...);
return buffer.toString();
```

Each time the code in the left example is executed, it will allocate an instance of a list iterator for the for-each loop. The compiler can transform the for-each loop into a counting for loop here, avoiding the allocation of the iterator. The StringBuffer in the example on the right-hand side is a synchronized data structure, so that every operation needs to acquire the object's lock. However, it is obvious that the buffer object does not escape to another thread in this example. The lock can therefore never be contended, so no synchronization is necessary.

4.1 Example

Listing 4.1 shows a small piece of code that will serve as an example to show the benefits of Escape Analysis: The getValue method creates a new Key object and checks whether it is in the cache. If so, the method returns the cached value. Otherwise, it creates and returns a new value (the contents of the createValue method are of no consequence and are therefore not discussed here).

```java
public class Example {
  private static class Key {
    public int idx;
    public Object ref;
    public Key(int idx, Object ref) {
      this.idx = idx;
      this.ref = ref;
    }
    public synchronized boolean equals(Key other) {
      return idx == other.idx && ref == other.ref;
    }
  }
  private static Key cacheKey;
  private static Object cacheValue;

  public static Object getValue(int idx, Object ref) {
    Key key = new Key(idx, ref);
    if (key.equals(cacheKey)) {
      return cacheValue;
    } else {
      return createValue(...);
    }
  }
}
```

Listing 4.1: Simple example.

When getValue is compiled, the compiler will most likely perform some inlining, which might cause the actually compiled code to look like Listing 4.2. The Key constructor and the equals method have been inlined into the getValue method, and a synchronized block was created to achieve synchronization on the inlined equals method.

When Escape Analysis examines the resulting method, it will come to the conclusion that no reference to the allocated Key object escapes from the current compilation scope.

```
public static Object getValue(int idx, Object ref) {
  Key key = alloc Key; // pseudocode
  key.idx = idx;
  key.ref = ref;
  Key tempKey = cacheKey;
  boolean temp;
  synchronized (key) {
    temp = key.idx == tempKey.idx && key.ref == tempKey.ref;
  }
  if (temp) {
    return cacheValue;
  } else {
    return createValue(...);
  }
}
```

Listing 4.2: getValue method from the example in Listing 4.1 after inlining.

This implies that no references to the object exist after the method has returned, and that no other thread can ever see a reference to this object. The compiler can use these observations to perform a number of optimizations, the result of which might look like Listing 4.3. The allocation was replaced with the local variables idx1 and ref1, and the synchronized statement was removed entirely.

Both Listing 4.2 and Listing 4.3 use a temporary variable for the value read from cacheKey. Re-reading the value would alter the behavior of the code, because different values could be read each time.

```
public static Object getValue(int idx, Object ref) {
  int idx1 = idx;
  Object ref1 = ref;
  Key tempKey = cacheKey;
  if (idx1 == tempKey.idx && ref1 == tempKey.ref) {
    return cacheValue;
  } else {
    return createValue(...);
  }
}
```

Listing 4.3: Example from Listing 4.2 after Scalar Replacement and Lock Elision.

4.2 Optimizations

The above example employs *Scalar Replacement* and *Lock Elision* to perform the actual optimization. There are a number of optimizations commonly used in conjunction with non-escaping objects:

Stack Allocation The allocation of an object on the garbage-collected heap can be replaced with an allocation on the stack. This reduces the pressure on the garbage collection subsystem, because less memory is allocated on the heap.

Zone Allocation Another way of reducing the pressure on garbage collection is allocation in non-garbage-collected areas such as zones. Zones are heap areas with a known, limited lifetime, which can be freed in bulk when a certain scope is left.

Scalar Replacement Scalar Replacement [11] substitutes the fields of an object by local variables, thus avoiding the allocation of the object on the heap. This also leads to less work for the garbage collector. Additionally, the fact that values will not flow through memory any more usually opens up opportunities for other optimizations such as *Constant Folding* and *Value Propagation*.

The Escape Analysis algorithm needs to show more than just the fact that the allocated object does not escape. When Scalar Replacement is performed, the object's identity gets lost and the object's class is not stored in the object itself. One implication of this is that Scalar Replacement is not possible for objects that are merged at phi functions. Consider the following examples:

```
Base x;
if (...) {
  x = new SubclassA();
  x.f = value1;
} else {
  x = new SubclassB();
  x.f = value2;
}
// ...
return x.f;
```

```
Base x, y;
if (...) {
  x = new SubclassA();
  y = b;
} else {
  x = new SubclassB();
  y = null;
}
// ...
if (x == y) {
  // ...
}
```

The example on the left would require the system to recognize that only the field "f" of the base class is required and needs to be retained at the merge. The example on the right uses object identity explicitly, which also prevents Scalar Replacement. Solutions to deal with these problems are presented in Section 10.1.

In essence, Escape Analysis needs to collect enough information to prove for every variable of the allocated object's type that it either always references the allocated object or never references it. If it succeeds, *Scalar Replacement* can be used to perform the replacement and remove the allocation.

Lock Elision An object's lock will never be contended if it cannot escape to other threads. Therefore, *Lock Elision* can remove the synchronization, a potentially very expensive operation. Java requires synchronization to also be a barrier for reordering memory accesses (see Section 2.1.3.1). However, given that Java requires this barrier only with respect to threads that can observe the synchronization event, it can also safely be removed.

For some methods that frequently allocate short-lived objects, these optimizations are key to achieving good performance. Compilers will perform *aggressive inlining* to make the compilation scope larger, in order to pull as many operations as possible into the current compilation scope and to expose as many operations as possible to Escape Analysis and the associated optimizations.

4.3 Classification of Algorithms

There is a wide range of algorithms for performing escape analysis. They differ mainly in the size of the scope they work on and how much context they take into account.

4.3.1 Object Scope

Basic Escape Analysis implementations, such as the one used in current versions of the v8 JavaScript engine [21] and in LuaJIT [44], work on one object allocation at a time. An object is considered to be non-escaping if it trivially stays within the allocating method, i.e., there is no point at which the object escapes into a field of another object, a global variable, a return value or a parameter to a method call. The effectiveness of such basic algorithms depends heavily on the structure of the code: In some cases, the application's code exhibits certain traits that benefit Escape Analysis, which are encouraged either by convention or by the programming language itself. In other cases, the compiler optimizations preceding Escape Analysis transform the code into a form better fit for Escape Analysis.

Working on one object allocation at a time cannot process complex cases, in which multiple objects depend on each other, or cases in which allocated objects are inputs to phi functions. More sophisticated Escape Analysis algorithms, such as *Equi-Escape Sets* introduced by Kotzmann and Mössenböck [32], therefore work on more than one object at a time. These algorithms build sets of objects that have the same escape state, with each object initially being in a separate set. By analyzing all operations in the method, the system can merge sets (e.g., when an object in one set is assigned to a field of an object in another set), or mark a set as escaping (e.g., when an object in this set is assigned to a global variable).

4.3.2 Control Flow Sensitivity

Escape Analysis algorithms can also be divided into control-flow insensitive and control-flow sensitive algorithms. Control-flow insensitive algorithms are usually faster, because they create a fixed data structure that describes all object allocations and object accesses within the compilation scope, and derive all decisions from this structure. Control-flow sensitive algorithms maintain such a data structure dynamically as they iterate over the intermediate representation, with special handling for loop and merge constructs.

```
if (...) {
  b.x = c;
} else {
  a.x = b.x;
}
```

Listing 4.4: Example which has different implications for object escapability for flow-sensitive and flow-insensitive analysis.

The example in Listing 4.4 will have different effects on the escapability of the objects `a`, `b` and `c` depending on whether the analysis is flow-sensitive or not:

- Flow-insensitive analysis will connect `a` and `c`: if `a` escapes, so does `c`, because `a.x` might have the value `c`.
- Flow-sensitive analysis will determine that `a.x` will never actually have the value `c`, and thus no such connection needs to be made.

4.3.3 Code Representation

Escape Analysis and the associated optimizations can be applied at different levels of the compilation chain:

- Some algorithms, such as the ones proposed by Beers et al. [3] and Jin et al. [31], parse the source code or bytecode, so that the input to the actual compiler already includes modifications or annotations. This has the advantage of requiring less or no modifications in the compiler, but will usually duplicate significant parts of the infrastructure for parsing and analyzing code. It also means that inspecting the original code is more complicated, since the modifications are visible during debugging sessions.

- Escape Analysis can also be performed immediately before the parsing step within the compiler, so that the optimizations facilitated by the analysis will be applied during bytecode parsing. This will make the parsing step significantly more complicated, but does not require information only necessary for Escape Analysis to be maintained after parsing.

- Escape Analysis can be performed on the compiler's intermediate representation, as done by, e.g., Kotzmann and Mössenböck [32]. For this, the intermediate representation needs to contain all information necessary for Escape Analysis. For example, allocations cannot be immediately decomposed into lower-level operations in a way that would prevent Escape Analysis from recognizing them.

- Escape Analysis could also be performed multiple times at different stages of the compiler. Escape Analysis has non-trivial interactions with other optimizations such as loop unrolling, so it may be beneficial to apply it multiple times. This requires all optimizations to maintain the intermediate representation in a state suitable for Escape Analysis.

4.3.4 Intraprocedural vs. Interprocedural Escape Analysis

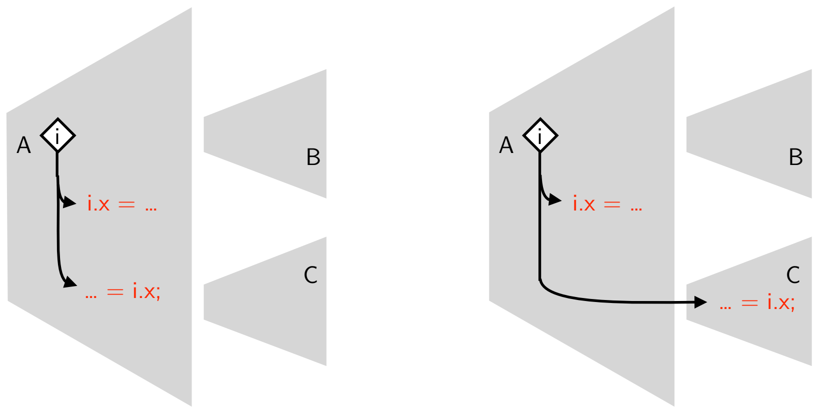


Figure 4.1: Examples for situations handled by Intra- and Interprocedural Analysis.

Intraprocedural Escape Analysis looks at all operations on a newly allocated object within one compilation scope. Any operation that lets the object escape causes Escape Analysis for the object to fail. Such a failure prevents optimizations on this object that can only be performed on non-escaping objects, in the whole method. An example of a situation handled by Intraprocedural Escape Analysis is shown on the left of Figure 4.1. None of the operations performed on the object i lets it escape, so that the whole state of the object is known at all times within compilation scope A.

Interprocedural Escape Analysis tries to statically determine if an object can escape into global state via subsequently called methods. It does so by recursively determining all methods that could possibly receive the object as an argument. If none of these methods perform an operation that causes the object to escape into global state, then the lifetime of the object is known to limited to the current compilation scope. On the right of Figure 4.1, the object `i` has a limited lifetime even though it is passed as an argument to compilation scope B and used by a read operation therein.

Interprocedural Escape Analysis can prove the non-escapability of objects in more cases than Intraprocedural Escape Analysis. However, it still requires objects to not escape in any branch of the application, so that it fails even if the object only escapes in an unlikely path of the compiled code. Also, it requires the object to be allocated at least on the stack, so that Scalar Replacement is not possible.

Generally, Interprocedural Escape Analysis is hard to implement in dynamic and complex systems such as a Java Virtual Machine, because dynamic binding makes it hard to efficiently determine a set of reachable methods without being too conservative.

Chapter 5

Partial Escape Analysis

This chapter introduces the new concept of Partial Escape Analysis in detail, starting with an example that motivates the new algorithm. It explains the state that is kept and updated during the iteration over the compiler graph and how the operations in the graph influence this state. Before ending with details about how Partial Escape Analysis is tied into the Graal system, the handling of control-flow merges and loops is presented.

5.1 Example

In many cases, making a global decision about the escapability of objects does not allow the compiler to perform optimizations, because most objects escape at least in some branches. For example, the object allocated in Listing 5.1 escapes into the global variable cacheKey, so that Escape Analysis would consider it to be escaping.

```
Object getValue(int idx, Object ref) {
  Key key = new Key(idx, ref);
  if (key.equals(cacheKey)) {
    return cacheValue;
  } else {
    cacheKey = key;
    cacheValue = createValue(...);
    return cacheValue;
  }
}
```

Listing 5.1: Motivating example for Partial Escape Analysis.

However, if we only consider the path through the true branch of the if statement, the object does not escape. Analyzing the escapability of objects for individual branches is called *Partial Escape Analysis*. Partial Escape Analysis iterates over the code and maintains the current escape state and the current contents of allocated objects during this process. Initially, each allocated object is in the state virtual, which means that there was no reason yet to actually allocate it. As the algorithm progresses along the control flow, it updates this state when instructions operate on the allocated object.

```
 1 Object getValue(int idx, Object ref) {
 2   Key key = alloc Key; // pseudocode
 3   key.idx = idx;
 4   key.ref = ref;
 5   Key tempKey = cacheKey;
 6   boolean temp;
 7   synchronized (key) {
 8     temp = key.idx == tempKey.idx && key.ref == tempKey.ref;
 9   }
10   if (temp) {
11     return cacheValue;
12   } else {
13     cacheKey = key;
14     cacheValue = createValue(...);
15     return cacheValue;
16   }
17 }
```

Listing 5.2: Example from Listing 5.1 after inlining.

The transition from Listing 5.2 to Listing 5.3 shows how Partial Escape Analysis lets the compiler optimize the code in the example:

- The allocation in line 2 is removed, and an entry for this object is created that specifies that it is virtual and that all fields have their default values.
- The assignments to the fields idx and ref of the virtual object in lines 3 and 4 are removed, and their effects are remembered by updating the object's field states.
- When entering the synchronized region in line 7, the object is still virtual. The monitor enter operation is removed, and the object's state is augmented with a locked flag that specifies that this object would have been locked if it actually existed at this point.
- The accesses to the idx and ref fields of the virtual object in line 8 can be replaced using the object's current field states.
- When exiting the synchronized region in line 9, the object is still virtual. Thus, the monitor exit operation is removed, and the locked flag is removed from the object's state.

- At the if statement in line 10, a copy of the current state is created, because it has to be propagated to both successors of this *control split*.
- When continuing at line 11, the object is still virtual, and the return statement ends the processing of this branch.
- When continuing at line 13, the object is still virtual, but the assignment to the static field cacheKey lets the object escape. In order for it to escape, it needs to exist, and therefore the object needs to be created and initialized with the current state of its fields at this point. This process is called *materialization* in our system.
- The object is transitioned to the state escaped at this point, and the state of its fields cannot be used from here on since there could be assignments to the fields from outside the compilation scope.
- Lines 14 and 15 do not affect the state of the object anymore.

```
Object getValue(int idx, Object ref) {
  Key tempKey = cacheKey;
  if (idx == tempKey.idx && ref == tempKey.ref) {
    return cacheValue;
  } else {
    Key key = alloc Key; // pseudocode
    key.idx = idx;
    key.ref = ref;
    cacheKey = key;
    cacheValue = createValue(...);
    return cacheValue;
  }
}
```

Listing 5.3: Example from Listing 5.2 after Partial Escape Analysis.

In effect, the allocation was moved into one branch of the if statement. While this did not lead to fewer allocation sites in the resulting code, it reduces the dynamic number of allocations at runtime. The actual reduction depends on the likelihood of the branch containing the allocation being reached, but there will never be more dynamic allocations than in the original code.

5.2 Partial Escape Anlaysis in Graal

Partial Escape Analysis is particularly effective if it can interact with other parts of the compiler, such as inlining, global value numbering, and constant folding. In order to do so, it needs to work on the same internal program representation as other optimizations, which, in case of Graal, is the high-level Graal IR [17] (see Section 3.2).

Figure 5.1 shows the Graal IR for the example in Listing 5.2 (after inlining). As the Graal IR is in SSA form, there are no more variables, and the local variable temp is expressed using a phi function. The result of the phi function is true if the conditions of both if statements are true (and therefore, the left branches of the If nodes are taken), and false otherwise.

Graal's Partial Escape Analysis starts iterating the IR graph at the Start node, and processes each node as soon as all its control flow predecessors have been processed. This means that it will follow the control flow, branch at control splits, and process Merge nodes as soon as all predecessors have been visited. Iteration stops at *control sinks* such as Return and Throw nodes.

During this iteration the system maintains a state that keeps track of previously encountered object allocations. For each node that is visited, the system takes the predecessor state and updates it by any effects of the current node. Merge nodes and loop entries are special in that there are multiple predecessor states (from merged branches and loop back edges), which need to be merged into one consistent state before processing the node.

If there was no reason yet to actually create (*materialize*) an allocated object, it is considered to be virtual. In this case the state of all fields and the number of held locks is known. When a previously virtual object needs to be created in the heap, an actual allocation needs to be inserted, which is considered to be the *materialized value*.

Figure 5.2 shows the result of performing Partial Escape Analysis on the graph in Figure 5.1. The allocation of the Key object was removed, along with the initializing field stores and the subsequent loads from these fields. The monitor operations were removed, and in the branch that actually needs the Key object to exist, a Materialize node was added. Finally, control flow optimizations coalesce the if with the subsequent merge, the result of which is shown in Figure 5.3. The compiler recognized that the phi function and the If node that depends on it is not necessary, because it can connect the predecessors of the phi's Merge node with the correct successors of the If node.

5.2.1 Blocks, Scheduling and Reverse Postorder Iteration

Graal IR contains both nodes that are fixed in control flow and nodes that are "floating", i.e., only constrained by their data flow dependencies. The fixed nodes define a basic control flow structure for the graph, into which all floating nodes can be inserted (*scheduled*) at positions that fulfill their dataflow dependencies.

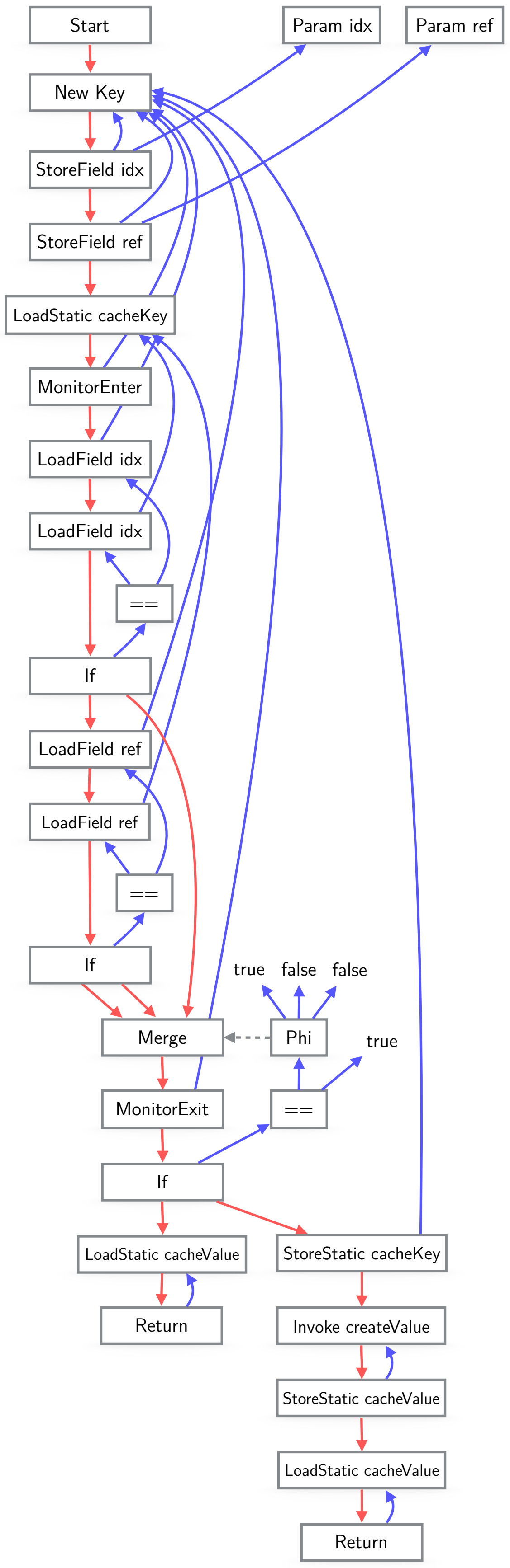


Figure 5.1: Graal IR of the example in Listing 5.2 (after inlining).

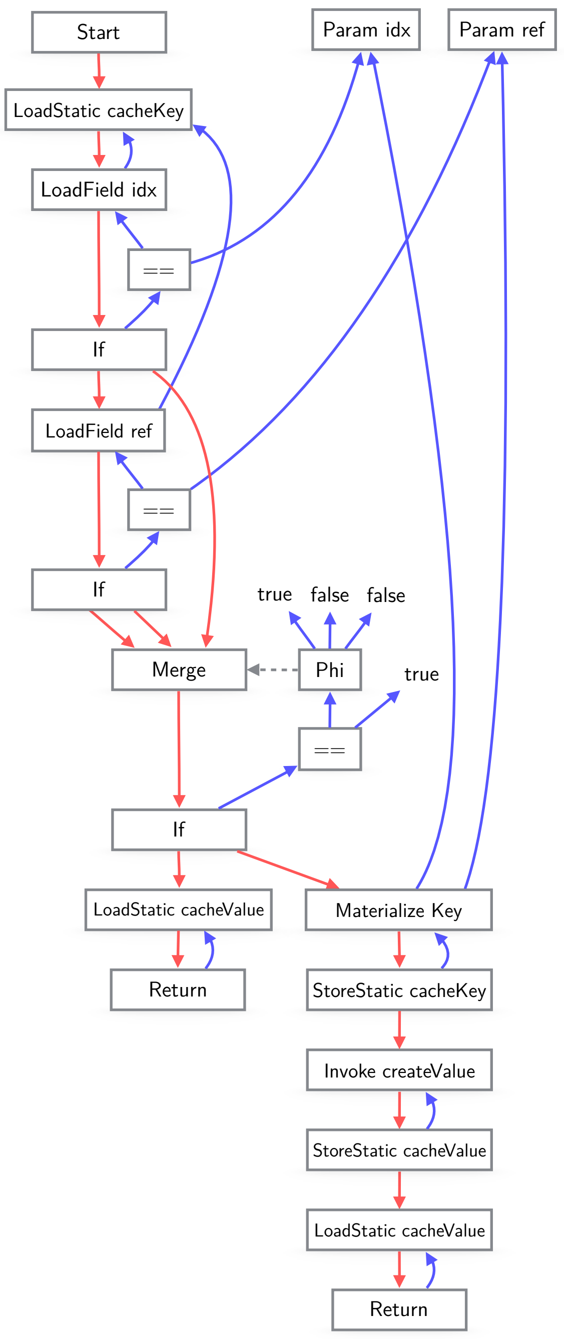


Figure 5.2: Graal IR of the example in Listing 5.2 (after Partial Escape Analysis).

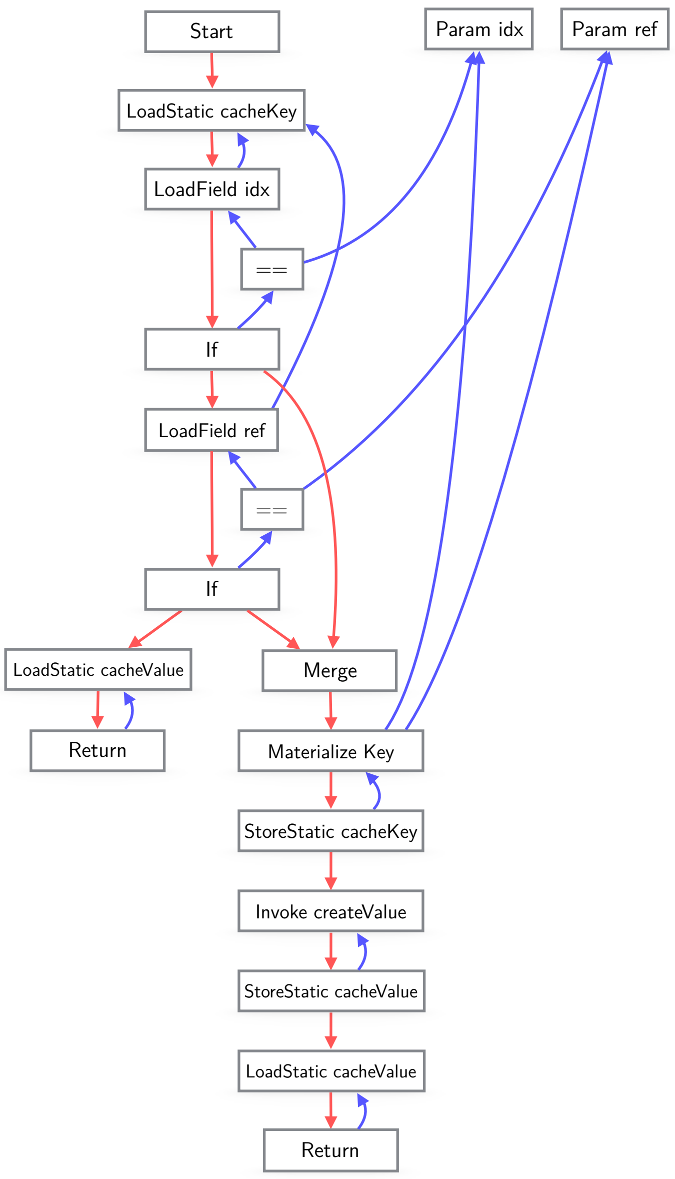


Figure 5.3: Graal IR of the example in Listing 5.2 (after control flow optimizations).

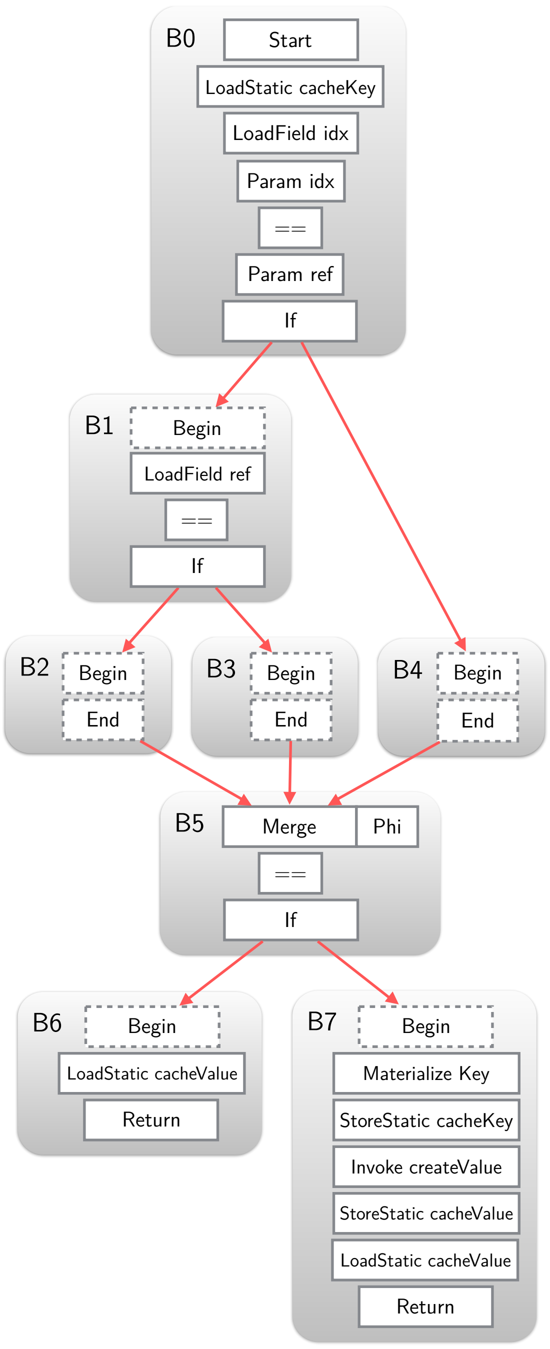


Figure 5.4: Block structure of the Graal IR in Figure 5.2. The dashed boxes represent begin and end nodes that were omitted in Figures 5.1 to 5.3.

In order to gain a higher-level view of a method's control flow, Graal splits the control flow into blocks, also called basic blocks. These blocks adhere to a predefined structure:

- Blocks only contain straight-line control flow, so the only entry into a block is its first node, and the only exit from a block, apart from implicit exceptions and guards, is its last node.

- The first node within a block is either a Start node, a Merge node or a Begin node. It is important to note that the LoopBegin node class derives from the Merge node class and is therefore considered to be a merge node.
- The last node in a block is either a ControlSink[1], a ControlSplit[2] or an End node.

Figure 5.4 visualizes the block structure of the previous example in Figure 5.3. It also includes begin and end nodes that were omitted in the previous figure for the sake of brevity; these nodes are drawn with a dashed outline. They will again be omitted in the rest of this thesis because they exist mainly to ease recognition of basic blocks and are of no consequence to the algorithms presented in this thesis.

Partial Escape Analysis traverses blocks in reverse postorder, i.e., every block is visited as soon as all its predecessor blocks have been visited. For the sake of this iteration, floating nodes are placed into blocks by the so-called *Scheduler*. The Scheduler also creates an ordering of both floating and fixed nodes inside the blocks.

5.2.2 Allocation State

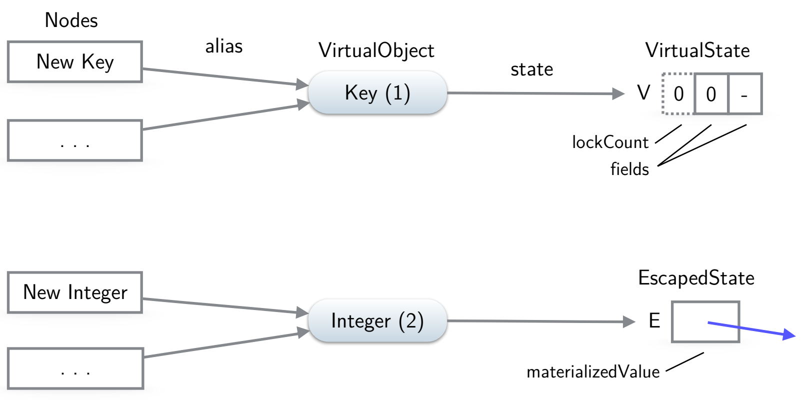


Figure 5.5: Visualization of the allocation state used in the rest of this thesis.

Figure 5.5 shows a visualization of the allocation state maintained during the control-flow iteration. Each object allocation encountered in the original code is represented by a VirtualObject node. For each of these VirtualObject nodes there is an ObjectState describing the current knowledge about this allocation. If the allocation is still virtual, the state is a VirtualState representing the field values and the lock count. If the allocation escaped, the state is an EscapedState containing the materialized value. Finally, aliases provide a mapping from Graal IR nodes to VirtualObject nodes. It will initially

[1]Return nodes, Deoptimize nodes and Unwind nodes are common control sinks.
[2]If nodes and Switch nodes are examples for control splits.

map from the New node of the original program to the allocation's VirtualObject node, but during further analysis more aliases for the same allocation might be added.

```
class VirtualObjectNode extends Node {
  Type type;
}
class ObjectState {
}
class VirtualState extends ObjectState {
  int lockCount;
  Node[] entries;
}
class EscapedState extends ObjectState {
  Node materializedValue;
}
class State {
  Map<VirtualObjectNode, ObjectState> state;
  Map<Node, VirtualObjectNode> alias;
}
```

Listing 5.4: The state that is propagated through the IR.

Listing 5.4 contains a simplified version of the data structures used to store the state described in the last paragraph. State contains the the information that will be updated continuously with new aliases and states during iteration.

5.2.3 Effects of Nodes on the Allocation State

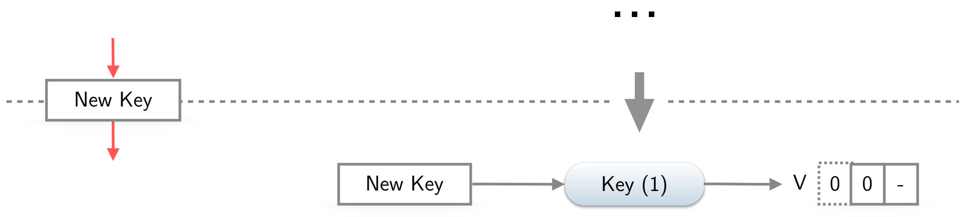
While iterating over the control flow, Partial Escape Analysis looks for operations that have an effect on the allocation state. There are three categories of nodes that require some action:

- Allocations create new virtual objects, therefore they always modify the state by adding new elements.
- If any of the inputs of a node is a key in the aliases map, then the node needs to be examined.
- Merge nodes and LoopBegin nodes (loop headers) merge multiple states.

5.2.3.1 Object Creation and Modification

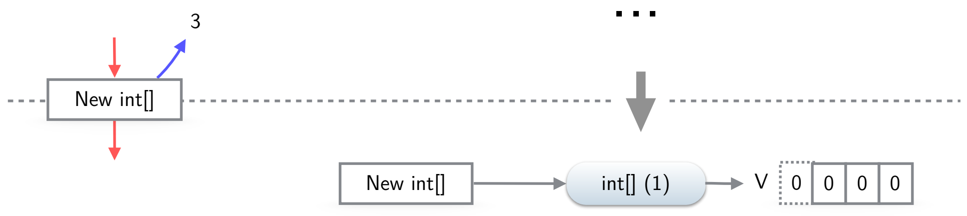
The following patterns introduce new virtual objects and change existing virtual objects. If all effects of an operation can be encoded in the virtual state the operation will be removed from the intermediate representation.

Instance Allocation:



For each instance allocation, a new VirtualObject node and a new VirtualState object is created, and the VirtualState object is initialized with default values. Also, new entries in the aliases and states maps are created that point from the allocation to the VirtualObject node and from the VirtualObject node to the VirtualState.

Array Allocation:

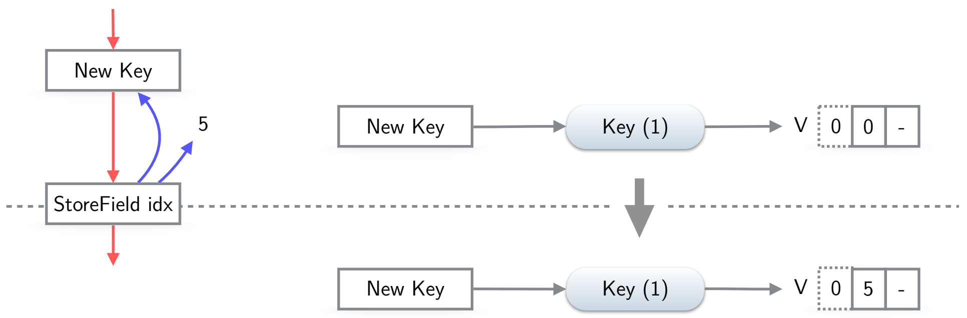


Only array allocations with a statically known size can be processed by Partial Escape Analysis. Similar to instance allocations, new VirtualObject nodes and new VirtualState objects are created and entered into the states and aliases maps.

The number of entries in the VirtualState corresponds to the array size. In order to limit the overhead introduced by Partial Escape Analysis, only arrays up to a configurable maximum size, which defaults to 32, are processed[3]. Arrays above this limit are treated as if their size was non-constant, i.e., no VirtualObject node will be introduced for them.

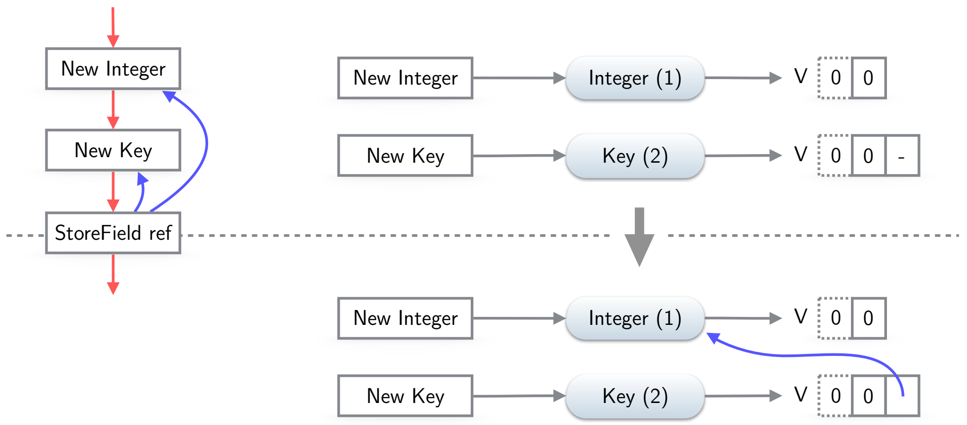
[3]In general, this limit should correspond to the maximum number of iterations with which a loop will be unrolled. Loops working on arrays of a larger size will not be unrolled, so that the array indexes are not constant, which will cause the array to be materialized anyway.

Field Store (non-virtual value):



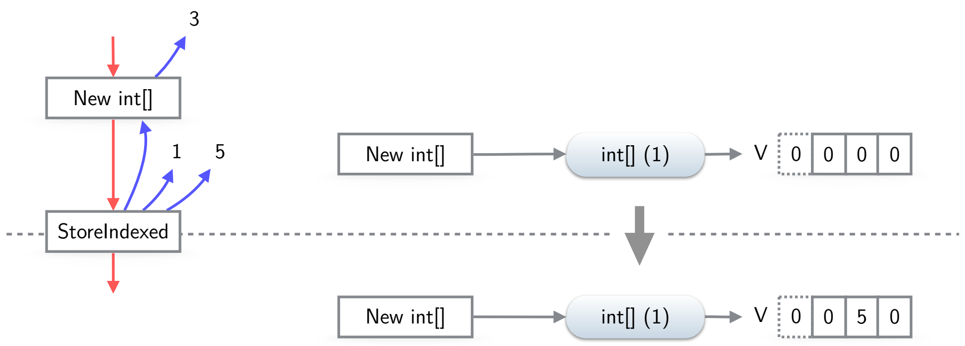
Storing the value generated by a node that is not aliased with any virtual object into a field of a virtual object sets the field value in the corresponding VirtualState object. While the example stores the value generated by the Constant node “5”, the value can also be a non-constant value generated by some other node.

Field Store (virtual value):



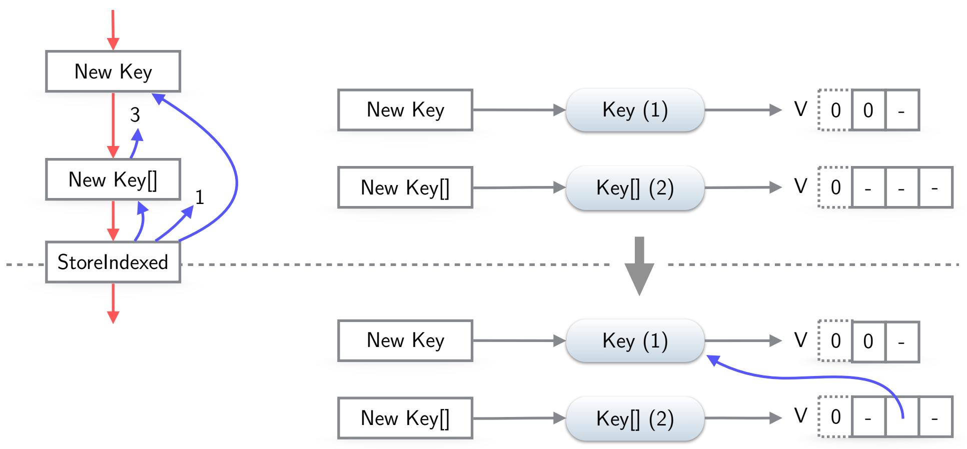
Storing the value generated by a node that is aliased with a virtual object into a field of another virtual object puts a reference to the VirtualObject node of the stored object into the field value in the target virtual object.

Array Store (non-virtual value):



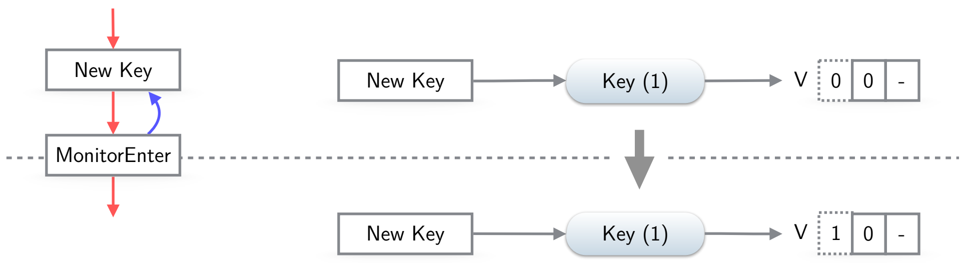
Array stores to virtual objects can only be processed by Partial Escape Analysis if the index is constant and within the bounds of the array. Similar to storing a non-virtual value into an instance, the value is put into the array elements in the array's VirtualState.

Array Store (virtual value):



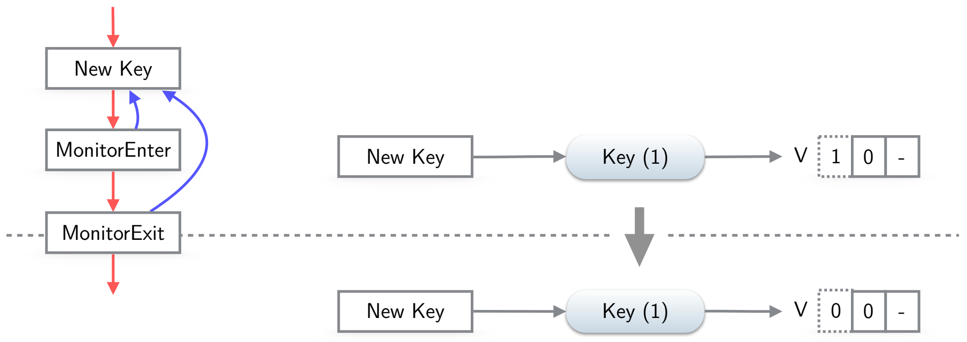
Similar to storing a virtual value into an instance, a reference to the value's VirtualObject node is stored into the array elements in the array's VirtualState.

Acquiring a Lock:



Entering a synchronized region (via a MonitorEnter node) with the locked object being a virtual object increments the lockCount.

Releasing a Lock:

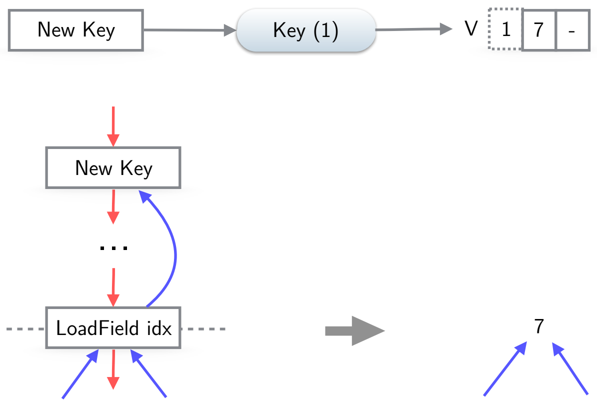


Exiting the synchronized region decrements the lockCount. The lockCount cannot drop below zero, because this would be an invalid pairing of monitor operations that will be rejected during bytecode parsing.

5.2.3.2 Accessing Object Properties

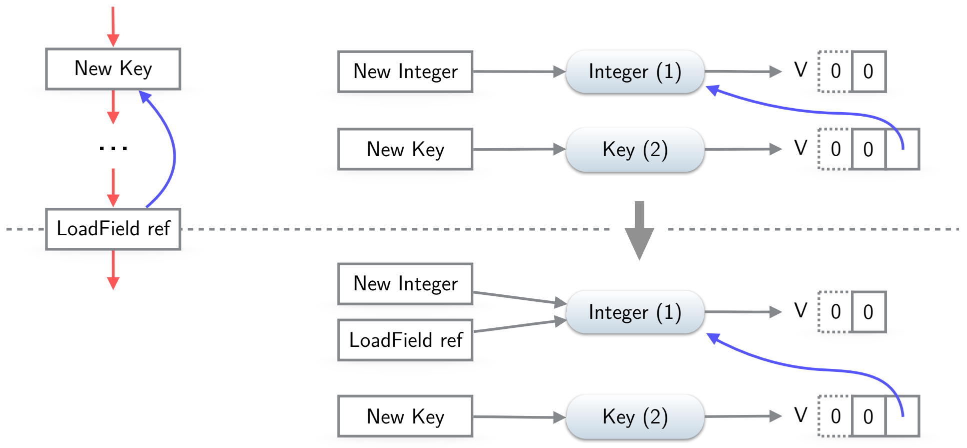
The following patterns query information about existing virtual objects. If an access can be satisfied from the information in the VirtualState, the node that represents this access will be removed from the intermediate representation and will be replaced with this information.

Field Load (non-virtual value):



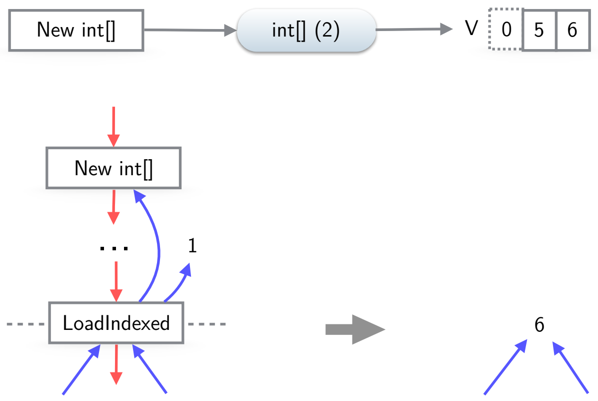
Loading a (non-virtual) value from a field of a virtual object replaces the LoadField node with the value from the corresponding field of the VirtualState at all its usages. While the example replaces the LoadField node with the Constant node "7", the replacement can also be some other, non-constant, node.

Field Load (virtual value):



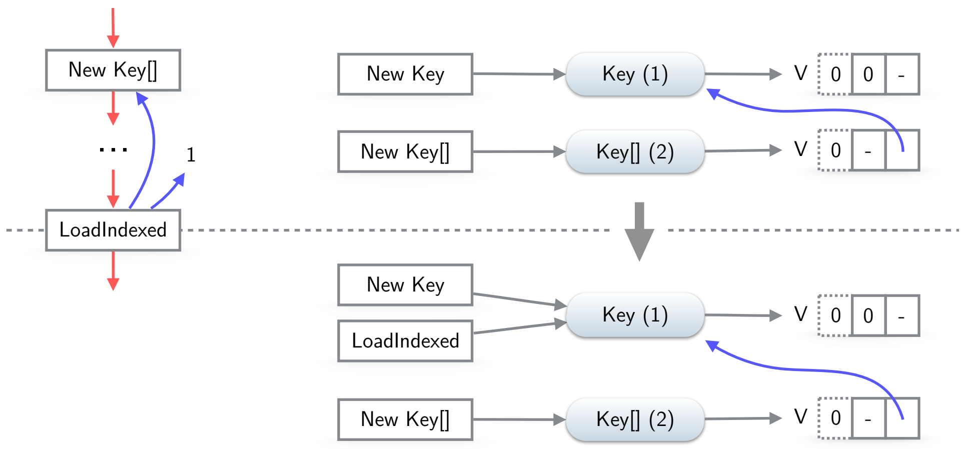
Loading a virtual object X from a field of some other virtual object makes the LoadField node a new alias of X. In our example, the LoadField node can thus be recognized as referring to the virtual Integer during further processing.

Array Load (non-virtual value):



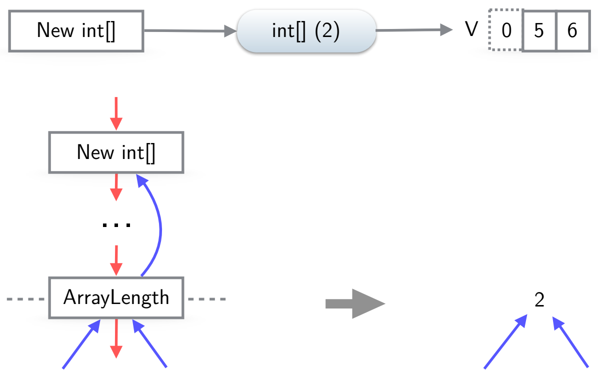
If the index is constant, an array load of a non-virtual value can be processed similar to a field load: the array load is replaced with the value from the corresponding array element from the array's VirtualState.

Array Load (virtual value):



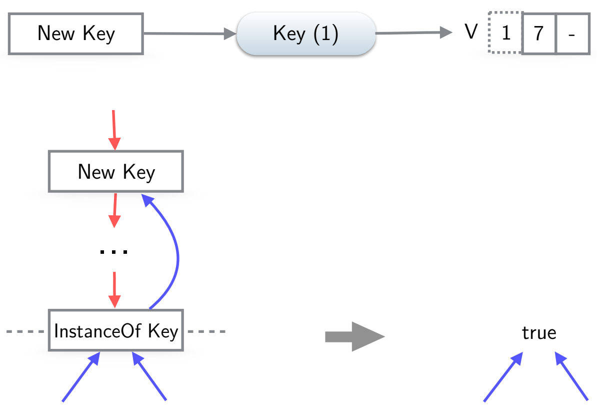
Similar to loading a virtual object from an instance, the LoadIndexed node becomes a new alias of the virtual object.

Array Length:



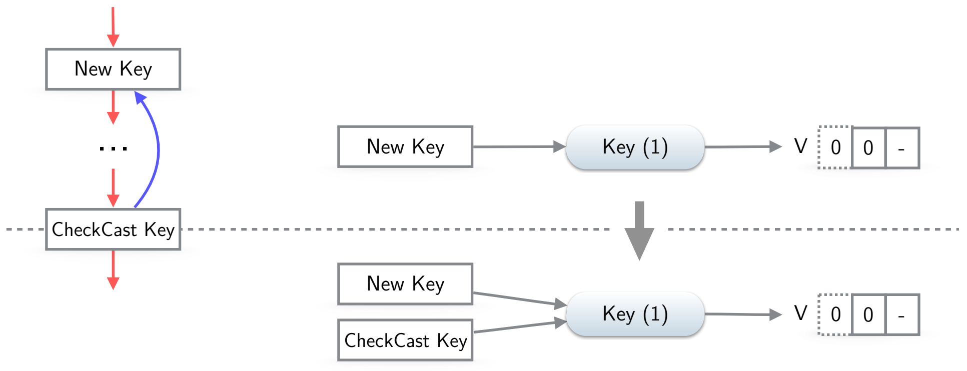
The ArrayLength node can simply be replaced with the constant array length taken from the VirtualState.

Type Check:



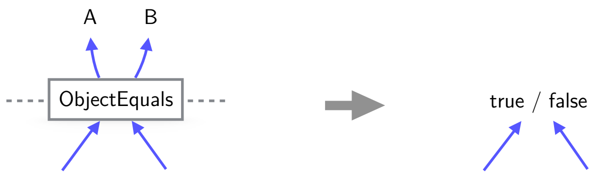
Since the exact type of a virtual object is known, the type check can be executed at compile time, and the InstanceOf node is replaced with the result.

Type Cast:



For virtual objects, type casts can also be executed at compile time. If the virtual object's type is equal to or a subtype of the target type, the CheckCast node will be added as an alias for the virtual object. Otherwise, the type cast will be replaced with a deoptimization, since it will always fail and therefore deoptimize at runtime.

Equality Comparison:



If either "A" or "B" is virtual, but the other is not, the comparison can be replaced with false, since a virtual object cannot be aliased with a preexisting value. If both "A" and "B" are virtual, the comparison can be replaced with true if both are aliased with the same VirtualObject node, and false otherwise.

5.2.3.3 Generic Case

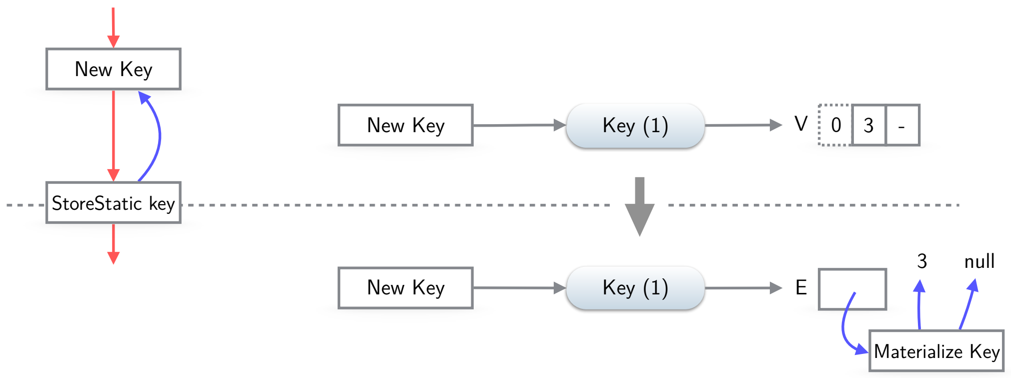


Figure 5.6: Storing a virtual value into a static field.

Any operation that does not fulfill the conditions for virtualization, e.g., a LoadIndexed with a non-constant index, or that was not explicitly described here is assumed to require an actual object reference. Therefore, any virtual object that is referenced from such an operation will be materialized, and the input that maps to the VirtualObject node of the now escaped object is replaced with the materialized value.

Figure 5.6 contains an example of this situation. Storing the virtual Key object to the static field key causes it to be materialized. The Materialize node will be inserted immediately before the StoreStatic node, and the store will reference this node instead of the original New. Also, the object's state changes to an EscapedState that references the materialized value.

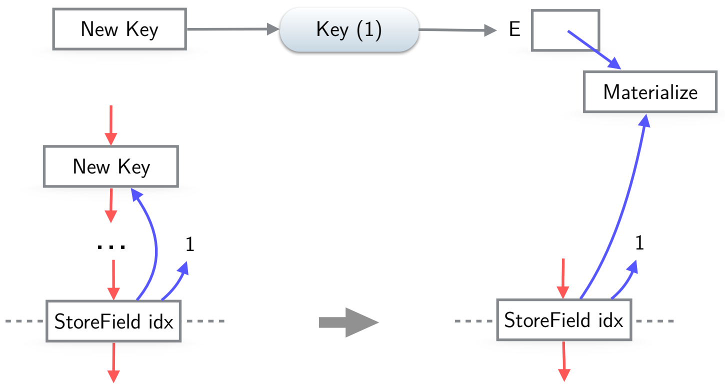


Figure 5.7: Store operation performed on an escaped object.

Figure 5.7 shows an example of a Store operation where the input is an escaped object. In general, inputs that refer to escaped objects are handled as if they were normal values, but they are replaced with the materializedValue during processing.

```
Key key = new Key();
if (...) {
  key.idx = 5;
} else {
  key.idx = 6;
}
cacheKey = key;
```

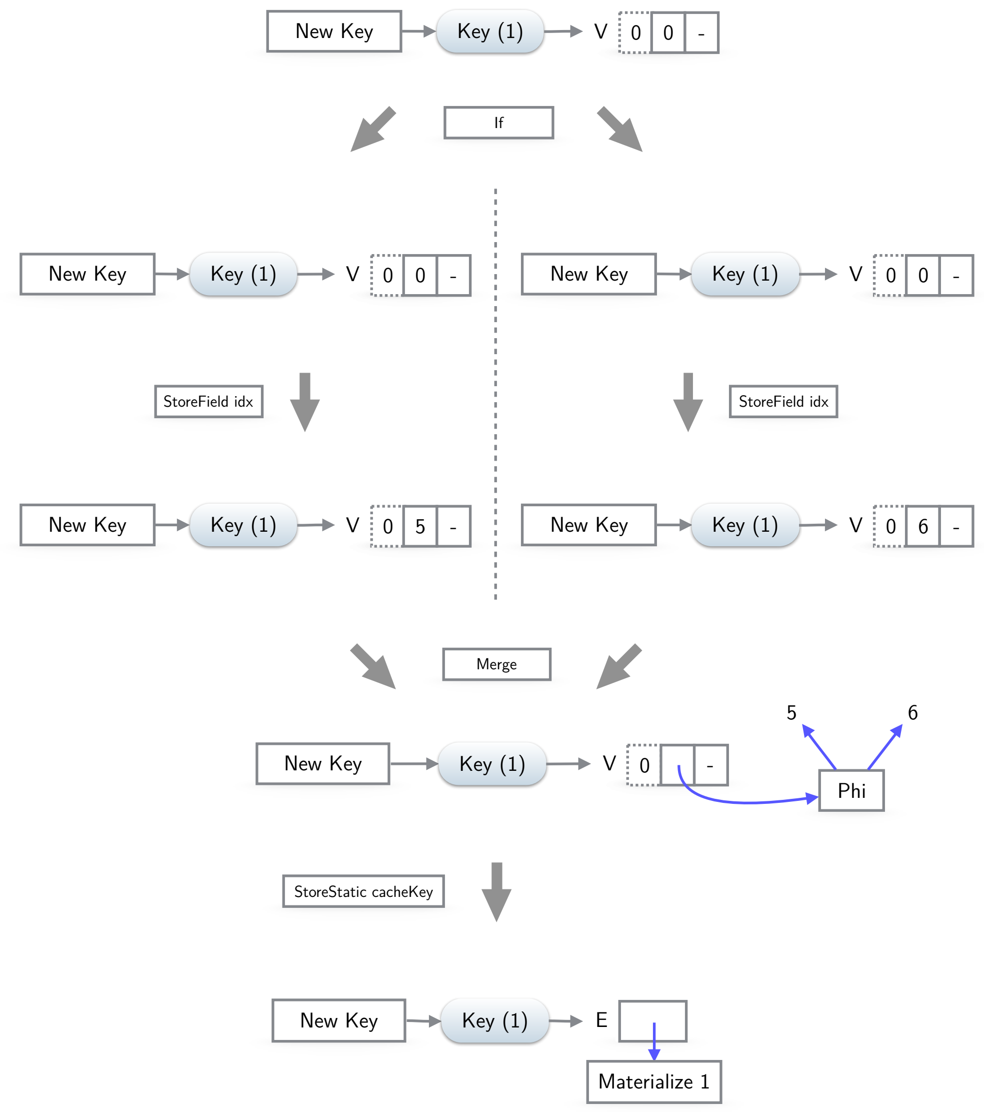


Figure 5.8: Splitting and merging of the allocation state for a small example.

5.2.4 Control Flow Splits and Merges

During control flow iteration, the allocation state is updated continuously with the effects of the nodes that are encountered. At control flow splits, e.g., if and switch statements, a copy of the state is created for each successor. At control flow merges multiple states need to be merged into a single one.

Figure 5.8 shows this splitting and merging of states for a small example that creates a Key object, assigns either 5 or 6 to it and lets it escape into the static cacheKey field. The state, which contains the virtual object created by the allocation, is split into two independent states at the if statement. These states will subsequently be modified by assigning the values 5 and 6 to the idx field. When merging the two states, a phi function is created for the value of the idx field. Finally, assigning the virtual object to the static field cacheKey causes the object to be materialized, so that the state is changed to an EscapedState.

Whenever multiple branches meet at a Merge or LoopBegin node, there are also multiple states that need to be merged into one consistent state. A so-called MergeProcessor is responsible for doing that, as shown in Figures 5.9 to 5.12.

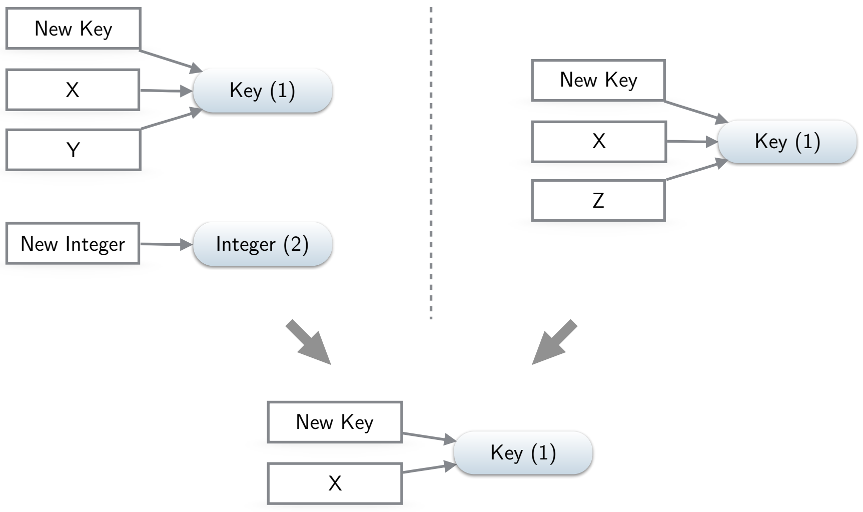


Figure 5.9: Merging of aliases performed by the MergeProcessor.

The MergeProcessor first creates the intersection of the aliases maps of all merged states, which implies that only VirtualObject nodes that exist in all predecessor states and have at least one common alias will survive the merge (Figure 5.9).

5.2.4.1 Merging of States

For each VirtualObject node, the MergeProcessor looks at the VirtualObject node's ObjectState in all predecessor states:

- If the VirtualObject node escaped in all predecessors states, the merged state of the VirtualObject node is an EscapedState whose materializedValue points to a newly created Phi function that merges the materializedValues of the predecessor states.

  Figure 5.10 contains an example in which two escaped states for the Key object are merged. The resulting state references a newly created Phi.

- If a VirtualObject node is in the virtual state in some predecessors and in the escaped state in others, then all virtual states need to be materialized at the corresponding predecessor in the control flow, and processing continues like in the previous case.

- If a VirtualObject node is in the virtual state in all predecessors, then the new state of the VirtualObject node will also be virtual, and all field values need to be merged. For each field, the MergeProcessor looks at the value of this field in all predecessor VirtualObjects:

  – If all field values are identical, this value will be the value of the field in the new VirtualState. Note that this applies to VirtualObject nodes that represent allocations as well: if all predecessor VirtualStates reference the same VirtualObject node, then so does the new one.

  – If some field values differ, the MergeProcessor creates a new Phi node for this field. If a field that should be merged references a virtual object (i.e., a VirtualObject node with a VirtualState), this object needs to be materialized before merging.

  Figure 5.11 shows an example in which virtual states are merged: A new Phi is created for the mismatching values 5 and 6, while the value 1 is the same in both merged states. The reference from the state of the second virtual object to the first one also remains the same in the resulting state.

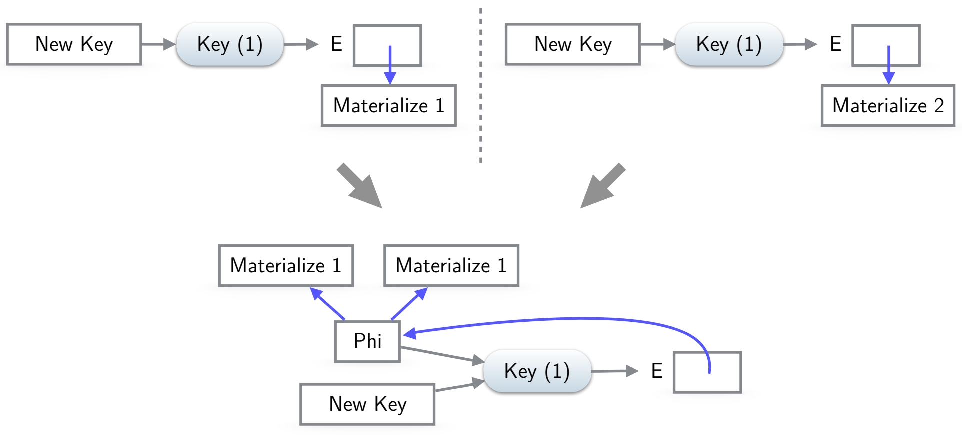


Figure 5.10: Merging of escaped objects performed by the MergeProcessor.

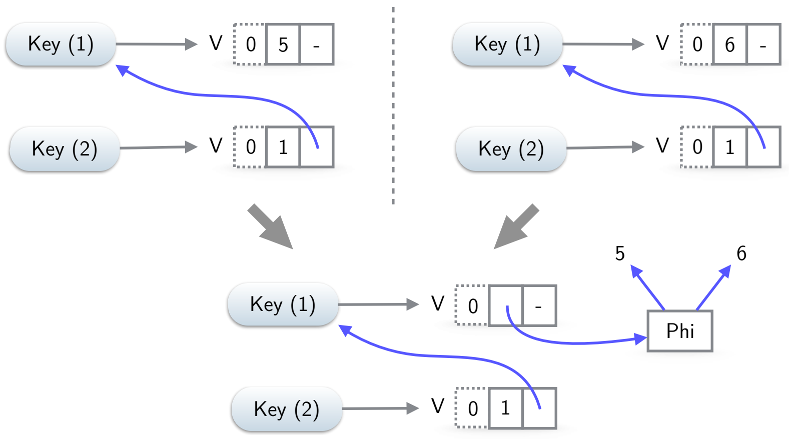


Figure 5.11: Merging of virtual objects performed by the MergeProcessor.

5.2.4.2 Merging of Phis

Values of the same variable that flow together at a merge are combined by Phi nodes attached to the Merge node. The MergeProcessor has to check whether the inputs of these Phi nodes are aliased with VirtualObject nodes:

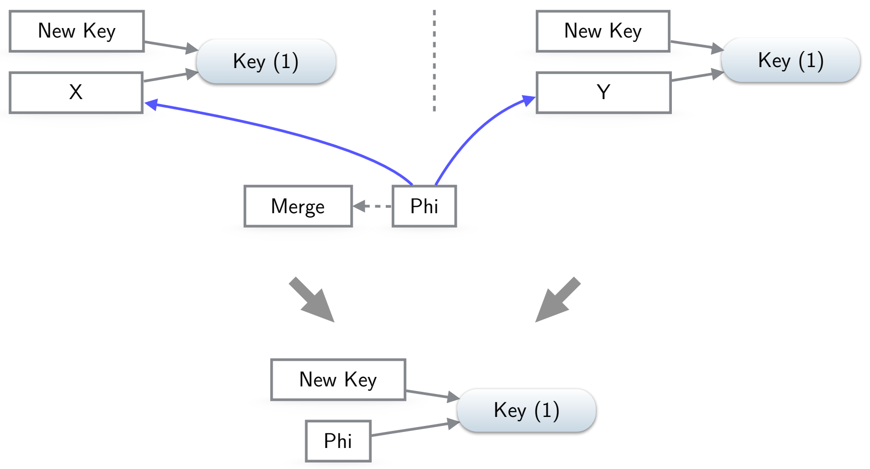


Figure 5.12: Merging of aliases for Phi nodes performed by the MergeProcessor.

- If all inputs are aliased to the same VirtualObject node , the Phi node will be added as an alias of the VirtualObject node in the merged state, as shown in Figure 5.12.
- Otherwise, any input that is aliased with a virtual object needs to be materialized, and the input in the Phi is replaced with the materialized value.
- If an input is aliased with an escaped object, the input in the Phi is replaced with the materializedValue.

During this process of merging states some virtual objects might be turned into escaped objects, which can invalidate previous assumptions about which objects are virtual. The merge process is therefore iterated until no more materializations happen during merging, at which point a stable state has been reached.

5.2.5 Loops

Loops are special in that the iteration algorithm needs to start traversing the loop's contents before its back edges are processed. Graal's Partial Escape Analysis solves this by processing loops iteratively. During the first iteration, the loop body is processed starting with a *speculative* state, which is taken from the loop's predecessor. Iteration will stop at the loop's back edges and at loop exits. As soon as the loop body has been processed, and the states at all back edges are available, the MergeProcessor is used to merge the states of the loop's predecessor and the loop back edges.

The state produced by the MergeProcessor is only valid if the speculative start state is correct. Therefore, the new state is compared to the speculative state. If they differ, the new state is used as the speculative start state, and the loop is re-processed. Once the state produced by the MergeProcessor equals the speculative state, processing continues at the loop's exits.

This loop processing also applies to nested loops: For each iteration of the outer loop, the inner loop will be processed until it reaches a fixed point.

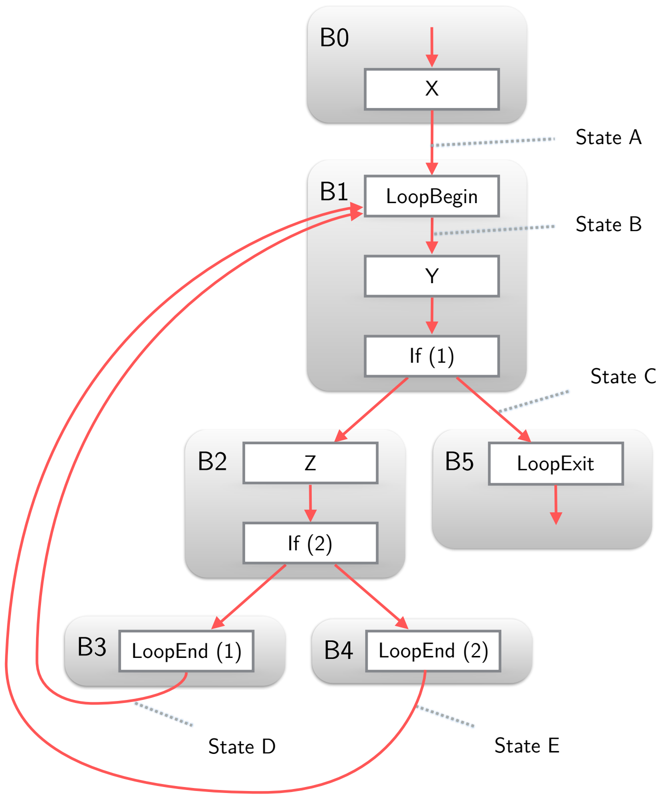


Figure 5.13: Example loop.

Figure 5.13 shows an example of a loop with one exit and two back edges. When the iteration encounters the loop, only state A is known. In order to be able to start processing the loop, it is assumed that B equals A. As the iteration continues processing all other nodes in the loop, it creates the states C, D and E. The MergeProcessor merges A, D and E to create a new state B'. If B' equals B, then C is correct and the iteration can continue at the LoopExit node. If B' does not equal B, then B is replaced with B' and the loop is reprocessed.

5.2.6 Handling of Effects

When a loop is reprocessed, all states generated during the previous iteration are thrown away. Additionally, all effects on the Graal IR, such as added or removed nodes and changed inputs, need to be undone, so that the graph is in the state in which it was at the loop entry.

An obvious solution would be to store the whole graph upon loop entry and restore it if the loop is reprocessed. But this is a prohibitively expensive operation, because graphs can consist of many thousands of nodes.

Graal's Partial Escape Analysis therefore implements this backtracking in a different way: effects on the nodes of the graph are not applied immediately, but they are stored in a list of effects that can be committed at a later point in time. Such a list is maintained for each basic block in the graph. Additionally, the system maintains a map of replaced objects that allows subsequent operations to use this information, e.g., that a LoadField node which is subsequently used as an array index was replaced with a constant value

Once a loop needs to be reprocessed, backtracking to the loop entry is a simple matter of clearing the list of effects for all blocks within the loop. In Figure 5.13, reprocessing the loop would clear the list of effects for the blocks B1, B2, B3 and B4.

The GraphEffectList provides the following types of graph changes:

addFloatingNode adds a given floating node to the graph. Nodes are not connected to a specific graph during construction, and they need to be associated with a graph explicitly.

addFixedNode adds a fixed node into the control flow, above a given position.

setPhiInput sets a specific input in a phi node.

deleteFixedNode removes a fixed node from the control flow. The fixed node cannot have any data flow usages.

replaceAtUsages replaces a node with another node at all its data flow usages.

replaceFirstInput replaces the first input of a given node that is equal to a certain value with another value.

addState adds a VirtualState or EscapedState for a given VirtualObject node to a frame state (see Section 5.2.7).

addMaterialization inserts the materialization of a virtual object into the control flow at a given position. This consists of an allocation, the initialization of all fields and additional code needed to restore the object's locking state.

5.2.6.1 Placement of Merge Effects

Since all effects are stored into lists of effects associated with a block, the MergeProcessor needs to take special care about where to place the effects it generates for a loop merge. It cannot simply place all effects into the loop entry block, because some of them, such as materializations in predecessor states, should not be discarded if the loop is reprocessed. The loop processing would not converge to a fixed state if the materializations were discarded.

If a virtual object in state A in Figure 5.13 is materialized by the MergeProcessor while merging the states A, D and E, e.g., because it is marked as escaped in the state at one of the back edges, the addMaterialization effect needs to be added to block B0.

Several different situations occur during merging:

- If a virtual object is materialized in one of the predecessor states, then the effect that adds the materialization logic is added to the predecessor block that corresponds to predecessor state.
- Newly created phi nodes are added in the loop entry block. They may be the initial value of a virtual object's field, and are therefore required to exist by the time effects are applied in the loop's body.
- The input values of Phi nodes are initialized in the loop exit block. The initialization requires all values to exist, which is only guaranteed once the whole loop's effects has been processed.

5.2.7 Handling Frame States

The HotSpot interpreter cannot work with virtual objects. Therefore, all virtual objects need to be materialized whenever a deoptimization occurs. The information required to create the objects needs to be added to the FrameState nodes that describe the mapping from machine state to Java Virtual Machine state whenever a virtual object is referenced by the frame state.

Figures 5.14 and 5.15 show two Graal IR fragments with frame states, corresponding to Listing 5.5. The FrameState nodes are expressed as dashed boxes; they contain

```
static Object global;
void foo(int x) {
  Integer i = new Integer(x);
  global = null;
  ...
}
```

Listing 5.5: Example shown in Figures 5.14 and 5.15.

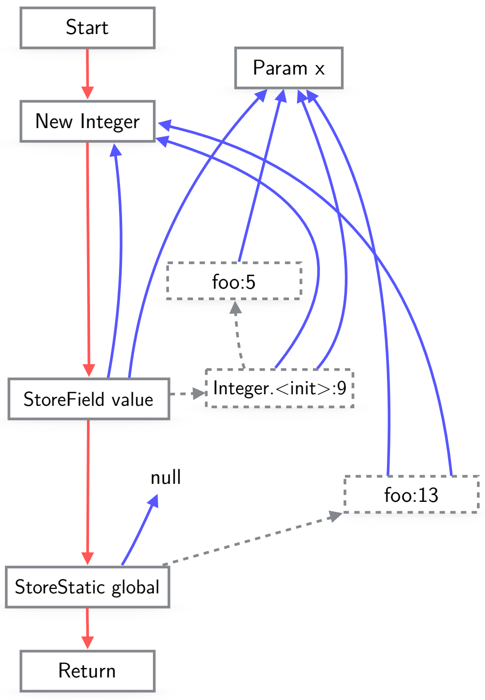


Figure 5.14: Example from Listing 5.5 with FrameStates, after inlining.

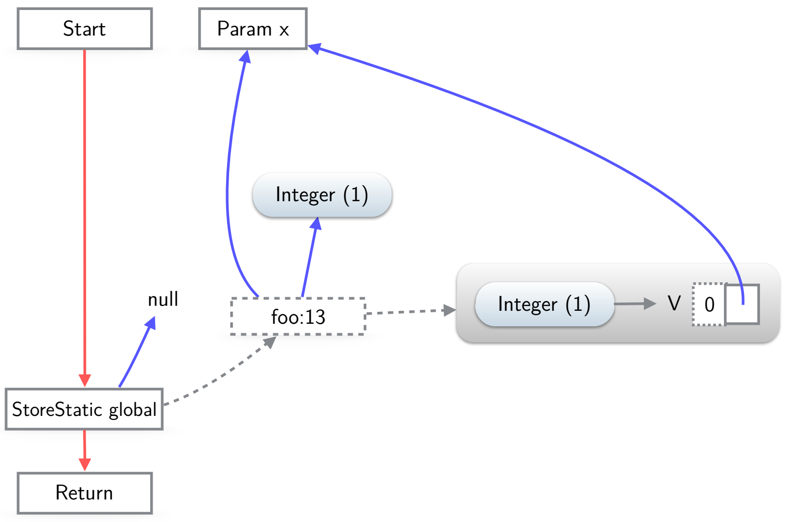


Figure 5.15: Example from Listing 5.5 with FrameStates, after Partial Escape Analysis.

the method name (with “<init>” being a constructor call) and the bytecode position. Their inputs describe the local variables and the contents of the expression stack. Uninitialized local variables, e.g., the variable i in foo:5, are omitted. It is important to note that the expression stacks in this example are empty.

Figure 5.14 contains a field store which is associated with bytecode index 9 in the constructor of Integer. At this point there are two local variables: the newly allocated Integer object and the value x. The constructor was inlined into the method foo, so it has a reference to the outer frame state at bytecode index 5 in foo. This outer frame state has only one initialized local variable, namely the value x.

Figure 5.15 shows the same Graal IR fragment after applying Partial Escape Analysis. The first field store was removed due to Scalar Replacement, which also removed the associated FrameState nodes. The second field store, however, was not removed. Its associated frame state contains a reference to the parameter x and to the virtual object whose allocation was removed. To be able to restore the object during deoptimization, a copy of the current VirtualObject to VirtualState mapping is added to the frame state.

5.2.7.1 Frame States for Inlined Methods

When methods are inlined, every method is represented by its own frame state. The inner frame state of the callee references the outer frame state of the caller. The outer frame states are not duplicated, so that one outer frame state can be referenced from multiple inner frame states. Duplicating outer frame states would be prohibitively expensive in situations with deeply nested inlining levels.

An outer frame state can contain VirtualObject nodes that are also used in the inlined method. Since this outer frame state can referenced from multiple inner frame states with a possibly different virtual/escaped state for the object, the state in the outer frame state depends on the state this object has at the position of the inner frame state. It is not possible to simply replace escaped object by their materialized value, since the fact that the virtual object was materialized is required to correctly interpret the outer frame state. In some situations an outer frame state will reference a VirtualObject node that is not referenced by the inner frame state. The mappings for all VirtualObject nodes referenced by the inner or any outer frame state need to be added to the inner frame state. This is required to be able to correctly restore the objects referenced by all frame states during deoptimization.

Listing 5.6 will be used to demonstrate the representation of nested frame states in combination with virtual and escaped allocations: the method foo allocates an Integer and a Key object, where the first is referenced by the latter. After that, it calls the method bar, which is inlined into foo and which contains three assignments to the static

```
static Object global;
void foo(int idx) {
  Object ref = new Integer(idx);
  Key key = new Key(5, ref);
  bar(key);
}
void bar(Key key) {
  global = null;
  global = key.ref;
  global = key;
}
```

Listing 5.6: Example used to show the representation of virtual and escaped objects in nested frame states.

variable global. The first assignment sets the static variable to null, the second one to the Integer object and the third one to the Key object. This also means that in the (after) frame state of the first assignment, both allocations are virtual, in the second one the Integer is escaped, and in the third one both allocations are escaped.

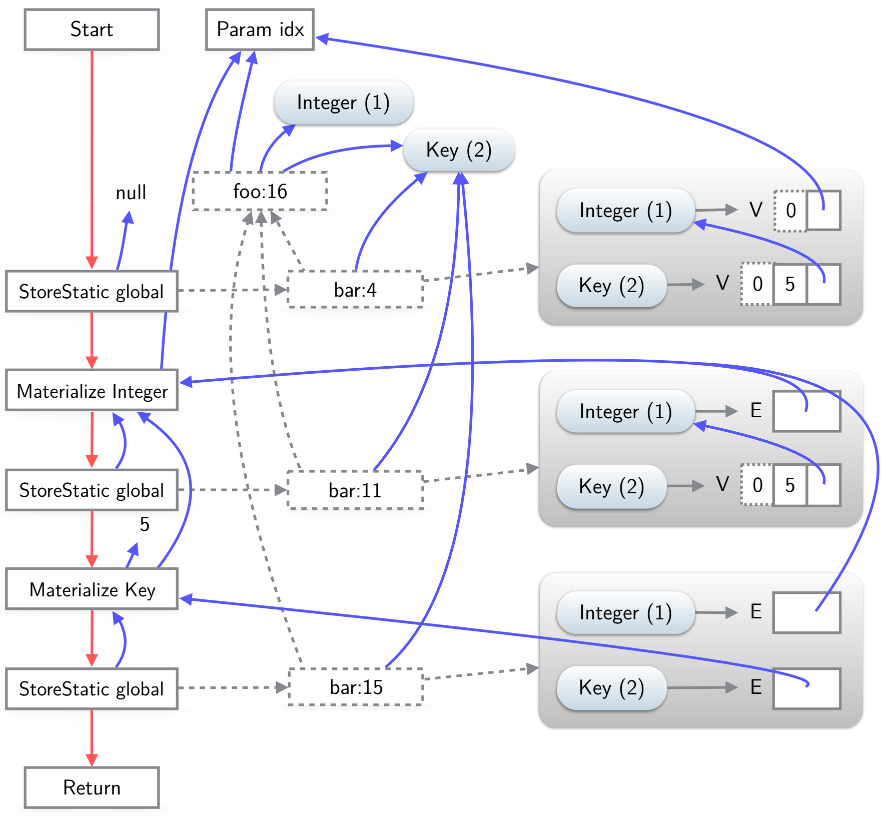


Figure 5.16: Example from Listing 5.6 after Partial Escape Analysis.

Figure 5.16 shows the Graal IR for method foo from Listing 5.6 after inlining and Partial Escape Analysis. The three leaf frame states for method bar reference a single frame state for method foo, with different mappings for the state of the allocations.

Chapter 6

Extensible Escape Analysis

6.1 Extensibility: Node Types

The implementations of Escape Analysis can be quite complex. The HotSpot server compiler, for example, contains 4,000 lines of code in escape.hpp and escape.cpp, and there is a large amount of supporting code scattered over the rest of the compiler.

Much of the server compiler's implementation of Escape Analysis consist of large switch statements dealing with different types of nodes in the compiler's intermediate representation. There is no easy way to extend the list of nodes that are processed by Escape Analysis, so that adding new node classes always incurs a change in escape.cpp. The compiler does not expect to encounter unexpected nodes during Escape Analysis.

Since Graal was designed to be extensible, such a centralized implementation would be very limiting, even if it has a default behavior for unknown node types. Graal therefore delegates the task of describing the effect of a node to the node itself. All node types, including basic types such as LoadField, are handled this way.

6.1.1 Virtualizable Interface

The default behavior of any node type is to cause all virtual inputs to be materialized. In some cases, such as a Return node, no further handling is required, since this is already the correct behavior.

If a node type wants to convey more information about its effects to Partial Escape Analysis, it can implement the Virtualizable interface. Every node that has an input to a node that is aliased with a virtual object will have its virtualize method called. Querying only these nodes limits processing to nodes that can have an influence on the allocation state.

```
public interface Virtualizable {
  void virtualize(VirtualizerTool tool);
}

public interface VirtualizableAllocation extends Virtualizable {
}
```

Listing 6.1: The Virtualizable and VirtualizableAllocation interfaces that can be implemented by node types.

However, nodes that introduce new virtual objects, i.e., allocations, need to be processed even if they do not reference any existing virtual object. By implementing the VirtualizableAllocation interface, which extends the Virtualizable interface, a node type can signal that its virtualize method should be called unconditionally.

The Virtualizable and VirtualizableAllocation interfaces, which are shown in Listing 6.1, contain only the method virtualize. The VirtualizerTool passed along as a parameter can be used by the node to describe its effects.

6.1.2 VirtualizerTool

The VirtualizerTool, shown in Listing 6.2, is the main interface through which a node can communicate with Graal's Partial Escape Analysis. It contains methods to query meta data and compiler configuration, methods to inspect and modify the state of virtual and escaped objects and methods to convey what changes to the current node are required:

- getMetaAccessProvider and getAssumptions provide access to facilities required to deal with types.
- getMaximumEntryCount returns the maximum number of entries an array can have in order to still be considered for Escape Analysis.
- createVirtualObject inserts a new VirtualObject into the allocation state and initializes its entries with the given values.
- getVirtualState queries the aliases map. If the given value maps to a VirtualObject node with a virtual state it returns the VirtualState, and null otherwise.
- getReplacedValue also queries the aliases map. If the given value maps to a VirtualObject node with an escaped state it returns the state's materialized value. If this is not the case it checks if the given node was replaced previously with a

```
public interface VirtualizerTool {
  // methods for querying compiler configuration
  MetaAccessProvider getMetaAccessProvider();
  Assumptions getAssumptions();
  int getMaximumEntryCount();

  // methods working on virtualized/materialized objects
  void createVirtualObject(
      VirtualObjectNode virtualObject, ValueNode[] entryState);
  VirtualState getVirtualState(ValueNode input);
  ValueNode getReplacedValue(ValueNode input);
  void setVirtualEntry(
      VirtualState state, int index, ValueNode value);

  // operations on the current node
  void replaceWithVirtual(VirtualObjectNode replacement);
  void replaceWithValue(ValueNode replacement);
  void replaceWith(ValueNode replacement);
  void delete();
  void addNode(ValueNode node);
}
```

Listing 6.2: The VirtualizerTool interface that is used by nodes to describe their effects.

replaceWithValue call, in which case it returns the replacement. If neither of these two checks succeeds, the passed node is returned[1]. Note that it is invalid to call this method with a node that is aliased with a virtual object.

- setVirtualEntry is used to set an entry in the current state of a virtual object[2]. The value can also be a VirtualObject node.

- replaceWithVirtual replaces the current node, i.e., the receiver of the virtualize call, with the given VirtualObject node, effectively deleting it and inserting an entry into the aliases map.

- replaceWithValue replaces the current node with a (non-virtual) value.

- replaceWith replaces the current node with the given value. If the value maps to a VirtualObject node, replaceWithVirtual is called, otherwise replaceWithValue.

- delete simply deletes the current node.

[1]One might expected this method to return null in this case, but for most use cases returning the original value is more useful.

[2]This method is part of the VirtualizerTool, and not VirtualState, for implementation reasons: It checks whether the given ValueNode was replaced with some other value during a previous virtualize call.

- addNode adds a new node to the graph. If the given node is a fixed node, it will be inserted into the control flow as the immediate predecessor of the current node.

It is important to note that the implementation of virtualize must not perform any modifications to nodes directly. Backtracking during loop processing may discard the current set of changes and reprocess the loop, as explained in Section 5.2.5.

```
class ObjectState {
  VirtualObjectNode getVirtualObject();
}

class VirtualState extends ObjectState {
  ValueNode getEntry(int index);
  void addLock(MonitorIdNode monitorId);
  MonitorIdNode removeLock();
}

class EscapedState extends ObjectState {
  ValueNode getMaterializedValue();
}
```

Listing 6.3: The interface provided by ObjectState and its subclasses.

Listing 6.3 shows the actual interface for virtual and escaped states. Every object state can be queried for its corresponding VirtualObject node. States for virtual objects additionally provide access to the object's entries and locks[3]. States for escaped objects provide the object's materialized value.

Listing 6.4 shows the actual interface for the nodes representing objects. These classes provide metadata about the object, i.e., its type and the number and type of its entries. There are two distinct subclasses for instances and arrays, VirtualInstance and VirtualArray. The instance variant provides additional methods to determine the entry index of a specific field and the field stored at a specific entry index. There are no additional methods in VirtualArray because getType and getEntryCount already provide all necessary information. ResolvedJavaType is a metadata class that represents instance types, array types and primitive types. ResolvedJavaField, which represents an instance field, can be queried for the type and name of the field.

[3]Every lock in Graal is associated with a MonitorId node, therefore a node of this type has to be provided if a lock is added. Also, the correct pairing of lock and unlock operations is checked during bytecode parsing, so that it is never necessary for a virtualize method to query the VirtualState whether a lock is held.

```
class VirtualObjectNode extends Node {
  ResolvedJavaType getType();
  int getEntryCount();
}

class VirtualInstanceNode extends VirtualObjectNode {
  VirtualInstanceNode(ResolvedJavaType type);
  int getFieldIndex(ResolvedJavaField field);
  ResolvedJavaField[] getFields();
}

class VirtualArrayNode extends VirtualObjectNode {
  VirtualArrayNode(ResolvedJavaType componentType, int length);
}
```

Listing 6.4: The interface provided by VirtualObject node and its subclasses.

6.2 Examples

6.2.1 LoadField Node

```
class LoadFieldNode extends FixedNode implements Virtualizable {
  @Input ValueNode object;
  ResolvedJavaField field;
  // ...
  @Override
  void virtualize(VirtualizerTool tool) {
    VirtualState state = tool.getVirtualState(object);
    if (state != null) {
      VirtualInstanceNode virtual =
          (VirtualInstanceNode) state.getVirtualObject();
      int fieldIndex = virtual.getFieldIndex(field);
      tool.replaceWith(state.getEntry(fieldIndex));
    }
  }
}
```

Listing 6.5: Implementation of virtualize for the LoadField node.

The implementation of the virtualize method that handles the loading of a value from an object's field is shown in Listing 6.5. If the receiver of the field load is a virtual object, then the LoadField node can be replaced with the field's current value. getFieldIndex is used to query the entry index for the field, which in turn is used to query the entry's value. Finally, the LoadField node replaces itself using replaceWith.

6.2.2 NewInstance Node

```java
class NewInstanceNode extends FixedNode
                      implements VirtualizableAllocation {
  ResolvedJavaType type;
  // . . .
  boolean isReference() {
    // returns whether the type subclasses java.lang.ref.Reference
  }
  ConstantNode defaultFieldValue(ResolvedJavaField field) {
    // returns the default value for the given field (0, null, ...)
  }
  @Override
  void virtualize(VirtualizerTool tool) {
    /*
     * Reference objects can escape into their ReferenceQueue at any
     * safepoint, therefore they're excluded from escape analysis.
     */
    if (!isReference()) {
      VirtualInstanceNode virtual = new VirtualInstanceNode(type);
      ResolvedJavaField[] fields = virtual.getFields();
      ValueNode[] newEntries = new ValueNode[fields.length];
      for (int i = 0; i < newEntries.length; i++) {
        newEntries[i] = defaultFieldValue(fields[i]);
      }
      tool.createVirtualObject(virtual, newEntries);
      tool.replaceWithVirtual(virtual);
    }
  }
}
```

Listing 6.6: Implementation of virtualize for the NewInstance node.

The NewInstance node, which creates a new object, implements the VirtualizableAllocation interface, because it introduces a new VirtualObject node.

If the type of the allocated object is java.lang.ref.Reference or a subtype of it, no VirtualObject node is created because objects of these types can escape at any safepoint. In all other cases, a new VirtualInstance node with default initial values (0, null, false) for all fields is created and used as a replacement for the NewInstance node.

[3]The garbage collection system will add the Reference objects to their corresponding ReferenceQueue.

6.2.3 ObjectGetClass Node

The ObjectGetClass node, which represents a call to Object.getClass, is shown in Listing 6.7. Since the exact type of a virtual object is known, the ObjectGetClass node can replace itself with the constant java.lang.Class object for the type.

```
class ObjectGetClassNode extends FixedNode implements Virtualizable {
  @Input private ValueNode object;
  // ...
  void virtualize(VirtualizerTool tool) {
    VirtualState state = tool.getVirtualState(object);
    if (state != null) {
      ResolvedJavaType type = state.getVirtualObject().getType();
      Constant clazz = type.getEncoding(Representation.JavaClass);
      tool.replaceWithValue(ConstantNode.forConstant(
          clazz, tool.getMetaAccessProvider(), graph()));
    }
  }
}
```

Listing 6.7: Implementation of virtualize for the ObjectGetClass node.

6.2.4 ObjectClone Node

The ObjectClone node, which represents a call to Object.clone, is one of the most complex implementations of virtualize. As shown in Listing 6.8, if can handle both virtual and non-virtual objects:

- If the object to be cloned is virtual and of a cloneable type, a new VirtualObject node is created by cloning the existing one. The new VirtualObject node is added with a copy of the state of the old one. Note that no locks are set on the new object - Object.clone always creates unlocked objects, even if the original object was locked.

- If the object to be cloned is a non-virtual instance of a known and cloneable type, a new VirtualInstance node is created. Its initial values are populated with newly created LoadField nodes for all fields.

  If only some of the fields of the new virtual object are used, unnecessary LoadField nodes will be removed by subsequent optimizations.

The handling of constant-size and non-virtual arrays was omitted for the sake of brevity. It is similar to the handling of non-virtual instances.

```
class ObjectCloneNode extends FixedNode
                      implements VirtualizableAllocation {
  @Input ValueNode object;
  // . . .
  static boolean isCloneableType(
      ResolvedJavaType type, VirtualizerTool tool) {
    // determines whether type implements Cloneable
  }
  static ResolvedJavaType getConcreteType(
      ValueNode value, VirtualizerTool tool) {
    // returns a concrete cloneable type for value, or null
  }

  @Override
  public void virtualize(VirtualizerTool tool) {
    VirtualState state = tool.getVirtualState(object);
    if (state != null) {
      VirtualObjectNode virtual = state.getVirtualObject();
      if (isCloneableType(virtual.getType(), tool)) {
        // virtual object
        int entryCount = virtual.getEntryCount();
        ValueNode[] newEntries = new ValueNode[entryCount];
        for (int i = 0; i < entryCount; i++) {
          newEntries[i] = state.getEntry(i);
        }
        VirtualObjectNode newVirtual = virtual.duplicate();
        tool.createVirtualObject(newVirtual, newEntries);
        tool.replaceWithVirtual(newVirtual);
      }
    } else {
      // non-virtual object
      ValueNode obj = tool.getReplacedValue(object);
      ResolvedJavaType type = getConcreteType(obj, tool);
      if (type != null && !type.isArray()) {
        VirtualInstanceNode newVirtual =
            new VirtualInstanceNode(type);
        ResolvedJavaField[] fields = newVirtual.getFields();

        ValueNode[] newEntries = new ValueNode[fields.length];
        for (int i = 0; i < fields.length; i++) {
          LoadFieldNode load = new LoadFieldNode(obj, fields[i]);
          tool.addNode(loads[i]);
          newEntries[i] = load;
        }
        tool.createVirtualObject(newVirtual, newEntries);
        tool.replaceWithVirtual(newVirtual);
      }
    }
  }
}
```

Listing 6.8: Implementation of virtualize for the ObjectClone node.

Chapter 7

Case Studies

This chapter evaluates the behavior and the effects of Partial Escape Analysis. It shows the benefits of Escape Analysis in general, and how Partial Escape Analysis improves upon it. The additional allocations introduced by abstractions of the Java language and the Java Class Library are explored, along with how Partial Escape Analysis helps to mitigate them.

7.1 Graal PhiNode

```
public ValueNode singleBackValue() {
  assert merge() instanceof LoopBeginNode;
  ValueNode differentValue = null;
  int start = merge().forwardEndCount();
  int end = values().size();
  for (ValueNode n : values().subList(start, end)) {
    if (differentValue == null) {
      differentValue = n;
    } else if (differentValue != n) {
      return null;
    }
  }
  return differentValue;
}
```

Listing 7.1: The singleBackValue function of Graal's PhiNode class.

The singleBackValue method shown in Listing 7.1 is an excellent example of why Escape Analysis is such an important optimization. The compiler uses this method during loop analysis in order to determine whether all loop ends have the same value for a given Phi node.

While it is not obvious that any objects are allocated in this method, three objects are allocated for each call:

- The call to subList will allocate an instance of java.util.SubList that represents the selected portion of the list.
- Subsequently, the for-each loop calls iterator on the SubList, which allocates an iterator.
- The constructor of this iterator calls iterator on the values collection, which allocates an instance of AbstractList.ListIterator.

The values collection is an ArrayList, so that iterating over it amounts to simply incrementing an index. However, the fact that the iteration is hidden behind three objects makes it very hard for the compiler to reason about the loop. The overhead for loading and storing the iteration index within the iterator is significant, and additional, unnecessary checks for modification counts are inserted.

Partial Escape Analysis manages to remove all object allocations from this method, so that the remaining code is similar to a simple for loop from start to end. It cuts the size of this method in half, from 245 nodes to 123 nodes.

7.2 DaCapo Sunflow

Listing 7.2 shows a slightly simplified version of one of the most important methods in the DaCapo Sunflow benchmark. It calculates the illumination caused by point light sources on a specific point in the scene. It is called 100.000 times during each benchmark iteration, and the scene contains 130 light sources on average. Graal aggressively inlines all method calls except for one inside traceShadow whose target method is too large to be inlined.

The Color class is a mutable data structure that contains three float values, and Ray is a mutable data structure that contains eight float values. After inlining, the getIrradiance method contains three allocations:

- Color.black() in line 3 allocates a new color value that is initialized to black.
- new Ray(...) in line 8 allocates a new ray from the current point to the light source.
- Color.blend(...) in line 14 allocates a new color value for the result of the blend operation.

```
1  public Color getIrradiance(ShadingState state, Color diffRefl) {
2    float b = (float) Math.PI * c / diffRefl.getMax();
3    Color irr = Color.black();
4    Point3 p = state.getPoint();
5    Vector3 n = state.getNormal();
6    int set = (int) (state.getRandom(0, 1, 1) * numSets);
7    for (PointLight pl : virtualLights[set]) {
8      Ray r = new Ray(p, pl.p);
9      float dotNlD = -(r.dx * pl.n.x + r.dy * pl.n.y + r.dz * pl.n.z);
10     float dotND = r.dx * n.x + r.dy * n.y + r.dz * n.z;
11     if (dotNlD > 0 && dotND > 0) {
12       float r2 = r.getMax() * r.getMax();
13       Color opacity = state.traceShadow(r);
14       Color power = Color.blend(pl.power, Color.BLACK, opacity);
15       float g = (dotND * dotNlD) / r2;
16       irr.madd(0.25f * Math.min(g, b), power);
17     }
18   }
19   return irr;
20 }
```

Listing 7.2: The InstantGI.getIrradiance function from DaCapo Sunflow.

Partial Escape Analysis has the following effects on this method:

- The allocation of irr in line 3 is moved to the end of the method. All field loads and stores on this object are removed and replaced with local variables in SSA form. The materialization at the end will immediately initialize the object with the correct values.

- The allocation of r in line 8 is moved down to before the non-inlined call in line 13. All field loads and stores up to this point are removed, while loads and stores after this point will remain the same. It is important to note that line 13 is only reached if the if condition in line 11 is true, which happens in 56% of the cases.

- The allocation of power in line 14 can be removed altogether, since the object does not escape at all. It will be completely replaced with local variables.

While only 33% of the allocations sites could be removed completely, 64% of all dynamic allocations were avoided. For each benchmark iteration, this amounts to 494 MB of memory that does not need to be allocated and collected. Without Partial Escape Analysis, only 46% of this (230 MB) could have been avoided. The size of the method was also reduced considerably, from 303 nodes to 247 nodes. Most of the removed nodes were loads and stores on the analyzed objects.

7.3 ArrayList Initialization

The example method shown in Listing 7.3 creates a new ArrayList instance that is initialized with a list consisting of the three given objects. This seemingly simple piece of Java code incurs numerous allocations:

- The call to the method asList, which has a variable number of arguments, leads to the allocation and initialization of an Object array for the three argument.
- The asList method creates an instance of List and stores the array in the a field.
- The example method then allocates an instance of ArrayList.
- The constructor of ArrayList calls List.toArray, which clones the Object array.

The first two allocations, i.e., the varargs array and the List, are not reachable by the time example returns. Therefore, they will be removed completely by Partial Escape Analysis.

The other two allocations will escape at the return statement within example. However, all operations upon them up to this point, i.e., field and array loads and stores, can be virtualized. This in turn will cause the checks on array and elementData.getClass() to fold away. All that remains of the example method is the materialization of the two objects which immediately initializes them with their final state.

7.4 Escaping and Non-Escaping Allocations

The example in Listing 7.4 demonstrates why the context sensitivity of Partial Escape Analysis is important. It allocates two objects: an instance of TestClassObject (a) and an instance of TestClassInt (b). Initially, a reference to b object is stored into a.

The TestClassObject referenced by a is returned, and therefore escapes, at the end of the method. However, since the reference to b is overwritten with null beforehand, the TestObjectInt object is recognized as non-escaping by Partial Escape Analysis. All intermediate loads and stores on the objects can be removed, and the TestClassObject object will be materialized with its final contents at the end of the method. Traditional Escape Analysis needs to tread a as escaping throughout the method, so that b, which is stored into a field of a, also escapes.

```
public class Arrays {
  private static class List {
    private Object[] a;
    public ArrayList(Object[] array) {
      if (array == null) {
        throw new NullPointerException();
      }
      a = array;
    }
    public Object[] toArray() {
      return a.clone();
    }
    // ...
  }
  public static List asList(Object... a) {
    return new List(a);
  }
  // ...
}

public class ArrayList {
  public ArrayList(Collection c) {
    elementData = c.toArray();
    size = elementData.length;
    if (elementData.getClass() != Object[].class) {
      elementData = Arrays.copyOf(elementData, size, Object[].class);
    }
  }
  // ...
}

Object example(Object a, Object b, Object c) {
  return new ArrayList(Arrays.asList(a, b, c));
}
```

Listing 7.3: A small example method that creates and initializes an ArrayList.

```
void example(int x) {
  TestClassObject a = new TestClassObject();
  TestClassInt b = new TestClassInt();
  b.intField = x;
  a.objectField = b;
  if (a.objectField.intField == 0) {
    // . . .
  }
  a.objectField = null;
  return a;
}
```

Listing 7.4: Code example with an escaping and a non-escaping allocation.

Chapter 8

Evaluation

After discussing the different ways in which Partial Escape Analysis can have a positive influence on compiled code, this chapter presents an evaluation of the performance impact of the algorithm on various benchmarks. Finally, this chapter looks at the impact of Partial Escape Analysis on compile time.

8.1 Sources of Performance Increases

Escape Analysis can improve performance in a number of ways:

- Fewer dynamically allocated bytes lead to less frequent garbage collection and fewer objects to be visited on the heap. The actual impact on performance depends heavily on the garbage collection strategy.

- Fewer dynamic object allocations lead to fewer executions of the allocation logic. Even though a fast-path allocation from the thread local allocation buffer usually takes only (8 + number of fields) CPU instructions, this is still a source of significant performance gains.

- Fewer lock operations lead to fewer executions of the monitor enter and exit logic. Even in the best case (biased locking [48] enabled and uncontended lock), entering a locked region takes 12 CPU instructions and exiting it takes 4 CPU instructions.

- Scalar Replacement avoids dereferencing pointers to objects and makes fields and array elements efficiently accessible as local variables. This leads to less CPU cycles and fewer cache misses.

- By Scalar Replacement, the compiler also gains more knowledge about the values propagated through objects. The effect is similar to a perfect alias analysis for these objects.

Partial Escape Analysis can realize these improvements in individual branches, even if the objects escape in other, potentially unlikely, branches. The first four effects are only significant if large numbers of allocations are removed, while the last one can also provide benefits if only a few key allocations are optimized.

8.2 Performance Impact

We evaluated our implementation of Partial Escape Analysis by running and analyzing a number of benchmarks. All benchmarks were executed on server-class Xeon E5-2690 CPUs, with the Java VM configured to use up to 2GB of heap.

Our benchmark process runs each of the 14 benchmarks of the DaCapo suite[1] and each of the 12 benchmarks of the ScalaDaCapo suite[2], warming them up for enough iterations to arrive at a stable peak performance. In addition to that, it runs the SPECjbb2005 benchmark, which also contains a warmup phase.

This process was executed 10 times for a configuration without Partial Escape Analysis and 10 times for a configuration with Partial Escape Analysis. The numbers we report in this thesis are the averages of the benchmark results for the 10 runs. In separate runs, we also collected statistics for the size and number of allocations and the number of lock operations for each benchmark.

The results of our evaluation are shown in Tables 8.1 and 8.2. For each benchmark, Table 8.1 contains the allocation size per iteration in MB as well as the millions of allocations per iteration. Table 8.2 shows the iterations per minute. Both tables show the measurements with and without Escape Analysis as well as their difference in percent, which shows the speedup achieved by our optimization. The “average” row shows the average percentage. Since SPECjbb2005’s iterations are much smaller than the ones in DaCapo and ScalaDaCapo, we scaled these numbers by 10^6. This makes the table more uniform, but does not influence the relative changes. There is no table showing the number of lock operations, because few of these numbers are significant.

Allocated Bytes Most benchmarks show a high allocation rate. DaCapo lusearch, sunflow, tradesoap and xalan, ScalaDaCapo factorie and kiama, and SPECjbb2005

[1]DaCapo version 9.12-bach [4]
[2]ScalaDaCapo version 0.1.0 (2012-02-16) [51]

		MB / Iteration			MAllocs. / Iteration		
		without	with	Δ	without	with	Δ
DaCapo	avrora	81	82	+2.0%	2	2	+2.3%
	batik	63	60	-3.8%	1	1	-7.9%
	eclipse	6,041	6,021	-0.3%	67	67	-0.4%
	fop	172	166	-3.5%	3	3	-5.6%
	h2	1,336	1,267	-5.2%	31	30	-5.9%
	jython	2,242	2,057	-8.3%	28	23	-15.2%
	luindex	25	23	-7.3%	–	–	-17.6%
	lusearch	5,496	5,473	-0.4%	10	10	-3.9%
	pmd	426	414	-2.9%	8	7	-5.2%
	sunflow	2,707	2,010	-25.7%	62	43	-30.6%
	tomcat	691	685	-0.8%	7	7	-2.4%
	tradebeans	3,640	3,354	-7.8%	64	57	-11.1%
	tradesoap	5,907	5,753	-2.6%	63	59	-6.2%
	xalan	1,289	1,270	-1.4%	10	10	-2.2%
	average			-4.9%			-8.0%
ScalaDaCapo	actors	1,866	1,550	-17.0%	56	45	-18.5%
	apparat	3,418	3,306	-3.3%	74	70	-5.5%
	factorie	43,393	17,996	-58.5%	1,397	547	-60.9%
	kiama	642	600	-6.6%	13	11	-11.2%
	scalac	758	648	-14.5%	19	15	-22.6%
	scaladoc	1,189	1,046	-12.0%	24	18	-24.0%
	scalap	68	62	-8.8%	2	2	-12.5%
	scalariform	337	292	-13.3%	10	8	-16.5%
	scalatest	263	261	-1.0%	4	3	-2.4%
	scalaxb	226	212	-5.9%	4	3	-13.8%
	specs	588	362	-38.4%	12	3	-72.0%
	tmt	2,798	2,698	-3.6%	38	34	-12.2%
	average			-15.2%			-22.7%
SPECjbb2005†		11,608	9,741	-16.1%	180	111	-38.1%

Table 8.1: Evaluation of size and number of allocations on DaCapo, ScalaDaCapo and SPECjbb2005.

† Scaling factor for SPECjbb2005: 10^6 (numbers are per one million iterations).

allocated more than 1GB per second. Most of ScalaDaCapo, some of DaCapo, and SPECjbb2005 see a large decrease in allocated bytes per benchmark iteration due to Partial Escape Analysis. ScalaDaCapo factorie has the highest decrease at 58.5% or 24.5GB per iteration.

The relative changes for benchmarks with very low allocation rates, such as luindex, batik and avrora, are not significant.

Number of Allocations In general, benchmarks with a high number of allocated bytes also show a high number of allocations. The relative decrease in the number of allocations is usually higher than the decrease in the number of allocated bytes,

		Iterations / Minute		
		without	with	Speedup
DaCapo	avrora	25.71	25.33	-1.5%
	batik	47.92	48.08	+0.3%
	eclipse	3.28	3.27	-0.2%
	fop	150.75	172.41	+14.4%
	h2	11.64	11.98	+2.9%
	jython	25.35	24.80	-2.1%
	luindex	118.81	118.58	-0.2%
	lusearch	111.32	112.15	+0.7%
	pmd	29.64	29.94	+1.0%
	sunflow	54.55	55.40	+1.6%
	tomcat	46.73	48.78	+4.4%
	tradebeans	9.97	10.61	+6.4%
	tradesoap	13.89	14.08	+1.4%
	xalan	156.25	159.15	+1.9%
	average			+2.2%
ScalaDaCapo	actors	17.10	18.81	+10.0%
	apparat	6.11	6.94	+13.7%
	factorie	1.95	2.59	+33.0%
	kiama	116.28	135.44	+16.5%
	scalac	23.09	24.12	+4.4%
	scaladoc	20.39	20.99	+3.0%
	scalap	472.44	555.56	+17.6%
	scalariform	127.66	137.61	+7.8%
	scalatest	58.14	62.24	+7.1%
	scalaxb	100.50	105.26	+4.7%
	specs	35.03	36.43	+4.0%
	tmt	13.06	13.50	+3.3%
	average			+10.4%
SPECjbb2005[†]		11.07	12.04	+8.7%

Table 8.2: Evaluation of performance on DaCapo, ScalaDaCapo and SPECjbb2005.
[†] Scaling factor for SPECjbb2005: 10^6 (numbers are per one million iterations).

since the allocations not removed by Partial Escape Analysis often contain large arrays.

Number of Locks We did not observe a significant reduction in the number of lock operations in most benchmarks. DaCapo tomcat shows a 4% or 155,000 operations per second reduction, and SPECjbb2005 shows a 3.8% or 2,400,000 operations per second reduction.

Iterations per Minute In order to be consistent we converted all timings into an iterations per minute metric. Most of the benchmarks show some improvement in performance, with many being above 10%. ScalaDaCapo factorie benefits the most in terms of performance, with a 33% improvement in iterations per minute.

Notably, the DaCapo jython benchmark shows a 2.1% decrease in performance. Partial Escape Analysis can in rare cases increase the size of compiled methods, which has a negative influence on this benchmark.

While there will never be more dynamic allocations in code optimized by Partial Escape Analysis than in the unoptimized code, the number of allocation sites can increase in some cases, e.g., when the same virtual object is materialized in multiple independent branches. Benchmarks that are sensitive to this may suffer from the increased code size of multiple allocation sites.

8.3 Comparison

The HotSpot server compiler, which is arguably the most widely used jit compiler performing Escape Analysis, benefits less from enabling Escape Analysis than Graal does from enabling Partial Escape Analysis (0.9% vs. 2.2% on DaCapo, 7.4% vs. 10.4% on ScalaDaCapo, 5.4% vs. 8.7% on SPECjbb2005). However, it is hard to tell the difference between better Escape Analysis and the rest of the compiler performing better in the presence of Escape Analysis.

8.4 Compilation Performance

Partial Escape Analysis is a non-trivial optimization that performs iterations over the whole compiler graph. Graal takes care not to invoke it on a graph that does not contain allocations, but most methods above a certain size do contain allocations.

As Table 8.3 shows, Partial Escape Analysis takes a very consistent fraction of the total compilation time over all benchmarks, which is in the order of 3.5% to 4.5%. This includes the time for the scheduling operation that is necessary in preparation for the analysis (see Section 5.2.1). The iteration over the compiler graph, the analysis and the changes to the graph together take approximately half of the total time spent in the Partial Escape Analysis phase, while the other half is required for the scheduling.

Since each node is only visited once under normal circumstances, the time for the analysis depends mainly on the number of nodes in the graph and the number of allocations. Because of the iterative loop processing the analysis time could, theoretically, rise exponentially for nested loops. The number of iterations it takes for a loop to

		Time taken (milliseconds)				
		Total	PEA Accumulated		PEA Flat	
DaCapo	avrora	19,033.6	679.1	3.6%	353.7	1.9%
	batik	39,037.8	1,358.5	3.5%	730.1	1.9%
	eclipse	193,950.5	6,168.1	3.2%	3,000.1	1.5%
	fop	45,561.0	1,768.1	3.9%	920.1	2.0%
	h2	44,423.1	1,704.5	3.8%	843.5	1.9%
	jython	150,858.2	3,951.3	2.6%	1,860.1	1.2%
	luindex	25,460.4	980.6	3.8%	479.2	1.9%
	lusearch	22,048.4	817.2	3.7%	440.3	2.0%
	pmd	67,803.9	2,520.8	3.7%	1,124.5	1.7%
	sunflow	22,627.0	1,032.7	4.6%	521.0	2.3%
	tomcat	123,435.3	2,816.4	2.3%	1,405.0	1.1%
	tradebeans	44,024.6	1,710.5	3.9%	877.2	2.0%
	tradesoap	97,618.0	3,861.8	4.0%	2,000.8	2.0%
	xalan	37,955.8	1,467.4	3.9%	758.0	2.0%
	average			3.6%		1.8%
ScalaDaCapo	actors	29,601.2	1,096.5	3.7%	501.5	1.7
	apparat	41,187.7	1,588.6	3.9%	699.1	1.7%
	factorie	18,094.5	724.6	4.0%	312.5	1.7%
	kiama	26,997.2	1,195.3	4.4%	527.9	2.0%
	scalac	103,993.9	3,555.9	3.4%	1,666.3	1.6%
	scaladoc	68,586.4	2,419.5	3.5%	1,152.7	1.7%
	scalap	23,667.2	888.5	3.8%	457.3	1.9%
	scalariform	28,330.8	1,025.8	3.6%	523.2	1.8%
	scalatest	40,168.8	1,668.4	4.2%	793.0	2.0%
	scalaxb	29,008.9	1,226.4	4.2%	561.4	1.9%
	specs	26,047.7	997.5	3.8%	529.4	2.0%
	tmt	27,116.7	1,115.2	4.1%	513.3	1.9%
	average			3.9%		1.8%
SPECjbb2005		30,750.2	1,169.3	3.8%	506.1	1.6%

Table 8.3: Relative amount of compilation time taken by Partial Escape Analysis. "PEA Accumulated" includes the time taken by the scheduling performed before the analysis, while "PEA Flat" does not.

stabilize is bounded by the number of `virtual` objects in the loop's predecessor state, since each time the loop is processed at least one object needs to be materialized for another iteration to happen. In practice, only 30% of all loops are processed twice, no loop is processed more than two times, and even deeply nested loops converge quickly.

Chapter 9

Related Work

9.1 Escape Analysis

Escape Analysis initially emerged as an extension of sharing and aliasing analyses, which were researched mainly in the context of functional languages at the time. The starting point was lifetime analysis, which tries to determine the relation between the lifetimes of objects, i.e., if the reachability of one object is depending on the reachability of another object.

Ruggieri and Murtagh introduced a lifetime analysis for dynamically allocated objects [47]. Taking as an inspiration the lifetime information collected by generational garbage collection, they argued that the lifetime of some objects can be tied to a specific function by compile-time analysis. Their analysis is limited to a very simple language with a restricted type system, and its result is used to put some objects into heap areas that can be reclaimed as soon as a specific method is left.

Chase discusses allocation optimizations with a focus on the safety of these optimizations [12]. While the algorithm presented in this work is very conservative, not described in detail and lacking a practical implementation, it can also be considered one of the first Escape Analysis algorithms.

Deutsch extended previous work on object lifetime analysis towards higher-order languages with first-class continuations [16]. The program is translated into sequences of abstract operations, on which the analysis, which is based on abstract interpretation, is performed. While this work includes a prototype implementation written in ML, it does not report any results. and the optimizations on object allocations that could be performed based on the analysis are not explored.

The work of Goldberg and Park introduced the term “Escape Analysis” and defines it as “a particular instance of lifetime analysis in which the lifetime of a function’s activation record is compared to the objects inside the function". They first use this analysis to determine whether the closure of a function call has a greater lifetime than the function call itself [20], in which case it cannot be allocated on the stack. Later, they apply the results of the analysis on reference counting schemes [45], so that useless reference increments and decrements can be avoided. This is a very different use of Escape Analysis, because it needs to prove that an object is always reachable within a certain region, as opposed to proving that an object is not reachable after a certain point. Finally, they apply the analysis to lists allocated by higher order functional programs [46]. They describe certain strategies for optimizations based on the escape information, like reuse of cons cells, stack allocation of cons cells, and bulk allocation of cons cells.

Carr and Kennedy introduced a source-to-source transformation called Scalar Replacement [11]. Drawing from previous work on dependency analysis in the context of automatic vectorization, they developed a data-flow analysis that replaces accesses to array elements with variables, up to the point where the uniqueness of the array element is no longer guaranteed. Their motivation was allowing the register allocator to assign registers to array elements. The combination of Escape Analysis and Scalar Replacement was introduced only later by Whaley [62].

Deutsch improves upon the complexity of previous Escape Analysis algorithms [15]. Drawing from his own previous work on lifetime analysis, he refines the work of Goldberg and Park to $O(n \log^2 n)$ instead of an exponential complexity for first-order functions, while keeping the exact same accuracy. This work also includes a formal proof that Escape Analysis on second-order functions has an inherent exponential complexity. He concludes with preliminary evidence showing that Escape Analysis "may [sic] be useful in practice".

9.1.1 Java

Blanchet extended previous work on Escape Analysis to allow for precise treatment of assignments, and uses the results of this control-flow-sensitive analysis for Stack Allocation [5]. The algorithm he presented extracts information from the source program into a data structure that is later processed iteratively until a fixed point is reached. He is the first to emphasize that indirect effects, e.g., improved data locality, play an important role in the performance improvements facilitated by Escape Analysis. The subsequent work in [6] was one of the first to completely support the Java language, and he later contributed a formal proof of the correctness of his transformations [7].

Choi et al. presented both a control-flow-sensitive and a control-flow-insensitive Escape Analysis for Java [13]. Even the control-flow-sensitive version, which has some similarities to our Partial Escape Analysis, is only used to make global decisions about escapability. In addition to the missing loop handling, it does not collect enough information to perform on-the-fly Scalar Replacement. The control-flow-insensitive version of this work is used to perform Scalar Replacement in the HotSpot server compiler [43] starting with version Java SE 6u23.

Whaley and Rinard introduced a new program abstraction called *points-to-escape graph* that combines pointer and Escape Analysis [63]. Their graph, which represents objects as nodes and connections between objects as edges, contains information about both objects created within a method and outside of a method. The information extracted from this graph is used to apply lock elision and stack allocation. They place a particular emphasis on the incremental nature of their analysis, i.e., on the ability to combine the graphs for independently analyzed methods in order to achieve a more precise result. Whaley later, in the context of partial method compilation, combined of Escape Analysis with Scalar Replacement to completely eliminate the allocation of objects instead of allocating them on the stack [62].

Beers et al. developed a mechanism to perform the Escape Analysis ahead of time and verify the analysis results when the code is loaded [3]. While their analysis is less precise than the work of Whaley and Rinard, it can support dynamic class loading. The results of the analysis are incorporated into annotations within the class files themselves. In their experiments, verifying the result of the analysis takes approximately one fourth the analysis time.

Kotzmann and Mössenböck introduced an implementation of Escape Analysis for the HotSpot client compiler that works on the compiler's high-level intermediate representation (HIR) [32, 33]. It uses the control-flow-insensitive *equi-escape sets* algorithm, and was the first to apply Escape Analysis in the presence of deoptimization. The result of the equi-escape sets algorithm is used to perform Scalar Replacement, stack allocation and lock elision. A similar algorithm was used by Molnar et al. [37] to perform stack allocation of objects in the Cacao VM.

Shankar et al. presented JOLT, a lightweight dynamic analysis which tries to reduce *object churn*, the excessive creation of short-lived objects [52]. It does so by guiding inlining so that Escape Analysis is more effective. Their approach significantly increases the number of allocations amenable to Escape Analysis. Combining it with our approach of Partial Escape Analysis would be an interesting future work.

9.1.2 Other Languages

Current development versions of the v8 JavaScript engine [21] contain an implementation of Escape Analysis which performs a very local analysis (looking only at the usages of an allocation) to determine whether an allocation escapes or not. The simulation of the effects of operations for different allocations happens independently, which implies that complex cases involving multiple objects cannot be handled.

9.2 Partial Escape Analysis

While there is no published work similar to Partial Escape Analysis, two algorithms implemented in production-quality Virtual Machines for Lua and Python show similarities to our technique. Both systems are tracing-based just-in-time compilers for dynamic languages, implementing the techniques pioneered by the Dynamo project [2].

9.2.1 LuaJIT

The LuaJIT compiler is a just-in-time compiler and runtime for the Lua language which is available for many platforms and architectures. It performs a so-called *Allocation Sinking* optimization [44], which is essentially an Escape Analysis tailored heavily towards trace compilation. Lua applications show the same behavior as many dynamic and object-oriented applications: they have many fallback paths in which objects escape, so that the benefits of traditional Escape Analysis techniques are limited.

LuaJIT is a trace compiler, i.e., it includes an interpreter that records sequences of executed bytecodes and collects these traces for later compilation. Once traces have been encountered often enough, they are combined, preprocessed, and scheduled for compilation. Whenever a check within the trace does not evaluate to the expected result, execution needs to exit the trace. Traces can have many exits, which are similar to deoptimization points in that they contain a description of the interpreter state at this point. In contrast to deoptimizations, however, some of these exits may be within the hot paths of the code, so that it is important that they can be taken efficiently. It is therefore possible for one compiled trace to directly jump to another compiled trace.

The basic principle of LuaJIT's Escape Analysis is that allocations within a trace will not escape at exits. If an allocation is only passed to the trace's exits, and does not escape within the rest of the trace, it can be "sunk" into the exits. The exits will thus not receive a reference to the object, but a description of the object's contents. It is then the responsibility of the callee of the exit to reallocate (and possibly, re-sink) the object.

LuaJIT relies on its advanced alias analysis to remove all loads from non-escaping objects. The actual Escape Analysis works on a concise tuple IR and consists of a backwards marking phase with iterative processing of Phi references, and a forward sweeping phase that tags non-marked instructions as removable, or “sunk”. While the algorithm still takes a global decision, the trace-based nature improves its efficiency, because unlikely branches will reside in separate traces. An escaping reference in one of these side traces will not cause the object to escape in the main trace.

While the trace-based approach to Escape Analysis cannot be compared directly to Partial Escape Analysis, the results are similar: object allocations will stay *sunk*, i.e., virtual, within some branches, while they escape in others.

9.2.2 Escape Analysis in PyPy

PyPy is a Python Virtual Machine that includes a meta-tracing just-in-time compiler developed by Bolz et al. [10]. Its tracing runtime system is written in RPython, a reduced version of Python which is amenable to ahead of time compilation. While LuaJIT is a hand-crafted system tuned for tightness, size and efficiency, PyPy uses techniques for automatically generating compiled code from interpreter snippets, i.e., partial evaluation of interpreters.

Bolz et al. also observed large numbers of object allocations for dynamic, object-oriented applications. Their approach to Escape Analysis [8, 9] is the closest equivalent to Partial Escape Analysis we are aware of. Object allocations within traces are replaced at the allocation site with *virtual objects*, which maintain the shape of the object and the current value of all fields. During a walk over the (linear) list of all instructions in the trace, these virtual objects are updated for stores and queried for loads. Whenever a virtual object escapes, it needs to be *forced*, i.e., allocated and initialized.

Only optimizing allocations within a trace is not sufficient for real-world applications, therefore they extended the algorithm to allow virtual objects to pass through trace exits. A subsequent trace can then start with virtual objects and continue the analysis.

An interesting property of PyPy’s virtual object technique is that whether an argument passed from one trace to another is virtual or not is a property of the trace’s exit. If one trace exits with a virtual object and another one with an ordinary object reference, two versions of the callee trace need to be compiled. This code duplication during compilation has an effect similar to automatic loop peeling and tail duplication.

Chapter 10

Summary

This chapter outlines possible future work and concludes this thesis.

10.1 Future Work

Partial Escape Analysis has not been explored extensively in academic context. We therefore propose future work in the following areas:

Unstructured Control Flow

The loop processing algorithm explained in Section 5.2.5 is tailored towards Graal IR, which is limited to structured control flow and provides an explicit representation of loops [19].

The same principle of data-flow analysis can, however, also be applied to unstructured control flow. Whenever there are no more nodes where all predecessor states are available, the system needs to speculatively assume a state. As soon as a new state is created as a replacement of the speculative state, the two states need to be checked for equality. If they differ, iteration needs to backtrack to this position.

Parallelized Iteration

The iteration that updates the allocation state could be run in parallel as soon as a control split is encountered. Depending on the shape of the compiled code, significant parts of the analysis could thus be executed in parallel. More fine-grained multithreading within the compiler will become more important as more cores are available for compilation.

Analysis on Unscheduled Graphs

Our algorithm currently relies on the scheduler to order the nodes, so that Partial

Escape Analysis can process them in a valid order. By adding simple invariants to the Graal IR, such as limiting the maximum distance from nodes fixed in control flow to nodes affected by Partial Escape Analysis, the analysis could be performed without a schedule.

Although this will restrict the expressiveness of Graal IR, it will also speed up the analysis and make it more suitable for faster baseline compilation.

Explicit Object Identity

Merging two distinct virtual objects into a Phi function requires the objects to be materialized if the object identity of one of the individual objects is still accessible after the merge, i.e., if one of the objects is not only referenced by the Phi function.

The problem is that the merged virtual object would not be always distinct from or always equal to the original object. This could be solved by making the object identity an explicit value: Each allocation site is assigned a unique id (which needs to be calculated using a loop counter inside loops) that can later on be used to determine equality of object identity.

As long as the object identity is not used, which is by far the most frequent case, all code generated in order to maintain the identity will be removed by dead code elimination later on.

Explicit Object Type

Similar to the object identity, an object's class could also be made explicit. Merging two objects with different types only succeeds of they have the same number and type of fields.

Making Virtual Objects part of the Calling Convention

Similar to the way trace exits are handled in PyPy (see Section 9.2.2), a method call parameter could be passed to the callee as a virtual object, i.e., as the contents of the virtual object. This would require solving several challenges:

- Different entry points or targets will be needed for each combination of virtual and non-virtual parameters. This incurs significant changes to the underlying Virtual Machine infrastructure.

- If the virtual object is materialized within the callee, the caller, which expects the object to still be virtual after the call, needs to be deoptimized. Another solution would be to have a separate return entry point for this case.

10.2 Conclusions

In this thesis, we presented a new approach to perform Escape Analysis, Scalar Replacement and Lock Elision in a fine-grained and control-flow sensitive way. Our analysis does not make a global decision about an object's escapability, but propagates the state of all allocations while iterating over the control flow. It can thus perform optimizations such as Scalar Replacement in one branch while an actual object is created in another branch.

Our implementation of Partial Escape Analysis is part of the open-source Graal compiler. It is a mature component that is extensive exercised by the Graal project's test and benchmark suite. It also plays a crucial role in the context of the Truffle framework, where it is used to virtualize execution frames.

We developed an interface to provide extensibility for new compiler components and new node types. This allows the effect of new node types on allocated objects to be specified without modifying the compiler itself. This interface is also used for all predefined node types.

We evaluated our implementation on the DaCapo, ScalaDaCapo and SPECjbb2005 benchmarks. It incurs a modest performance overhead of 4% on compilation time. Partial Escape Analysis can reduce memory allocated by up to 58.5% and shows an improvement in performance of up to 33%. We did not observe a significant benefit from Lock Elision in the benchmarks we evaluated.

Partial Escape Analysis is a novel technique that is well-suited for modern dynamic compilers. It is easy to understand, easy to implement and shows a significant performance improvement.

List of Figures

List of Tables

Listings

Bibliography

[1] Ken Arnold, James Gosling, and David (David Colin) Holmes. *The Java^{TM} Programming Language*. Addison-Wesley, fourth edition, 2005. ISBN 0-321-34980-6.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000. doi: 10.1145/349299.349303.

[3] Matthew Q. Beers, Christian H. Stork, and Michael Franz. Efficiently verifiable escape analysis. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 2004. doi: 10.1007/978-3-540-24851-4_4.

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

[5] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 25–37. ACM Press, 1998. doi: 10.1145/268946.268949.

[6] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–34. ACM Press, 1999. doi: 10.1145/320384.320387.

[7] Bruno Blanchet. Escape analysis for Java^{TM}: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003. doi: 10.1145/945885.945886.

[8] Carl Friedrich Bolz. Escape analysis in PyPy's jit, 2010. URL `http://morepypy.blogspot.co.at/2010/09/using-escape-analysis-across-loop.html`.

[9] Carl Friedrich Bolz. Using escape analysis across loop boundaries for specialization, 2010. URL `http://morepypy.blogspot.co.at/2010/09/escape-analysis-in-pypys-jit.html`.

[10] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing jit compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM Press, 2009. doi: 10.1145/1565824.1565827.

[11] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software: Practice and Experience*, 24(1):51–77, January 1994.

[12] D. R. Chase. Safety consideration for storage allocation optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–10. ACM Press, 1988.

[13] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19. ACM Press, 1999. doi: 10.1145/320384.320386.

[14] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. doi: 10.1145/115372.115320.

[15] Alain Deutsch. On the complexity of escape analysis. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 358–371. ACM Press, 1997. doi: 10.1145/263699.263750.

[16] Alan Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 157–168. ACM Press, 1990. doi: 10.1145/96709.96725.

[17] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[18] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, pages 1–10. ACM Press, 2013. doi: 10.1145/2542142.2542143.

[19] Zdenek Dvorak. [lno] enable unrolling/peeling/unswitching of arbitrary loops, March 2004. URL `http://gcc.gnu.org/ml/gcc-patches/2004-03/msg02212.html`. GCC Patch Mailing List.

[20] B. Goldberg and Y. G. Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *Proceedings of the European Symposium on Programming*, pages 152–160. Springer-Verlag, 1990.

[21] Google. The v8 JavaScript engine, 2013. URL `https://developers.google.com/v8/`.

[22] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An efficient native function interface for Java. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 35–44. ACM Press, 2013. doi: 10.1145/2500828.2500832.

[23] Christian Häubl and Hanspeter Mössenböck. Trace-based compilation for the Java HotSpot virtual machine. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 129–138. ACM Press, 2011. doi: 10.1145/2093157.2093176.

[24] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Optimized strings for the Java HotSpot^{TM} virtual machine. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 105–114. ACM Press, 2008. doi: 10.1145/1411732.1411747.

[25] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Evaluation of trace inlining heuristics for Java. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1871–1876. ACM Press, 2012. doi: 10.1145/2245276.2232084.

[26] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Context-sensitive trace inlining for Java. *Comput. Lang. Syst. Struct.*, 39(4):123–141, 2013. doi: 10.1016/j.cl.2013.04.002.

[27] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Deriving code coverage information from profiling data recorded for a trace-based just-in-time compiler. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 1–12. ACM Press, 2013. doi: 10.1145/2500828.2500829.

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

[29] *JSR 133: Java^{TM} Memory Model and Thread Specification Revision.* JCP, 2004. URL `http://jcp.org/en/jsr/detail?id=133`.

[30] *JSR 342: Java^{TM} Platform, Enterprise Edition 7 (Java EE 7) Specification.* JCP, 2013. URL `http://jcp.org/en/jsr/detail?id=342`.

[31] Yongxian Jin, Wei Hu, Fengzhen Chen, and Gaofeng Che. Thread escape analysis for Java programs based on soot. In *Proceedings of the International Conference on Test and Measurement*. IEEE Computer Society, 2009. doi: 10.1109/ICTM.2009.5412985.

[32] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 111–120. ACM Press, 2005. doi: 10.1145/1064979.1064996.

[33] Thomas Kotzmann and Hanspeter Mössenböck. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60. IEEE Computer Society, 2007. doi: 10.1109/CGO.2007.34.

[34] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot^{TM} client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5 (1):Article 7, 2008. doi: 10.1145/1369396.1370017.

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

[36] Jeremy Manson. *The Java Memory Model.* PhD thesis, University of Maryland, College Park, 2004.

[37] Peter Molnar, Andreas Krall, and Florian Brandner. Stack allocation of objects in the CACAO virtual machine. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 153–161. ACM Press, 2009. doi: 10.1145/1596655.1596680.

[38] Hanspeter Mössenböck and Michael Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *Proceedings of the International Conference on Compiler Construction*, pages 229–246. Springer-Verlag, 2002.

[39] Hanspeter Mössenböck. Adding static single assignment form and a graph coloring register allocator to the Java HotSpot^{TM} client compiler. Technical Report 15, Institute for Practical Computer Science, Johannes Kepler University, 2000.

[40] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[41] OpenJDK Community. *Graal Project*, 2013. URL `http://openjdk.java.net/projects/graal/`.

[42] OpenJDK Community. *Sumatra Project*, 2013. URL `http://openjdk.java.net/projects/sumatra/`.

[43] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*, pages 1–12. USENIX, 2001.

[44] Mike Pall. Allocation sinking optimization for the LuaJIT compiler, 2013. URL `http://wiki.luajit.org/Allocation-Sinking-Optimization`.

[45] Young Gil Park and Benjamin Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 178–189. ACM Press, 1991. doi: 10.1145/115865.115883.

[46] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127. ACM Press, 1992. doi: 10.1145/143095.143125.

[47] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 285–293. ACM Press, 1988. doi: 10.1145/960116.53991.

[48] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Not.*, 41:263–272, October 2006. doi: 10.1145/1167515.1167496.

[49] Thomas Schatzl, Laurent Daynès, and Hanspeter Mössenböck. Optimized memory management for class metadata in a JVM. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 151–160. ACM Press, 2011. doi: 10.1145/2093157.2093182.

[50] Arnold Schwaighofer. Tail call optimizations for the Java HotSpot^{TM} VM. Master's thesis, Johannes Kepler University, Linz, 2009.

[51] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: design and analysis of a scala benchmark suite for the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 657–676. ACM Press, 2011. doi: 10.1145/2076021.2048118.

[52] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 127–142. ACM Press, 2008. doi: 10.1145/1449764.1449775.

[53] Lukas Stadler. Serializable coroutines for the HotSpot^{TM} Java virtual machine. Master's thesis, Johannes Kepler University Linz, Austria, 2011.

[54] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy continuations for Java virtual machines. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 143–152. ACM Press, 2009. doi: 10.1145/1596655.1596679.

[55] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. Efficient coroutines for the Java platform. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 20–28. ACM Press, 2010. doi: 10.1145/1852761.1852765.

[56] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, pages 49–58. ACM Press, 2012. doi: 10.1145/2414740.2414750.

[57] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An experimental study of the influence of dynamic compiler optimizations on scala performance. In *Proceedings of the 4th Workshop on Scala*. ACM Press, 2013.

[58] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 165:165–165:174. ACM Press, 2014. doi: 10.1145/2544137.2544157.

[59] TIOBE Software BV. Tiobe programming community index, April 2010. `http://www.tiobe.com/tpci.htm`.

[60] Ben L. Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra. Improving compiler-runtime separation with xir. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 39–50. ACM Press, 2010. doi: 10.1145/1735997.1736005.

[61] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 27–51. Springer-Verlag, 2008. doi: 10.1007/978-3-540-70592-5_3.

[62] John Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–179. ACM Press, 2001. doi: 10.1145/504311.504295.

[63] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206. ACM Press, 1999. doi: 10.1145/320384.320400.

[64] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 132–141. ACM Press, 2005. doi: 10.1145/1064979.1064998.

[65] Christian Wimmer and Hanspeter Mössenböck. Automatic object colocation based on read barriers. In *Proceedings of the Joint Conference on Modular Programming Languages*, pages 326–345. Springer-Verlag, 2006. doi: 10.1007/11860990_20.

[66] Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object inlining in the Java Hotspot^{TM} virtual machine. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 12–21. ACM Press, 2007. doi: 10.1145/1254810.1254813.

[67] Christian Wimmer and Hanspeter Mössenböck. Automatic array inlining in Java virtual machines. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 14–23. ACM Press, 2008. doi: 10.1145/1356058.1356061.

[68] Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object fusing. *ACM Transactions on Architecture and Code Optimization*, 7(2):7:1–7:35, 2010. doi: 10.1145/1839667.1839669.

[69] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4):30:1–30:24, January 2013. doi: 10.1145/2400682.2400689.

[70] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the Java HotSpot^{TM} client compiler. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 125–133. ACM Press, 2007. doi: 10.1145/1294325.1294343.

[71] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Visualization of program dependence graphs. In *Proceedings of the International Conference on Compiler Construction*, pages 193–196. Springer-Verlag, 2008.

[72] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming*, 74(5-6):279–295, 2009. doi: 10.1016/j.scico.2009.01.002.

[73] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. Improving aspect-oriented programming with dynamic code evolution in an enhanced Java virtual machine. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 5:1–5:5. ACM Press, 2010. doi: 10.1145/1890683.1890688.

[74] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. Applications of enhanced dynamic code evolution for Java in GUI development and dynamic aspect-oriented programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 123–126. ACM Press, 2010. doi: 10.1145/1868294.1868312.

[75] Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. Safe and atomic run-time code evolution for Java and its application to dynamic AOP. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 825–844. ACM Press, 2011. doi: 10.1145/2048066.2048129.

[76] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming Software*, pages 187–204. ACM Press, 2013. doi: 10.1145/2509578.2509581.

[77] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the Dynamic Languages Symposium*, pages 73–82. ACM Press, 2012. doi: 10.1145/2384577.2384587.

[78] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Unrestricted and safe dynamic code evolution for Java. *Science of Computer Programming*, 78(5): 481–498, May 2013. doi: 10.1016/j.scico.2011.06.005.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, Mai 2014

Dipl.-Ing. Lukas Stadler

“A wizard is never late, nor is he early, he arrives precisely when he means to.”

- Gandalf