Value Type Support in a Just-in-Time Compiler

Master Thesis

Value Type Support in a Just-in-Time Compiler

Christian Hagedorn

Tobias Hartmann
Zoltán Majó
Responsible assistants

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

29. November 2017

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
Laboratory for Software Technology

Abstract

The current type system of Java lacks the possibility to efficiently combine multiple immutable primitive types, which directly contain the value, to form a new value based type. This is only possible by creating a class and using a reference type to access created objects from it on the heap. Those, however, suffer from a performance penalty by a pointer traversal to the heap and an additional space consumption to maintain their identity, their state, and the possibility to mutate the objects. This also entails work of garbage collection as limiting factor.

The Valhalla research project proposes *value types*, a third type in the type system, to exploit the advantages of the existing primitive and reference types. They can combine multiple concrete types together to form a new type, like classes, while treating an instance as being completely immutable and identityless. Therefore, value types can be seen as primitive user-defined types which "code like a class but work like an int".

Over the past three years, a first simplified prototype, called *minimal value type*, emerged for the HotSpot™ JVM which currently supports value types in the interpreter and the server just-in-time compiler (also called C2 compiler). Value type support for the client just-in-time compiler (also called C1 compiler) does not exist yet. This thesis addresses this issue and provides a first working implementation for value types in the C1 compiler. The approach treats value types as immutable objects and exploits their nature to avoid expensive heap allocations with a buffer concept.

The main advantage of this work's design is not only to improve the performance by avoiding heap allocations but also to reduce the directly correlated time spent for garbage collection. Moreover, field accesses of value types from the heap can directly be replaced by statically known buffered values.

A detailed evaluation shows the advantages of the present work compared to the use of normal Java objects. The performance gain becomes even more significant with loops in which allocations can be omitted completely. Even in a method with a single allocation with 16 fields for objects, the value types are up to 35.4 times faster by removing the allocation. In the worst case, value types work like objects but still have a performance advantage of exploiting static field information and thus avoiding field loads from the heap. In addition, the implementation of this thesis opens possibilities for future optimizations and extensions for value types in the C1 compiler.

Zusammenfassung

Dem aktuellen Typsystem von Java fehlt die Möglichkeit einer effizienten Kombination aus mehreren unveränderbaren Primitivtypen, die direkt den Wert enthalten, um einen neuen Typ zu formen. Das ist momentan nur mit einer Klasse möglich, von der Objekte im Heap erstellt und auf die über Referenztypen zugegriffen werden kann. Diese Lösung hat jedoch eine Performance-Einschränkung einer Pointer-Traversierung zum Heap zur Folge. Ein weiterer Nachteil stellt der zusätzlich gebrauchte Heap-Speicherplatz dar, um die Identität und den Zustand von Objekten zu speichern und die Möglichkeit zu gewähren, die Objekte später zu modifizieren. Dies hat auch zur Folge, dass mehr Zeit für Garbage Collection benötigt wird.

Das Valhalla-Forschungsprojekt schlägt *Wertetypen* (English: *value types*) als dritte Kategorie im Typsystem vor, um die Vorteile von Primitiv- und Referenztypen auszunutzen. Diese können mehrere konkrete Typen zu einem neuen Typ zusammenfügen, wie es auch Klassen können, während sie eine Instanz aber als unveränderbar und identitätslos betrachten. Wertetypen können als eine Art primitive, benutzerdefinierte Typen gesehen werden, die sich "als Klasse programmieren lassen, aber wie int-Typen funktionieren". In den letzten drei Jahren ist ein erster vereinfachter Prototyp für Wertetypen in der HotSpot™ JVM entstanden. Dieser wird *minimal value type* genannt und wird momentan im Interpreter und im Server-Compiler (C2-Compiler) unterstützt. Eine gegenwärtige Unterstützung im Client-Compiler (C1-Compiler) existiert jedoch noch nicht. Dieses Problem wird mit dieser Arbeit behoben, welche eine erste funktionierende Implementation von Wertetypen für den C1-Compiler präsentiert. Der Ansatz behandelt Wertetypen als unveränderbare Objekte, um teure Heap-Speicherungen mit einem Buffer-Konzept zu vermeiden.

Der Hauptvorteil dieses Designs bietet eine Performance-Steigerung durch vermiedene Heap-Speicherungen und damit verkürzte Zeit für Garbage Collection. Zusätzlich können Heap-Zugriffe für Felder von Wertetypen direkt durch statisch bekannte und gespeicherte Werte ersetzt werden. Eine detaillierte Auswertung veranschaulicht diese Vorteile im Vergleich zu einer Lösung mit Objekten. Der Performance-Gewinn wird mit Loops, die Heap-Speicherungen komplett vermeiden, noch drastischer. Sogar in einer Methode, die nur eine Speicherung mit 16 Feldern für Objekte durchführt, sind Wertetypen bis zu 35.4 Mal schneller, indem sie die Speicherung komplett entfernen. Im schlimmsten Fall werden Wertetypen als Objekte behandelt. Doch auch dann profitieren sie noch vom Vorteil, Heap-Zugriffe für Felder zu vermeiden. Die Implementation dieser Arbeit öffnet Türen für zukünftige Optimierungen und Erweiterungen für Wertetypen im C1-Compiler.

Acknowledgments

First, I want to thank my supervisors Zoltán Majó and Tobias Hartmann for giving me this big opportunity to work on this project. I appreciated the technical help and the personal support during the thesis, especially at the weekly meetings.

I also want to thank my family and Darina Schweizer for their support, feedback, and big encouragement throughout the entire thesis. I probably would not be at this point in my life where I am today without you.

Christian Hagedorn

November, 2017

Contents

1 Introduction

1.1 Motivation

Many Java programs use small immutable data structures, such as complex numbers with a real and an imaginary part or two-dimensional integer points in the plain, that do not require an *identity*. The type system of Java provides two kinds of types for this task: hardwired basic types, known as *primitive types* without identity (`boolean, char, byte, short, int, long, float,` and `double`) and *reference types* with identity [20]. Variables of a primitive type directly contain the value. They do not have an identity and therefore do not need to store any additional information. For example, an `int` needs exactly four bytes containing a 32-bit integer value [51]. No identity also means that two primitive type variables containing the same value cannot be distinguished. However, combining multiple primitive types to form a new type can only be achieved by writing an appropriate class and creating objects from it. These objects are mostly stored on the heap and not directly as value in a variable, as it is done for primitive types. Instead, Java provides reference type variables which contain an *address* to the actual content of an object on the heap [1]. Therefore, reference accesses need an additional level of indirection (compared to accessing primitive types). An important advantage of objects is the ability to distinguish them from each other due to their inherent identity, even if they contain the same values.

The straightforward way to create an immutable and conceptually identityless type is to write a class. For example, the two-dimensional integer points from the previous paragraph could only be implemented as a new type by writing a `final` class that contains two `final int` fields for the x- and y-component, respectively. However, the created immutable objects at runtime still get an inherent identity due to their nature, even if it is not intended. At first sight, this approach looks simple and efficient. But these small immutable objects have a significant additional cost in Java compared to their actual payload of two 32-bit integer numbers represented by two primitive typed `int` variables without an identity. Each object requires up to 24 extra bytes to store the header (essential information about the object and its state) and a reference [19]. Furthermore, new objects occupy heap space which results in more work for garbage collection. A later access to the heap requires an additional performance limiting pointer traversal [60].

Thus, small immutable data structures with no identity cannot be implemented in a satisfying way in the current Java version. What they actually need is a combination of the advantages of

performance and memory of identityless primitive types and of creating new types by coding a class. *Value types* try to fill this gap in Java by providing a third type in the type system. They can define immutable types without identity that can be directly stored in memory, like primitive types (which could also be in registers or on the stack). Value types can therefore be seen as user-defined primitive types which follow the motto "codes like a class, works like an int" [18].

Since the first proposal three years ago, a minimized but viable subset of value types emerged from various discussions and prototyping attempts in the HotSpot™ Java Virtual Machine. This has been explored in the scope of the *Valhalla* project of the OpenJDK [58] (also see Section 3.2). The goals were set to create a first prototype for value types, called *minimal value types*, without constraining future developments of the Java language and the Java Virtual Machine (JVM) [18]. Its focus lies on exploration and is not fixed to be released in a specific Java release (especially not in the newest Java SE 9) [3].

The current version of minimal value types has some limitations. The HotSpot™ JVM, consisting of an interpreter and two just-in-time compilers (client and server compiler, also known as C1 and C2 compiler) for producing optimized assembly code for frequently executed code (see Section 2.5.3), only implements value types in the interpreter and the C2 compiler. The purpose of this thesis is to set the foundation for the missing value type support in the C1, to explore optimization possibilities and to provide a first implementation.

This thesis treats value types as immutable objects without an identity by reusing parts of the normal object implementation. Hence, the possibility to remove expensive allocations completely is enabled. Therefore, the main goal of this thesis is to apply optimizations which exploit the immutable and idetitlyless nature of value types (including only `final` fields) and the fact that they do not rely on escape analysis[1] (compared to normal objects) [37] to avoid heap allocations. An allocation is a slow operation inside the JVM. An object has to be stored on the heap by possibly invoking another expensive runtime call. Moreover, it entails the work of garbage collection which has to manage all objects and needs to keep track of the new active references to them. Removing such allocations completely not only benefits from a direct performance improvement but also avoids expensive heap accesses later by buffering the immutable field values in registers or on the stack. This buffering process is the key aspect of the entire allocation and field load removal approach performed in this thesis. The following list presents all achieved goals of the thesis in chronological order:

- **Bytecodes:** Supporting the new value type related byte codes `vdefault`, `vwithfield`, `vload`, `vstore`, `vreturn` and `getfield`[2] in the C1 compiler

[1] If a value type escapes a method scope, its field values cannot be changed anymore under any circumstances and thus the JVM does not need to analyze potential modification scenarios.

[2] Originally a separate `vgetfield` bytecode was used instead but was substituted by `getfield` during the progress of this thesis.

- **Performance Optimizations (Main Goal):** Eliminating allocations and field accesses on the heap for value types in the C1 compiler
- **On-Stack Replacement (OSR):** Supporting OSR-compilations for value types in the C1 compiler
- **Deoptimization:** Exploring C1 compiler support for deoptimization with live value types

Further details about these goals and how they were achieved are explained in the following chapters. For an improved readability throughout the thesis the term *value types* is used instead of *minimal value types* in the context of Java. This thesis not only contributes the design and a (first) implementation of value types in the C1 compiler but also serves as a documentation. Furthermore, it provides an evaluation of the implementation. The entire code of this thesis was written and tested in the context of the Intel x86-64 Linux architecture. The changes to the original starting point changeset 13528:f017570123b2 of project Valhalla from 21. August 2017[3] (the entire code was merged at that point due to a bug in the interpreter) is available as patch in `http://cr.openjdk.java.net/~thartmann/thesis_hagedorn/webrev/hotspot.patch`.

1.2 Structure of the Thesis

- Chapter 2 provides more technical information about value types, the Java language in general and the structure of a Java Virtual Machine and specific details of the HotSpot™ Java Virtual Machine, such as just-in-time compilation with two compilers or tiered-compilation.
- Chapter 3 presents related work done in other languages to support value types and compares it to Java. It also gives more information about project Valhalla and its other feature projects next to value types.
- Chapter 4 presents the design of value types in the C1 compiler on a high level. It differs between a simple, unoptimized version (relevant for the evaluation in Chapter 6) and the core version with optimizations, such as the main allocation and field load removal approach. The design decisions are illustrated with various examples.
- Chapter 5 maps the design decisions introduced in Chapter 4 to the the code implementation in the C1 compiler. An additional introductory section gives an overview of how the C1 compiler compiles a method and where the value type implementation starts to take action.
- Chapter 6 evaluates the optimized value type implementation and compares it to the baseline with normal objects and additionally to the unoptimized value type implementation.
- Chapter 7 gives a summary of the work and the results of this thesis and highlights opportunities for future work which can also be seen as limitations of the current implementation.
- The appendix provides a description of the timeline of the work done in this thesis in Section A.1, next to additional information, such as listings or a list of used tools.

[3]Changeset link: `http://hg.openjdk.java.net/valhalla/valhalla10-old/hotspot/rev/f017570123b2`

2 Background Information

This chapter presents more background information about the scope of the thesis in order to follow the work done. The first section gives an overview of the Java language and how it is distributed. Afterwards, Section 2.2 presents a short introduction of bytecodes. A more detailed discussion about why value types are important and how they are implemented in project Valhalla follows in Section 2.3. The general structure of a Java Virtual Machine (JVM) is presented in Section 2.4, while Section 2.5 describes in more detail the specific structure of the HotSpot™ JVM that is used for this thesis. The chapter ends with a brief overview of the SSA-Form in Section 2.6 that is used by the compilers of the HotSpot™ JVM.

2.1 The Java Language

Java is a general purpose, high-level, object-oriented, and platform-independent programming language. It was originally developed by Sun Microsystems and is nowadays owned by Oracle. Java pursued a new "Write Once, Run Anywhere" philosophy, unlike other conventional natively compiled (e.g. C/C++, Fortran or COBOL) or interpreted from source (e.g. SmallTalk, BASIC or Perl) languages at the time of its initial release. Java program files are first compiled to a platform-independent and easily sharable bytecode class-file which is then run by a Java Virtual Machine that interprets and compiles it to native code on a specific platform (see Section 2.4). [2]

Java was initially designed for interactive television but failed to materialize [12]. Only after that, it targeted the World Wide Web. Along with the first release of Java, Sun Microsystems provided *HotJava*, an own web browser which could run Java Applets [23]. Through the quick spread of the Internet, Java gained popularity and became one of the most popular languages. Nowadays, it is still the most used programming language (12.7%), followed by C (7.4%) and C++ (5.6%) according to the TIOBE index of September 2017 [72].

The core strength of Java is its object-oriented concept with a simple object model and the possibility to run programs anywhere. It has a design that is easy to learn and understand. The built-in garbage collection automatically takes care of the memory management. There is no need to manually free memory (like in conventional C/C++ programs) which is error-prone and can lead to severe bugs. As a statically typed language, Java already catches a lot of errors at compile time compared to programs written in a dynamically typed language which only fail at runtime.

Java programs do not run directly as native code on a computer's processor (as for example C or C++ programs do). They run on a JVM as an additional layer between the program and the hardware. This means that Java, especially in the earlier versions, lagged behind other natively compiled languages in terms of performance. However, JVMs have been continuously improved and nowadays just-in-time compiled code (see Section 2.5.3) and optimized native code are close in terms of speed. Java is significantly faster than dynamically typed scripting languages, like Perl, Python or JavaScript, by more than a factor of ten. [65]

The Java platform, known as Java Standard Edition (Java SE), implements all features described in the Java Language Specification[1] [56]. It provides three different packages: (i) the Java Runtime Environment, (ii) the Server Java Runtime Environment (Server JRE), and (iii) the Java SE Developmenet Kit (JDK). The JRE is designed for end users that only need to run Java on a desktop. It provides everything needed to run a Java application on the system. The Server JRE is used for deploying Java applications on servers and does not include browser integration. Finally, the JDK includes a complete JRE and supports the development, debugging, and monitoring of Java applications [44]. The platform specific Java Virtual Machine is part of the JRE and has its own (abstract) Java Virtual Machine Specification[2] [57] from which different implementations are available. This thesis works on the HotSpot™ JVM from Oracle which can be seen as the primary reference implementation (more details in Section 2.5). [45]

2.2 Bytecode

Java source code is first compiled to bytecode before its execution by the JVM. This is usually done by the integrated *javac*[3] compiler in the JDK (see Section 2.4). There are over 200 different bytecode instruction codes, each stored in a single byte. Some instructions require multiple bytes if they provide additional parameters. The bytecode execution involves an operand stack and a local variables array. The operand stack is used for the input and output of each bytecode operation. An example is shown in Listing 2.1. The bytecodes `iconst_1` and `iconst_2` push the integer value 1 and 2, respectively, onto the stack as an output. The `iadd` instruction pops two integer values from the stack as an input and adds them together. The output of the addition is pushed again

[1] Current version Java SE 8, State: 20. September 2017
[2] Current version Java SE 8, State: 20. September 2017
[3] `javac` is a Java-to-bytecode compiler which reads programs written in Java and compiles them into bytecode class files. It can also process annotations [46].

Listing 2.1: Example of using the operand stack for bytecode execution

```
1 iconst_1  # Push the int value 1 onto the stack
2 iconst_2  # Push the int value 2 onto the stack
3 iadd      # Pop two values from the stack, add them, push the result
```

Listing 2.2: Example of using the local variable array for bytecode execution

```
1 iconst_1  # Push the int value 1 onto the stack
2 istore_1  # Pop value from stack, store it into local variable 1
3 iload_2   # Push the int value from local variable 2 onto the stack
```

onto the stack. The local variable array is used to hold all local variables of a method including its parameters. Static methods parameters start at location zero, whereas instance methods have the zero slot reserved for the `"this"` reference. Thus, they have its parameters starting at location one. Bytecodes can load and store from the local variable array as shown in Listing 2.2. The `istore_1` bytecode pops an integer value from the stack and stores it in the local variable array at location 1 (called *local variable 1*). Finally, the `iload_2` bytecode loads the integer value stored at location 2 (called *local variable 2*) and pushs it onto the stack.

Bytecode instructions can be divided into categories such as arithmetic and logical instructions, object accesses, conditionals, type conversions, local variable accesses, or method calls. Additional bytecodes were introduced for value types, which are described in the next section. More information and a complete description of the bytecodes available in Java SE 8 can be found in [57].

2.3 Value Types

2.3.1 Why Value Types?

Back in the mid-nineties, when Java was born, the magnitude between the cost of an arithmetic operation and a memory fetch was comparable. However, nowadays the memory cost is clearly dominating by a factor of several hundreds compared to arithmetic costs [24]. Therefore, Java lacks the possibility to represent efficient small immutable aggregate types without identity as described in Section 1.1. The type system of the current Java release[4] only offers the two aggregate types: *classes* (heterogeneous) and *arrays* (homogeneous) [20] coupled with an inherent identity. All Java objects are heap-allocated. Therefore, each newly allocated object gets a new, unique address assigned in the heap memory and a well-defined identity. These objects are allowed to be compared independently of their actual contents. For example, two `String` objects could contain the same value "HelloWorld" but still have different identities. They can be compared using the `==` operator. On the contrary, two `int` variables which both contain the same value cannot be

[4] Java SE 8 in September 2017

distinguished. Comparing them with `==` always gives the result `true`. Therefore, identities enable the important concept of mutability of object states while remaining the same intrinsic object. They can be accessed and possibly altered through one or more pointers (unless dead) pointing to them through references in Java. [20]

This often unneeded identity for objects in many programming idioms has a significant cost compared to their actually used payload. In terms of performance, a field of an object requires a pointer traversal to the corresponding memory location. Moreover, objects additionally entail the work of garbage collection limiting the performance. Many optimizations are conservative with objects, as they could escape a method body and be modified, even if they are immutable. In terms of memory footprint, each object consists of two extra words[5], a mark word and a klass pointer, representing the object header. The mark word encodes essential information like states for biased locking (used for synchronization) [4] or for garbage collection [16]. The klass pointer points to metadata about the object, such as information about the object layout and its behavior [60]. Array object headers contain an additional word to store its length. For example, each object of a class `Point` that contains two `int` fields for the x and y-component requires at least twice as much space as the actual payload of eight bytes would need: For a 32-bit architecture, the object header adds an extra eight bytes of heap space plus a reference of four bytes to access it resulting in 12 bytes of additional space (for a 64-bit architecture it is up to 24 bytes). Storing the `Point` objects in an array makes things even worse. Besides having an additional four or eight bytes for the array length, the inherent advantage of locality in the array is lost. An iteration needs a pointer dereference for each `Point` object which could possibly be scattered all over the heap [19]. Without value types, programmers have tried to bypass object creation and its associated cost by using primitive types. For example, an array of `Point` objects could be replaced by creating two separate `int` arrays for the x- and y-component, where a point P_i is defined as *(x_arr[i]/y_arr[i])* for a suitable i. However, this suffers from a bad maintainability and a lost of encapsulation [20].

Therefore, the need for value types as a third type in the type system becomes evident, especially for performance critical programs. This opens possibilities for many optimizations due to their immutable nature. Compared to the normal object concept, value types become more restrictive:

- Immutability
- No identity
- No object header
- No object semantics like locking or synchronization
- No inheritance and thus no dynamic dispatch

The resulting value types can be described as user-definable primitive types or lightweight classes with value semantics [67].

[5] On a 32-bit architecture the word size is four bytes and on 64-bit architectures eight bytes.

2.3.2 Value Types in Java

The project Valhalla is an experimental project for the exploration of advanced language features for Java[6] such as value types [3] in the HotSpot™ JVM. More information about Valhalla can be found in Section 3.2. To use the described value types in the following paragraphs, the Valhalla repository has to be cloned, built and enabled by setting certain flags. More information can be found in [21].

There are currently two possibilities to work with value types in Valhalla: (i) using value-capable classes and method handles or (ii) generating value type bytecodes with the experimental `javac` version of Valhalla[7]. The first approach can mark classes with a special annotation `@DeriveValueType`. From this marked class the JVM generates a normal class (value-capable class) and a value type class as a source for a value type (called *derived value type*). The method handle API[8] can be used to manipulate value types, perform load and stores to fields or invoke methods. However, this limits the possibilities for optimizations due to type erasure of method handles [34]. Therefore, the second approach was chosen with the help of the experimental version of `javac` (compared to the normal `javac` version in (i)) which supports an own experimental syntax for creating value type classes and value types from it. This syntax and the value type related bytecodes do not commit to the language syntax and bytecode design. Those temporary decisions can change any time during the development process. For example, the bytecode `vgetfield` was present at start of the thesis but was removed and replaced by the `getfield` bytecode in the meantime. [17]

The experimental `javac` compiler defines a special `__ByValue` modifier for generating a value type class which has to fulfill certain requirements[9]: All associated fields must be immutable by declaring them `final`. The class itself is not allowed to be extended and hence is also declared `final`. Value types have no identity and currently cannot be compared with the `==` or `!=` operator[10]. There is also no special `null` value for value types. The default value at the creation of a value type represents an instance whose fields are set to their respective default values. This also forbids a recursive (cyclic) definition. For example, a field cannot reference the declaring value type class since the initialization process of this field would involve an infinite loop. In terms of value type allocations, the JVM just needs to preserve the value semantics. The decision of how to store them (on the heap, on the stack or in registers) is specific to the JVM implementation and is not restricted. The JVM might try to avoid value type allocations completely whenever possible (main part of this thesis for the C1 compiler) or to keep the fields of value types in registers (not part of this thesis and left as future work for the C1 compiler) to get good performance. A new value type instance can be created by replacing the `new` keyword by the special `__MakeDefault` modifier inside a method declared with the special `__ValueFactory` modifier in a value type class. A dummy constructor has

[6] Those are not available in the Java product release.

[7] A third option would be to write bytecode manually by hand.

[8] An API to get a method reference (i.e. a method handle) that can directly be executed. More information can be found in [33].

[9] The value-capable classes from the first approach in the last paragraph must also fulfill these requirements.

[10] State: 21. September, 2017

to be added which assigns every field a value. It is completely ignored and is only there to avoid an error from `javac`. The need for this constructor will certainly be removed at a later stage for a possible release. A field can be assigned in any method with a prefixed `__ValueFactory` modifier inside the value type class. However, a single assignment also creates a new value type instance since they are conceptually immutable. An example of a valid value type class `VTPoint` illustrating its constraints together with a corresponding creation of a value type instance can be found in Appendix A.5 in Listing A.1. In class-files value types are described by an own "Q-type" descriptor. For example, the method `set` in Listing A.1 has the signature `(QVTPoint;II)QVTPoint;`.

This thesis supports the following value type bytecodes generated by `javac` for the C1 compiler:

- **vdefault:** Creates a value type instance with all fields initialized to their default values
- **vwithfield:** Creates a new value type instance by copying the existing value type instance and reassigning one of the fields specified in the bytecode input parameter with the value provided in the second input parameter
- **vload:** Loads a value type instance from a local variable
- **vstore:** Stores a value type instance into a local variable
- **vreturn:** Returns a value type instance from a method
- **getfield:** Adapts the existing `getfield` bytecode to fetch a field from a value type instance

A description of all value type related bytecodes and more information about the changes in the JVM specification for value types can be found in [53]. Value type arrays and boxing operations are not supported by this thesis and are left as future work (see Section 7.1). [17, 67, 68]

2.4 The Java Virtual Machine

This section first gives an overview of the general structure of a Java Virtual Machine (JVM) as specified in the Java Virtual Machine Specification [57]. The actual implementation of a JVM, like the HotSpot™ JVM (see Section 2.5), is not part of the specification. Furthermore, it is also not defined how the objects are internally represented. Moreover, the specification does not state how the bytecodes are executed. They could be interpreted (see Section 2.5.2), compiled to native code before execution (see Section 2.5.3), or be executed in a mixture of both (see Section 2.5.4). This implementation freedom enables sophisticated optimizations. The specification defines just sufficiently enough details to guarantee compatibility and portability while it is abstract enough to cede implementation details to a vendor of a JVM.

Figure 2.1 first shows how the JVM acts as a black-box in the process of executing a Java program. The source code of every Java program is contained inside one or multiple `.java` files. Those are compiled by the `javac` compiler, available as part of the JDK, into platform-independent bytecode

instructions. From each `.java` file a corresponding `.class` file with bytecode instructions is generated in a well-defined format [31]. For this thesis, the experimental version of `javac` from the project Valhalla is used which additionally processes the value type syntax explained in Section 2.3 to generate all value type related bytecodes described in [53] and in Chapter 4 and 5 for those specific in this thesis. Finally, the HotSpot™ JVM gets a `.class` or `.jar` file (created by the Java Archive Tool [36] to bundle multiple `.class` files together) input which is then interpreted or compiled to platform-specific native code and run.

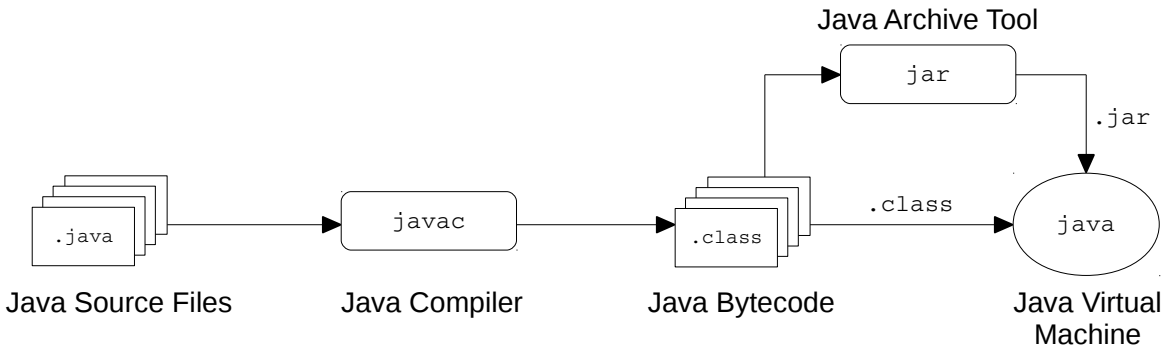


Figure 2.1: The process of executing a Java program

The bytecode input to the JVM is completely independent of the source language. Thus, the JVM is also capable to run other languages which provide a compiler to produce bytecode in form of class-files. For example, Oracle ships *Nashorn*[11], an engine which can interpret JavaScript code. Its compiler produces bytecode which can be executed by the JVM. For more information see [54, 43].

Figure 2.2 gives an overview of the internal structure of a JVM as specified in the Java Virtual Machine Specification [57]. The path of the program to be run (in a `.class` file) is first passed to the class loader subsystem. In the next step, the class loader loads all required classes used in the program in three stages: (i) In the loading stage, it tries to load the bytecode of the classes into memory. This is done by two different types of class loaders: the bootstrap class loader and the user-defined class loaders. The bootstrap class loader is responsible to load the required basic classes from the Java library like `String` or `Date`. In the JDK those class-files reside in the rt.jar file [30]. In addition, there are many user-defined class loaders. One of them is the application class loader which loads the class-file of the program as specified in the class path or `CLASSPATH` variable. (ii) In the linking stage, the loaded bytecode gets verified (and checked if it meets the required format). This verification is an important preventative process to ensure that the JVM does not get corrupted. All necessary data structures, like the method tables, are allocated together with the static storage. Static fields are allocated and initialized with their appropriate default values (see [51]). (iii) In the initialization stage, all static initializers are executed which can set the class variables to their specified initial values.

[11] A new engine in Java SE 8 which replaced the old *Rhino* engine of Java SE 7.

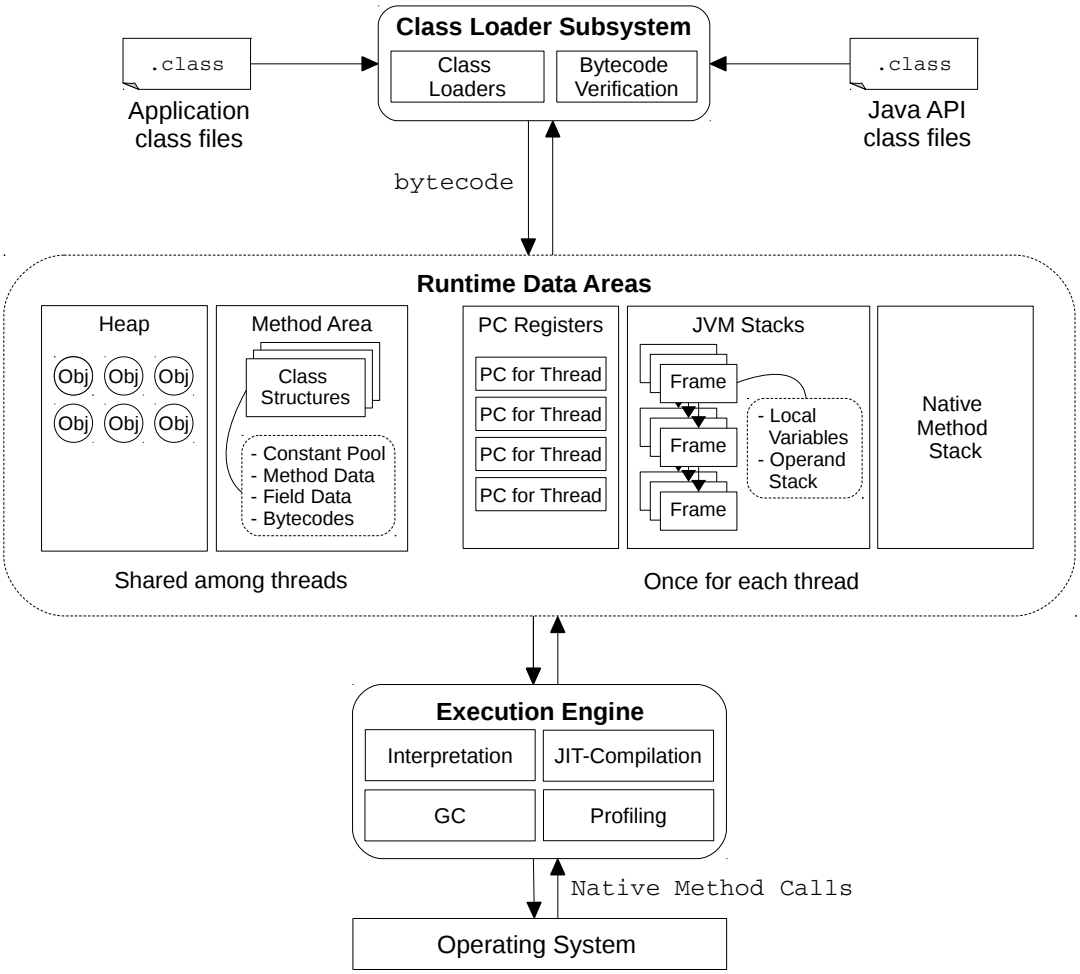


Figure 2.2: The internal structure of a JVM as specified in the Java Virtual Machine Specification

The JVM uses five different run time data areas during program execution: heap, method area, program counter (PC) registers, JVM stacks, and native method stack. The first two are only created once and are shared among all threads, while the others are created per-thread.

- The heap memory area stores the object data of all created class instances and arrays. The stack can only hold references to the heap and not the objects itself. The heap storage is automatically managed by the garbage collector and is never explicitly deallocated by the programmer as it is done in languages such as C/C++. This avoids memory leakage bugs.
- The method area stores meta data for each class. It contains the runtime constant pool to cache field and method references, field data and method data, such as names, type information, and the bytecode code for each method.
- A PC register contains a pointer to the next instruction to be executed. There are possibly many active JVM threads at once. Therefore, each thread has an own PC register.

- Each JVM thread has a JVM stack which is only visible to the thread itself. It contain frames which corresponds to the current execution of a JVM thread. For each (non-inlined) method invocation, a new stack frame is pushed and popped again on return. Each frame contains information including the parameters, local variables and the operand stack for the execution of bytecodes.
- A JVM can load and invoke `native` methods (methods written in a different language from Java, such as C/C++). Each thread has its own native method stack (also called "C stack" in connection with the *Java Native Interface* (JNI) which enables Java to use native methods written in C/C++ or assembly code) for native method invocations. However, the specification does not require the support of native method loads and the corresponding need for native method stacks in a JVM.

The execution engine executes the next instruction to be ready once the runtime data areas are loaded. Depending on the implementation of the JVM, the execution engine can either interpret, directly compile and then run the instructions or first interpret and then compile the bytecodes dynamically (just-in-time). For just-in-time compilation, it is essential to gather profiling information to decide which methods to compile and to guide optimizations. Garbage collection is also an important part of the execution engine. The algorithms for garbage collectors are left to the implementor of a JVM and are not defined by the specification.

2.5 HotSpot™ Java Virtual Machine

The HotSpot™ Java Virtual Machine (JVM) can be seen as the reference implementation of the Java Virtual Machine Specification [57]. It implements the abstract model described in Section 2.4. This section gives an overview of the HotSpot™ JVM and explains the execution engine which is very specific to HotSpot™ and possibly implemented differently in other JVMs. The relevant parts for this thesis are presented in more detail in order to follow the discussion about the design and implementation later in Chapter 4 and 5, respectively.

2.5.1 Overview

The HotSpot™ JVM is the core part of the Java Runtime Environment (JRE), which also provides base libraries, such as *lang* or *util*, and is responsible for executing Java programs. It is mainly written in C/C++ and contains around 250'000 lines of code distributed in over 1500 header and source files [55]. There are many command line options available to enable or disable additional functionality, print runtime information or set environment variables. This thesis adds some custom flags and modifies existing flags as described in Appendix A.3. A full list of all flags can be found in [38]. The HotSpot™ JVM is available on various platforms and operating systems including Linux and Windows on Intel x86 and x86-64, Linux on ARM, Sparc, and PPC.

The architecture of the HotSpot™ JVM follows the one described in Figure 2.2 in Section 2.4. In a first step, the initial class gets loaded, initialized, and verified by the class loader. Afterwards, the `main` method of the initial class is invoked. The class loader dynamically loads and verifies additionally required classes in the bytecode and their class-files the first time they are referenced at runtime. Value type related bytecodes in the project Valhalla are currently not verified[12]. Before the program execution starts, the code of a method is loaded from the method area. Up to Java SE 7 the HotSpot™ JVM used a memory area called *permanent generation*. In Java SE 8 it is replaced by a native memory area called *metaspace* [22]. The main difference is that its size is not fixed anymore and automatically increases compared to the fixed-sized permanent generation [61].

The execution of a method always starts in the interpreter. The HotSpot™ JVM gathers profiling information about how often each method and each loop in a method is executed at runtime. If a certain threshold is crossed, the method is considered as *hot spot* (hence the name *HotSpot* for the JVM) and scheduled for compilation. The JVM uses one of the two dynamic just-in-time compilers (depending on the used settings and the currently tracked profile of a method) which transforms the bytecode of the method to optimized machine code (see Section 2.5.3). In the upcoming Java SE 9 release, there is also a possibility to compile Java code before launching the virtual machine. This feature is called *ahead-of-time compilation*. More information can be found in [62].

The following sections gives an overview of the HotSpot™ JVM specific components of the execution engine shown in the abstract structure of a JVM in Figure 2.2 in Section 2.4.

2.5.2 Interpreter

A method in a program is always interpreted first in the HotSpot™ JVM (including the `main` method)[13]. The current interpreter of the HotSpot™ JVM is template-based. It uses a template table for each architecture which contains a template (i.e. a description) for each available bytecode. Those templates provide the platform specific assembly code for the corresponding bytecode. The interpreter uses a bytecode pointer to point to the current bytecode in the bytecode stream to be executed. This bytecode can then be looked up in the template table to apply the corresponding template code. The interpreter can call into the *runtime* to get C/C++ support for more complex operations that cannot or should not be directly dealt with in assembly code. For example, in some cases, object creation needs to call into the runtime of the JVM to setup everything which would be too complex with an assembly template only. The interpreter can be compared to a simple processor: It loads each bytecode, looks the corresponding template up, and then executes the assembly code together with the required runtime calls.

[12] State: September 2017

[13] All methods need to be executed at least once in the interpreter before switching to (just-in-time) compilation.

There is also a C++ interpreter available which is purely written in C++ (without emitting assembly code directly). However, it is slower than the template-based approach according to [35]. Among other reasons, it has to compare the current bytecode instruction with all bytecodes but one in the worst case. Therefore, the template interpreter is set to be used as a default.

The interpreter analyzes the running code and performs some basic profiling, such as keeping track of the types of local variables. This runtime information, which would not be available if the code was compiled statically, is essential for later applying sophisticated optimizations in the compilers. The interpreter also detects the critical and hot code spots by using counters for method invocations (method entries) and loop repetitions (back-branches in loops). Once a certain threshold is reached, the method is scheduled for just-in-time compilation, which is explained in the next section in more detail.

2.5.3 Just-in-Time Compilation

Running a method with the interpreter only is slow. Each bytecode requires a template lookup which invokes a template consisting of a fixed set of machine instructions. Frequently called methods are directly compiled and optimized to machine code to replace interpretation on each invocation. This compilation is performed dynamically during execution. This process is called *just-in-time compilation* or *dynamic compilation*.

Almost all programs spend most of their execution time in a small part of the code [35], for example, in one or more long-running methods or loops. Therefore, compilation thresholds attached to each method are important since the compilation of every method has a negative impact on the overall program performance. The gained execution time for compiled methods that are only executed once or a few times and do not contain long-running loops often cannot compensate the additional time spent for the compilation itself compared to using the interpreter only. Thus, just-in-time compilation can spend more time on optimizing the few hot spots in the code instead of compiling everything directly.

Figure 2.3 shows the different method state transitions for just-in-time compilation in the HotSpot™ JVM. Each method is first executed in the interpreter. Besides gathering profiling information, the interpreter already loads the encountered classes used by the method. Moreover, additionally called methods are also known before switching to compilation. Depending on the configuration, the method gets either scheduled for the *client compiler* (C1) (see Section 2.5.3.1) or for the *server compiler* (C2) (see Section 2.5.3.2). Prior to Java SE 7, a user of a program had to decide to run a program either with the client compiler (client VM), suitable for interactive client applications, or with the server compiler (server VM), suitable for long-running server applications with a good peak performance. Using a mix of both compilers was not possible. This changed with the introduction of tiered-compilation with Java SE 7 (see Section 2.5.4). This approach first compiles and instruments a method with the C1 compiler. It continues to gather profiling infor-

mation, counting loop back-branches and method entries. If the method turns out to be used even more extensively, it is finally compiled with the heavily optimizing C2 compiler. The created code is stored in the code cache to avoid recompilation of already compiled methods [49].

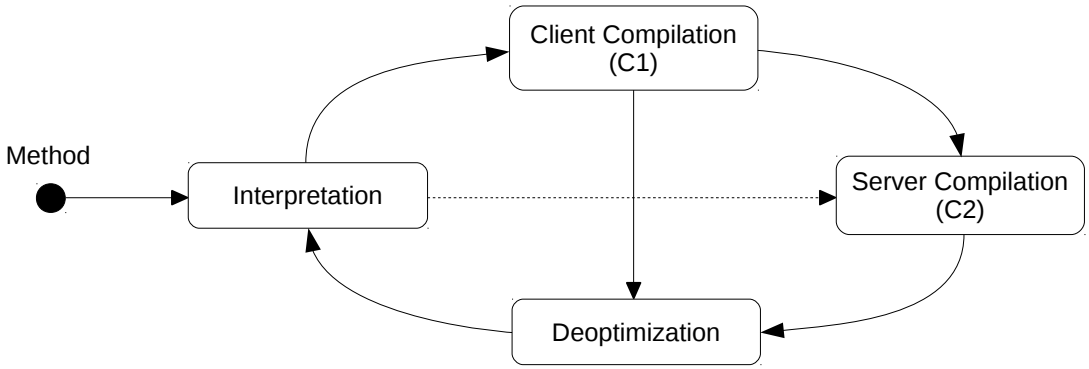


Figure 2.3: The cycle of compilation in the HotSpot™ JVM

The dynamic compilers, especially C2, use optimistic assumptions about the code. For example, a branch that was never taken, is not compiled or a compiled method is optimistically inlined based on the available information (which might be incomplete). If one of those assumptions later turns out to be wrong, for example, if a new subclass is loaded or the branch condition suddenly evaluates to a different result than expected, the compiled method becomes invalid and is discarded. The control is transferred back to the interpreter. This process is called *deoptimization*. The compiler must provide enough information about certain points in the compiled code for restoring the state in the interpreter later. Therefore, the compiler stores meta data for compiled code such as local variables, the operand stack or the current bytecode index (see Section 5.8). The semantics of the code must not change. Even though this reconstruction process from compiled to interpreted code is complex, it enables many performance improving optimistic optimizations.

Methods are normally compiled once the counter for a method entry overflows. However, there are also situations where a method is only called a few times but contains a long-running loop and possibly never even exits. The loop back-branch counters identify these cases. Once such a threshold is crossed, the running method is replaced by a compiled version. This process is called *On-Stack-Replacement* (OSR). The interpreter calls into the runtime which stores the necessary information about the interpreter frames such as local variables. After the runtime is finished, it jumps to a special entry, called *OSR-entry*, just before the loop condition. The compiled frame is created and the execution resumes in the loop.

The next two sections describe the two just-in-time compilers in more detail with special focus on the C1 compiler which is relevant for this thesis.

2.5.3.1 Client Compiler (C1)

The value type approach of this thesis is implemented in the context of the C1 compiler. More concrete details about the actual implementation are explained in the Chapter 5. This section only intends to give an overview of the compiler.

The C1 compiler aims for fast compilation which matters to the entire program performance while producing compiled code with a good execution time. This allows the C1 compiler to be used for interactive applications, where slow compilation affects their responsiveness. It also helps to increase the performance at the start of an application. This is especially useful in combination with tiered-compilation. During the entire compilation process, the memory footprint is kept low.

Methods are compiled as a whole in multiple phases as shown in Figure 2.4. In a first step, the bytecodes of a method are analyzed by the machine-independent front-end to build a *high-level intermediate representation* (HIR) by using abstract interpretation. There are already some (local) optimizations performed while generating the HIR in the parsing stage, such as *constant folding* or method inlining. The core parts of the value type optimizations in this thesis are also done during this generation phase[14]. The HIR consists of an explicit *control flow graph* (CFG). Its nodes are represented by *basic blocks* (i.e. a sequence of instructions with no branches or jump targets). A basic block can have multiple predecessors or successors and is connected by a link from the last instruction in the block to the first instruction of a successor block. The resulting HIR is in *static single-assignment (SSA) form* (see [66] and Section 2.6). This simplifies some global optimizations, such as *null-check elimination* [29], *conditional expression elimination*, or *global value numbering* [63]. Those are applied once the HIR is generated by continuously iterating over it. The largest part of the implementation in this thesis is done in the HIR since it is easier to apply the optimizations.

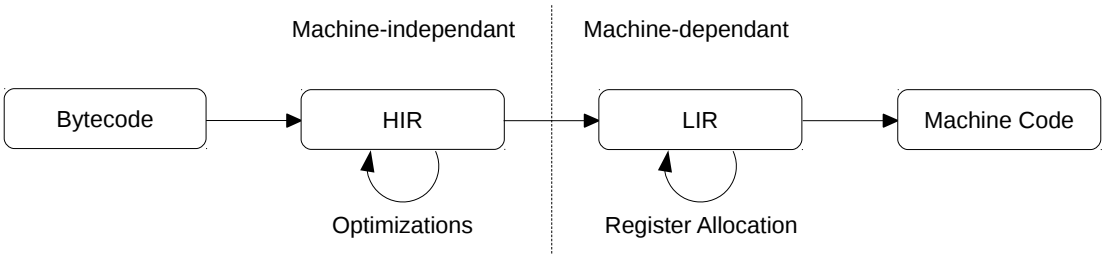


Figure 2.4: Structure of the client compiler (C1) of the HotSpot™ JVM

In the second phase, the platform-dependent back-end generates a *low-level intermediate representation* (LIR) from the HIR. The information of the CFG data structure from the HIR can be directly used. The LIR looks similar to three-operand assembly code. However, it also contains some higher level instructions, for example, for object allocation. The LIR uses virtual registers

[14]Some cleanup steps need to be done just at the end of the generation phase when everything is parsed (see Section 5.6) and some additional adaptations need to be done for the LIR generation (see Section 4.1 and 5.2).

(containing the local and temporary values) which are mapped to physical registers by the linear scan register allocator [73]. Some low-level optimizations work better on the LIR than on the HIR. For example, all operands that need a register in the LIR are visible, which can be exploited by the register allocator. In the last phase, the machine-dependent assembly code is generated. The code generator traverses the LIR and emits corresponding assembly instructions.

The C1 compiler also supports profiling to eventually switch to C2 compilation. This is similar to the profiling steps done in the interpreter. Additional profiling instructions can be added to the LIR depending on the setup of tiered-compilation (see Section 2.5.4). More information about the C1 compiler can be found in [71].

2.5.3.2 Server Compiler (C2)

The server compiler (C2) aims for peak performance and is suitable for long-running server applications. The resulting code is highly optimized. The C2 compiler does not focus on keeping the actual compilation and the startup time low. This allows the compiler to perform many additional and more extensive optimizations on top of those applied in the C1 compiler, such as loop optimizations or optimistic inlining, which can be very time-consuming.

The C2 compilation process of a method includes the following steps: parsing bytecodes, optimizations, instruction selection, global code motion, register allocation, peephole optimizations, and code generation. The C2 compiler parses the bytecode into an *intermediate representation* (IR) in SSA-form which is used through all steps of the compilation (compared to the C1 compiler which uses two intermediate representations). The data structure for the IR [5] is more complex, as it contains control dependence and data dependence in contrast to the C1 compiler that uses a CFG with basic blocks which contain a sequence of instructions. In the IR of the C2 compiler, edges point from the usage of a value to its definition like in a program dependence graph [13]. This simplifies the application of global optimizations.

As for the C1 compiler, some local optimizations, such as constant folding, are preformed during the parsing phase. Others, such as loop unrolling, have to be performed after parsing since they might not have all the required information otherwise. After the parsing stage, machine-independent optimizations, like dead code elimination or global value numbering, are applied on the IR graph by continuously iterating over it until a *fixed-point* is reached (i.e. no changes after an application of the optimizations) and no optimization can be further applied. This can be very time-consuming and thus is not suitable for interactive or startup dependent applications. The C2 compiler applies a slower graph coloring algorithm for register allocation than used in the C1 compiler. However, it produces better results with less spills in practice. The IR graph is eventually converted into non-SSA-form and iterated over to emit machine-dependent assembly code. The C2 compiler has been completely disabled during this thesis, as it focuses on the C1 compiler only. More information about the C2 compiler can be found in [27].

2.5.4 Tiered-Compilation

In the earlier days of the HotSpot™ JVM, only one compiler could be chosen to be used during the execution of a program. This changed with the introduction of *tiered-compilation* in Java SE 7. The advantages of the C1 and C2 compiler can be used together in a mixed execution. The C1 compiler guarantees a fast startup while the C2 compiler overtakes later to guarantee peak performance (see Section 2.5.3). However, tiered-compilation generates more compiled code (from both compilers) that can take up more space in the code cache. Tiered-compilation can be enabled by the `-XX:+TieredCompilation` and disabled by the `-XX:-TieredCompilation` flag. In Java SE 7 it is disabled and since Java SE 8 enabled by default.

The HotSpot™ JVM does not only use three *tiers* (interpreter, C1, and C2 compiler) but five. Those are also called *execution levels* and are shown in Table 2.1. The C1 compiler offers three different levels of execution depending on the amount of profiling. A method always has to be executed in the interpreter at level 0 first. If a method becomes hot, it usually gets scheduled for compilation in the C1 compiler at level 2 or 3 depending on the policy [52]. A key factor in this decision is the current queue length of compilation tasks for the C2 compiler. Level 3 uses more extensive profiling and is therefore about 30 % slower than level 2 [11]. If the C2 compiler queue has a heavy load, the method is rather compiled with level 2 to avoid unnecessary time spent at level 3. Once a method reaches its threshold again, it is finally scheduled for C2 compilation at level 4 which does not collect any profiling information anymore. If a method turns out to be small and trivial, it is compiled at level 1 without any profiling information. In this scenario, the C2 compiler would not be able to produce a more optimized version of the method and thus is not invoked. The most common transition is from level 0 to the C1 compiler at level 3 and then to the C2 compiler at level 4. In case of a deoptimization at any level, the execution jumps back to the interpreter at level 0. In this thesis, the flag `-XX:TieredStopAtLevel=1` is used to switch only from the interpreter to compilation at level 1 in the C1 compiler since the C2 compiler is disabled completely and no profiling is required.

Table 2.1: The five tiers (execution levels) of tiered-compilation

Tier	Execution	Profiling
Level 0	Interpreter	yes
Level 1	C1	no
Level 2	C1	yes (limited)
Level 3	C1	yes (full)
Level 4	C2	no

2.5.5 Garbage Collection

To free up heap space, the HotSpot™ JVM can halt the current program execution to perform garbage collection. Together with compilation, the garbage collection determines peak performance. There are three different kinds of garbage collectors available: (i) a serial collector [39], (ii) a parallel

collector [40], and (iii) two mostly concurrent collectors [41]. The serial collector halts the current execution of a program. It uses only a single thread for garbage collection and thus works best on a single processor machine. The parallel collector, also called *throughput collector*, is similar to the serial collector but works with multiple threads for a better performance. The two mostly concurrent collectors, the *Concurrent Mark Sweep Collector* (CMS) and the *Garbage-First Garbage Collector* (G1), aim to shorten the pause times. The CMS collector is suited for applications that require short pauses. It shares the processor resources with the application while it is still running. The G1 collector, on the other hand, is suited for large memory usage. It aims for high throughput and low pause times. More information about garbage collection can be found in [42].

2.6 SSA-Form and Phi Nodes

This section provides a brief overview of how SSA-based code is constructed and highlights the key feature of *phi functions*. These concepts are important to later follow the discussions in Chapter 4 and 5 about the design and the implementation, respectively.

An SSA-based intermediate representation enforces that each variable has exactly one assignment. Consequently, each use has exactly one point of definition. This simplifies and improves many compiler optimizations, such as constant propagation or dead code elimination. A simple example of translating code into SSA-form is shown in Listing 2.3. Listing (a) shows a code snipped with two variables x and y. The variable x is assigned twice and thus is not in SSA-form. In Listing (b), the variables are replaced with unique version numbers for each assignment. Each variable has now only one unique definition and therefore is in SSA-form.

Listing 2.3: Simple example of SSA-form translation in pseudo-code

(a) Original

```
1 x = 1;
2 x = x + 2;
3 y = x;
```

(b) SSA-form

```
1 x_1 = 1;
2 x_2 = x_1 + 2;
3 y_1 = x_2;
```

The SSA construction gets more complicated when branches are included and merged again. Listing 2.4 shows an example where the variable a is set to a different value in one branch on line 4 if the condition is true but remains unchanged otherwise. Translating the (non-SSA-form) code snipped in Listing (a) in a straightforward way to a SSA-form as shown in the previous example fails on line 6 in Listing (b). There is no way to statically know if a_1 or a_2 should be used. Therefore, a special *phi function* (also denoted as φ function), as shown on line 6 in Listing (c), is introduced to merge a_1 and a_2 and assign it to a new variable a_3 satisfying the constraint of SSA-form. Such a phi function can be seen as a "magic" function which guarantees to choose the correct version depending on the taken path at runtime. In the example in Listing 2.4 (c) a_3 is set to a_1 if b_1 equals 0 and to a_2 otherwise. The code snippet is now presented in valid SSA-form.

Listing 2.4: Example of SSA-form translation with branches in pseudo-code

(a) Original

```
1 a = 1;
2 b = read();
3 if (b == 0) {
4   a = 2;
5 }
6 x = a;
```

(b) SSA-form failing

```
1 a_1 = 1;
2 b_1 = read();
3 if (b_1 == 0) {
4   a_2 = 2;
5 }
6 x = a_?? ; // Cannot use a_1 nor a_2
```

(c) SSA-form with φ functions

```
1 a_1 = 1;
2 b_1 = read();
3 if (b_1 == 0) {
4   a_2 = 2;
5 }
6 a_3 = φ(a_1, a_2); // φ function
7 x_1 = a_3; // Merge resolved
```

A basic block entry of a control flow graph can contain multiple phi nodes. They are not only used for branches with if/else but also for loops. An example is shown in Listing 2.5. The code in Listing (a) is translated into SSA-form shown in Figure (b). The condition is located in an own basic block *B2* since it is used as jump target from the basic block *B3* containing the loop-body. The phi functions for a and b are inserted at the entry of basic block *B2* as it merges the basic blocks *B1* and *B3*. This example also shows that the phi function for a is not necessary. The operands of a_2 are the same and thus the entire phi function could be removed and a_2 be replaced by a_1 in basic block *B3* and *B4*.

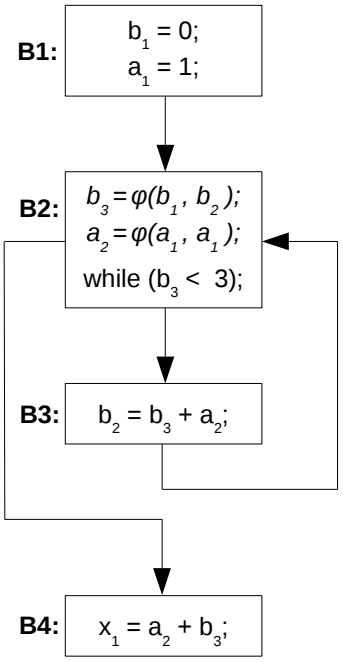
After an intermediate representation in SSA-form has applied all its SSA-based optimizations, it is *destructed* before code generation. The destruction process replaces all phi functions since they are not real methods which could be invoked from code and thus cannot be executed. Therefore, each phi function is replaced by additional instructions at the end of the predecessor blocks to properly assign the variable to which the phi function was assigned to. A simple example is shown in Listing 2.6. Listing (a) presents a code snippet in SSA-form while the destruction process is shown in a control flow graph view in Figure (b). The phi function for a_3 in basic block *B4* is replaced by pushing an assignment $a_3 = a_2$ into basic block *B2* and an assignment $a_3 = a_1$ into basic block *B3*, respectively. This assignment process is straightforward for all Java variables but requires additional instructions for value types (see Chapter 4). More information about the SSA-form in general can be found in [7, 14, 66].

Listing 2.5: Example of SSA-form translation containing a loop

(a) Original

```
1 b = 0;
2 a = 1;
3 while (b < 3) {
4   b = b + a;
5 }
6 x = a + b;
```

(b) SSA-form in control flow graph view

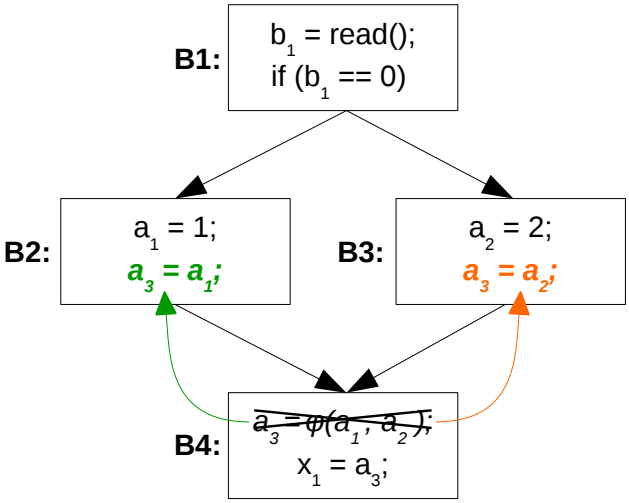


Listing 2.6: Example of replacing phi functions in SSA-form

(a) SSA-from with φ functions

```
1 b1 = read();
2 if (b1 == 0) {
3   a1 = 1;
4 } else {
5   a2 = 2;
6 }
7 a3 = φ3(a1,a2);
8 x1 = a3;
```

(b) SSA-form destruction in control flow graph view



3 Related Work

This chapter shows related work done in other languages to support value types or value type-like implementations compared to the approach in Java. It follows a description of the experimental project Valhalla and its related work.

3.1 Value Types in Other Languages

Scala: Scala proposed value types in SIP-15 [26]. The value types are still in an early stage of the process and are not yet implemented. The goal is to avoid the allocation of runtime objects and to provide new numeric classes, such as unsigned ints, without the need of boxing. This is done with user-defined *value classes*. However, the proposal has some restrictions for writing such a class:

- A value class extends from `AnyVal` and only allows to define a single `val` parameter but no additional fields such as fixed values (with `val`) or variables (with `var`).
- A value class can only extend *universal traits* which explicitly extend from `Any`. It cannot extend traits which extend from `AnyRef`.

Compared to value types in Java, they are very restricted in Scala. Moreover, SIP-15 also states that it is not sure if the suggested implementation will work since it is too complicated.

C#: C# also provides a type called *value types* (next to reference types). These contain the values directly like primitive types in Java. The C# value types are used in association with structs and enumerations. However, they only define that they are no reference types and thus can also be put directly on the stack (instead of the heap). An implicit immutability is no requirement compared to value types in Java. Therefore, C# cannot apply the same sophisticated optimizations as in Java, even if it could enforce immutability manually. The compiler stays conservative. [6]

C++: Programmers in C++ can directly use pointers to the heap. Therefore, they have the choice to store a variable directly on the heap or on the stack. It does not matter if it is an object or just a value like an integer. Immutability can be reached by using `const` fields and `const` methods. However, this is just a simulation of value types. It is not provided as standard type for C++ and thus the compiler cannot apply the same value type optimizations as in Java. Moreover, due to the nature of the language, the customary assigned value type property can also be bypassed by casting away the constness for example. [28]

Go: The programming language Go (also called *golang*) offers structs (and interfaces) which are value types. These are stored in memory and directly contain the value without an additional object header, as needed for Java objects. Everything is passed by value which also means that, for example, structs are copied to a function. Go additionally offers the possibility to apply a method to a pointer to a struct. However, this allows a modification which is not allowed for value types in Java. There is no concept of immutability in Go and thus its value types are not real value types in the sense of Java. [9]

3.2 Project Valhalla

The purpose of project Valhalla is to explore advanced language features for Java and is sponsored by the HotSpot Group [55]. The goal of Valhalla is not only to gain performance but also to address safety, abstraction, encapsulation, expressiveness, maintainability, and compatible library evolution [25]. Apart from value types, there are other important features like *generic specialization* or *enhanced volatiles* which are about to be explored or already completely delivered.

Generic Specialization: This is an enhancement feature to support generic types and interfaces over primitive types. The usage of a generic type requires an extension of the type `Object`. Thus, it is not compatible with primitive types which can only be used with boxing. For example, using a list of integers needs to be defined as `List<Integer>` and cannot be directly written as `List<int>`. The additional performance and memory cost by the boxing operations becomes even more problematic with value types. Thus, this feature tries to directly support generics over primitive types. A specific release version is not yet targeted. More information can be found in [48].

Enhanced Volatiles: This is an enhancement feature that defines a standard means for various atomic operations from the `java.util.concurrent.atomic` package [50] and operations from the `sun.misc.Unsafe` API on array elements and object fields. Main goals are safety, not to corrupt the JVM, and integrity, to use the same access rules as for the `getfield` and `putfield` bytecode and prohibiting an update of a `final` field. Others are performance, to match the same characteristics as for the `sun.misc.Unsafe` operations, and usability, to provide a better usage as with the `sun.misc.Unsafe` API. This feature was released with the newest Java SE 9 version. More information can be found in [47].

4 Design

This chapter describes the design of value types in the context of the C1 compiler (see Section 2.5.3.1) and the reasons behind the design decisions. The entire idea of this first design of value types is to treat value types as immutable objects with no identity. This not only allows the reusability of the functionality of objects but also opens the possibility to remove a heap allocation and field loads completely. This allocation and field load removal approach represents the main goal of this thesis. While this chapter presents only the high-level design choices, the proceeding Chapter 5 discusses the concrete implementation details with references to the actual code.

One of the general design challenges was to find and reuse existing components and features without introducing fundamental and extensive changes. Section 4.1 describes the design adaptations and extensions made to the two intermediate representations HIR and LIR (see Section 2.5.3.1) for value types in the C1 compiler. A straightforward unoptimized approach (Section 4.2) is used to support the new bytecodes by treating the value types as normal objects without any optimizations. This is especially useful in comparison to the evaluation of the benefits of the optimized value type implementation as presented in Chapter 6. Afterwards, a description of the allocation and field load removal approach with a buffer concept follows in Section 4.3, which is the core part of the design. Section 4.4 presents an approach to buffer results of field loads from value type instances on the heap. Finally, Section 4.5 briefly explains the considerations for OSR-compilation in association with value types. The exploration of deoptimization support for value types in the C1 compiler is explained later in Section 5.8 as it needs direct reference to code details which are part of Chapter 5.

4.1 Changes to the Intermediate Representations

The idea is to reuse as many parts of the two intermediate representations, HIR and LIR, as possible. A value type is in some sense very similar to an actual object. Instances of value types are created, stored, and loaded from local variables. The fields of a value type instance can be read and written to (at the creation of a new value type). The store and load mechanism for a local variable is independent of its actual type. Thus, value type local variables can simply be mapped to the already existing concept (and implementation) of loading and storing local variables without the need to change anything. A required field store (associated with the creation of a new value type instance with `vwithfield`) or field load to and from an allocated value type instance on the heap are also similar to the ones to and from an object instance on the heap. The HIR and LIR

are not affected for those value type features. However, creating a new value type instance has to be distinguished from an object instance. Therefore, a new node `NewValueTypeInstance` is added to the HIR representing such an instance. To later remove the allocation completely (see Section 4.3), the node contains a flag that indicates if the value type is allocated or not. In the scope of this thesis, the `NewValueTypeInstance` node is translated to a normal object instance in the LIR if the allocation is not removed by an optimization. The interpreter and the C2 compiler offer the functionality to pass and return the fields of a value type in registers and thus avoiding the allocation. This is left as future work for the C1 compiler as it would have exceeded the scope of this thesis (see Section 7.1).

Besides adding a new node for value type instances, the existing *phi node* (see Section 2.6) implementation in the HIR and LIR has to be adapted and extended for value type instances to enable their removal at some point. As for the `NewValueTypeInstance` node, the phi node is extended with a flag to indicate if the instance needs to be allocated. Phi nodes are additionally used for all the fields of a value type instance (i.e. field-phi nodes). More concrete details can be found in Section 4.3 about the approach to remove allocations and field loads.

4.2 Unoptimized Value Types

In this section, the first simple unoptimized implementation approach for value types in the C1 compiler is described. At the start of the thesis the C1 compiler supported no value type related features at all. However, the C1 and C2 compiler share a common *compiler interface* which already defines some fundamental and reusable value type information for the C2 compiler implementation, such as a description of a value type instance class or a new type class for a value type. Thus, the first goal was to get familiar with the HotSpot™ JVM environment and to provide in a simple way a basic support for the value type related bytecodes `vdefault, vwithfield, vload, vstore, vreturn` and `getfield`.

In this simplified unoptimized approach, all bytecodes, except `vwithfield`, are handled very similar to the ones used for objects. A value type is always allocated on the heap like an object. This means that `vdefault` can simply reuse the existing implementation for the `new` bytecode as they have the same behavior. The `vreturn` bytecode always returns an allocated value type (i.e. an object) and therefore can reuse the implementation for the `areturn` bytecode for returning a normal object. As described in the last section, a store and a load of a local value type variable is similar to any other variable. Thus, the `vload` and `vstore` are simply mapped to the existing common implementations for loading and storing such a variable (e.g. `istore, iload, astore, aload etc.`). A field load of an allocated value type also works in the same way as for an object. Only the `vwithfield` bytecode has to be treated differently. It does not only require a field store but also an allocation of a completely new value type by copying the field values of the original instance. This is achieved by iterating over all field values (except the one to update) of the old value type instance and storing them to the new value type instance. In the last step, the attached value

in the `vwithfield` bytecode is stored to the specified field. The required phi nodes for value type instances at merge points are created and handled similarly to the ones for normal object instances.

There are no optimizations applied and thus the design is very similar to treating a value type completely as an object. There is no real benefit visible by this simplified approach except for not needing to invoke a constructor at creation. However, this unoptimized version was useful for getting familiar with the C1 compiler environment and to set a basis for further optimizations like the allocation and field load removal (see Section 4.3). Running a program with this unoptimized implementation can be done by explicitly adding the new JVM flag `-XX:-OptimizeValueTypes` (see Appendix A.3).

4.3 Allocation and Field Load Removal

An allocation of an object is expensive (see also Chapter 6) compared to the use of values in a register or from the stack. The HotSpot™ JVM provides a special *thread local allocation buffer* (TLAB) which reserves a small area in the heap just for a thread itself to allow faster allocations (also called fast-path). However, this is still slow compared to the use of registers or the stack. Moreover, the space of those TLABs are limited. If it is not possible to use them, the JVM needs to perform an even more expensive runtime call to allocate a new instance (also called slow-path). This, for example, requires the storing and restoring of all registers before and after the allocation. The additionally required heap accesses to load field values are another limiting factor for the performance together with the increased time spent for garbage collection due to managing more objects on the heap. Therefore, it would be beneficial to exploit the properties of value types to avoid those heap allocations and heap accesses for field loads whenever possible. This is set as the main task of this thesis. The design of this core optimization of removing value type allocations completely (together with field loads) is presented in the present section.

Throughout the following sub sections, the value type class shown in Listing 4.1 is used as a basis for all examples. Each method shown in a listing is implicitly part of the `VTPoint` class without repeating the entire class structure again. Hence, it simplifies the examples and improves their visibility. The C1 compiler inlines the `create()` method due to its small size. The `uninlinedCreate()` method is identical to the `create()` method and would also be inlined. But for the examples in the chapter the special compile command `-XX:CompileCommand=dontinline,VTPoint::uninlinedCreate` is used to explicitly force the compiler not to inline the method.

4.3.1 Buffering Field Values

The key idea behind the removal of allocations is based on the insight that all fields of a value type are `final` and thus immutable. They cannot be modified anymore. An assignment of a field is definite and done only once with the `vwithfield` bytecode which furthermore creates a completely new instance. Each time a new value type is created with the `vdefault` bytecode, its

Listing 4.1: The `VTPoint` value type class which is used as a basis for the following code listings

```
1  __ByValue final class VTPoint {
2    final int x;
3    final int y;
4
5    VTPoint() { this.x = 0; this.y = 0; }
6
7    // Method inlined
8    __ValueFactory static VTPoint create() {
9      return __MakeDefault VTPoint();
10   }
11
12   // Method NOT inlined
13   __ValueFactory static VTPoint uninlinedCreate() {
14     return __MakeDefault VTPoint();
15   }
16
17   __ValueFactory static VTPoint set(VTPoint vt, int x, int y) {
18     vt.x = x;
19     vt.y = y;
20     return vt;
21   }
22 }
```

fields get their default values assigned (see Section 2.3.2). New instances can be created from it with different field values with `vwithfield`. Performing those operations inside the same method offers a decisive advantage. The actual values of the fields are statically known inside the method scope and do not change anymore. Hence, it seems to be a good idea to keep track of those values. This is done in a *method-global*[1] data structure at compile time during the parsing stage. An inlined method also uses the same buffer of the unlined root caller method since the inlinees are directly parsed into the method scope of the root method. Every time a field is accessed with the `getfield` bytecode or needs to be written with `vwithfield`, the buffer can provide the field values immediately. This makes a heap allocation of completely newly created value type instances in a method redundant if the value type does not explicitly need to be allocated (see Section 4.3.2). Additionally, the associated field loads and stores can be removed. Even if an allocation is needed, the field loads from the heap can still be avoided by directly providing a buffered value.

An example to illustrate this design idea is shown in Figure 4.1. The field values of the newly created value type *vt* with `__MakeDefault VTPoint()`[2] (i.e. uses `vdefault`) get their default values assigned (in this case to 0 for `int`). Those are stored in a buffer. Once another new value type is created, as on line 3 with `vwithfield`, a new entry can be added to the buffer with the

[1] This means that the buffer is available during the entire compilation of a single method.

[2] Syntax to create a new value type instance with `vdefault` for the experimental version of `javac` included in the project Valhalla (see Section 2.3.2 and Listing A.1).

new values. Whenever a field is accessed, as on line 2 and 4, the buffer can provide the required values immediately. Each *newly* created value type instance can be used as a key to look up the actual field values associated with the instance. These field values are managed as an array per instance inside the buffer. The field offset can be used as a unique index to read the required field value (see Section 5.4.1). As a consequence, the heap allocation of the value type creation on line 1 can be completely removed.

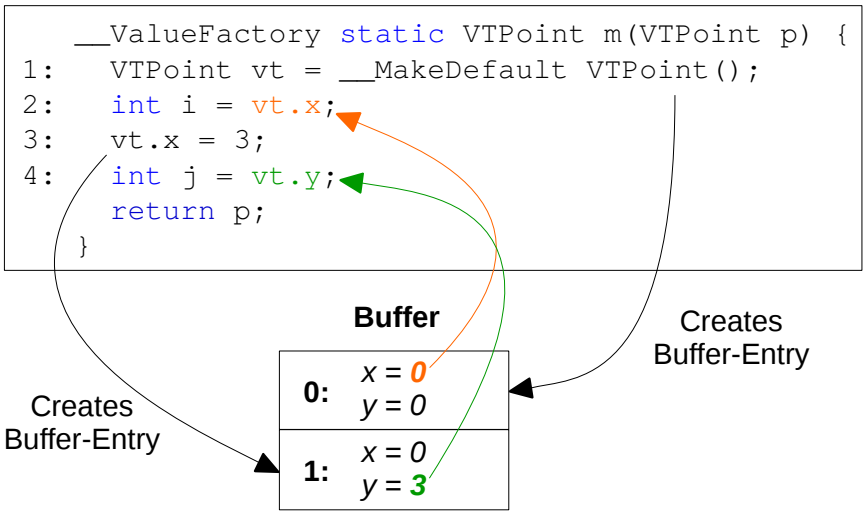


Figure 4.1: Example of introducing a buffer to make value type allocations and the associated field loads redundant

The buffer can store all primitive typed fields. It can additionally hold object references which, however, still require a possible pointer traversal to get to the object stored on the heap. Since this buffer concept replaces actual load and store nodes in the HIR (i.e. `LoadField` and `StoreField` instances as described in Chapter 5), the buffer stores HIR nodes directly in the field value arrays. For example, the primitive types are represented by `Constant` nodes.

There are certain situations where an allocation is performed for a value type instance but the buffer can still be used to avoid a field access from the heap as shown in Listing 4.2. The value type *vt* created on line 2 is assumed to be allocated afterwards before it reaches line 9 for some reasons explained in Section 4.3.2. Moreover, *vt* is assumed not to be reassigned and still represents the same value type instance throughout the method. Therefore, the field load from the heap of the allocated value type *vt* on line 9 can be omitted since the buffer has the (immutable) field values of *vt* buffered. The call $vt.x$ is simply replaced by 0.

4.3.1.1 Lazy Buffering

When a new instance is created with `vdefault`, the fields get their default values assigned. This property can be exploited to use a *lazy* buffer design. A new instance is recorded but only creates an empty field value array in the buffer. Accessing a field later whose index is not in range of the

Listing 4.2: Example of avoiding field loads from an allocated instance.

```
1 __ValueFactory static VTPoint m() {
2   VTPoint vt = __MakeDefault VTPoint();
3
4   /*
5    * Some code which triggers allocation of 'vt' (see Section 4.3.2)
6    * NO reassignment of 'vt'
7    */
8
9   int i = vt.x; // Replaced by 0 from the buffer, NO field load
10  // ...
11 }
```

Listing 4.3: Accessing an unknown field of an allocated value type requires more information

```
1 // Method NOT inlined
2 __ValueFactory static VTPoint m(VTPoint vtArg) {
3   VTPoint vt = VTPoint.unlininedCreate(); // NOT inlined
4   int i = vt.x; // What is 'vt.x'? Information not available
5   int j = vtArg.y; // What is 'vtArg.y'? Information not available
6   // ...
7 }
```

array, implies that a default value can be used. Similarly, if the index of an accessed field is in range, the buffer uses `NULL` to indicate that a default value can be used instead. This design is more compilation-memory efficient since the field value arrays are only as big as the largest field offset that contains a non-default value. For example, a new value type instance that is created with `vdefault` has all its fields set to their default values and therefore just creates a new entry in the buffer with an empty field value array.

4.3.1.2 Instances with Statically Unknown Field Values

Buffering fields becomes impossible if a value type is used in a method without having any further statical information about its field values. For example, a value type instance could have been passed to the current (uninlined) method as a parameter or is returned from a (uninlined) method call. The problem with those instances is that they were not created in the current scope of the method. Hence, their field values are not statically known. An example of this situation is shown in Listing 4.3. Both value type instances *vtArg* and *vt* were created (and allocated) outside of the scope of `m()`. Thus, they do not provide any static information about its field to be used on line 4 and 5. Even though a new value type is created inside a method, its field value information can remain incomplete as shown in Listing 4.4. While the x field value is statically known, the y field value of *vtArg* still remains unknown. Therefore, the buffer concept needs to be adapted for those situations as shown in the following paragraph.

Listing 4.4: A more profound example with the same information problem as in Listing 4.3

```
1 // Method NOT inlined
2 __ValueFactory static VTPoint m(VTPoint vtArg) {
3   vtArg.x = 3;
4   int i = vtArg.x; // Buffered value 3
5   int j = vtArg.y; // What is 'vtArg.y'? Information not available
6   // ...
7 }
```

The simple *policy* introduced for the buffer states that each *newly* created value type instance inserts an entry into the buffer. The already existing and allocated value types with unknown field values do not get and conceptually do not need an entry in the buffer since there is no information available anyways. Each time a value type is not found in the buffer automatically means that a field load has to be performed due to missing information. However, there are situations where a new value type is created from such an allocated value type with unknown field values as shown above in Listing 4.4 on line 3. A new buffer-entry gets created with the field value array that contains the value 3 for the field *vtArg.x*. The problem faced is what value to store for the *vtArg.y* field. The `NULL` constant cannot be used since it already has the meaning of a default value as described in Section 4.3.1.1 above. Thus, a special, designated, and unique constant *LOAD_REQUIRED* is introduced which is treated as a `FieldLoad` instance. It is used for all field values of a value type instance inside a field array buffer-entry that require a load due to missing static field information. Whenever this constant is read from the buffer, the HIR will simply insert a `LoadField` node to perform the field load from the value type on the heap. This is similar to a normal field load from an object. Storing and reusing direct `LoadField` nodes in the buffer seems to be a straightforward optimization but is more complicated as discussed in the Section 4.4. In this first approach, the fields are loaded each time the *LOAD_REQUIRED* constant is read or the value type instance has no entry in the buffer. The implementation of the buffer has to ensure that a value type instance entry maintains a pointer to the last instance on which a `vwithfield` was applied to have a reference to the instance from which a possible field needs be loaded from the heap (see Section 5.4.4) since the *LOAD_REQUIRED* constant does not provide this information. An example of a buffer-entry is shown in Figure 4.2. It presents the buffer-entry (i.e. the field value array) for *vtArg* in Listing 4.4 after line 3. The entire buffer only contains this entry at that point in the method.

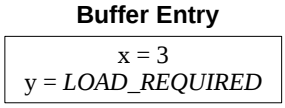


Figure 4.2: Buffer-entry (i.e. the field value array) of *vtArg* in Listing 4.4 after line 3

Reading the *LOAD_REQUIRED* constant or from an instance that is not present in the buffer requires a field load as stated above. An example of this is illustrated in Listing 4.5. The corresponding buffer states and the created pseudo-HIR nodes are shown in Figure 4.3. The allocated value type instance *vtArg* is not found in the buffer on line 3 (which indicates that its

Listing 4.5: Example of reading unknown fields from value type instances

```
1 // Method NOT inlined
2 __ValueFactory static VTPoint m(VTPoint vtArg) {
3   int i = vtArg.x; // Not found in buffer, needs load
4   vtArg.x = 3; // New entry in buffer
5   int j = vtArg.x; // Replaced by 3, stored in buffer
6   int k = vtArg.y; // Found in buffer: LOAD_REQUIRED , needs load
7 }
```

field values are statically unknown). Therefore, a field load is appended to the HIR for *vtArg.x*. After creating a new value type instance on line 4, which adds a new buffer-entry at index 0, the access of the x field on line 5 is replaced by the stored value from the buffer (i.e. the value for x from the field value array of *vtArg*). The HIR appends a `Constant` node accordingly. The access of *vtArg.y* on line 6 reads the *LOAD_REQUIRED* constant from the buffer. Hence, a `LoadField` is appended to the HIR to perform the field load.

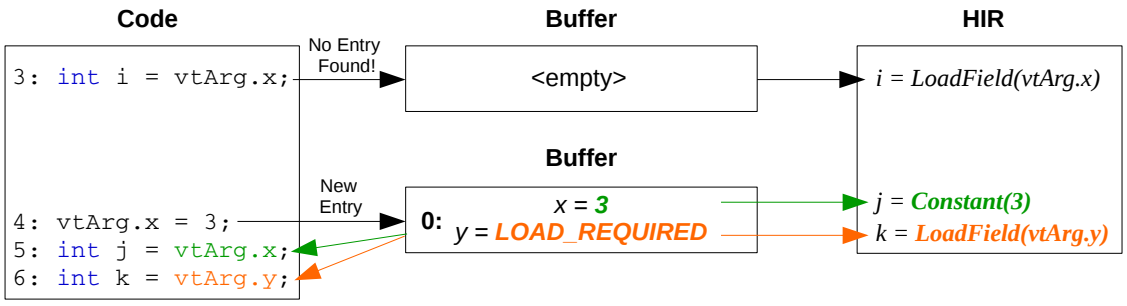


Figure 4.3: The buffer states and the created pseudo-HIR of Listing 4.5

Algorithm: The final algorithm for creating a new value type instance and how to set up the buffer accordingly is summarized and shown as pseudo-code in Algorithm 1 in Appendix A.5. The algorithm for accessing a field value with the help of the buffer and the created nodes for the HIR is shown as pseudo-code in Algorithm 2 in Appendix A.5. The latter algorithm is improved later with the introduction of phi nodes in Section 4.3.3.

4.3.2 Requirements for an Allocation

There are certain situations where a value type instance allocation is inevitable. This is the case if it escapes the scope of the current uninlined method. This happens either:

(i) if a value type is used as an argument for an uninlined method call or

(ii) if a value type is returned at the end of the method

(iii) if a field is itself a value type

Listing 4.6 illustrates (i) in method `m()` and (ii) in method `m2()`. A `VTPoint` instance is created (but not allocated) on line 3 in method `m()` which then escapes over an unlined method call on

Listing 4.6: The two requirements (i) and (ii) for an allocation shown in `m()` and `m2()`, respectively

```
1 // Method NOT inlined
2 void m() {
3   VTPoint vt = VTPoint.create(); // Assume inlined and NOT allocated
4   uninlined(vt); // Escapes m() -> 'vt' needs to be allocated
5 }
6
7 // Method NOT inlined
8 void uninlined(VTPoint vt) {
9   /* Some code */
10 }
11
12 VTPoint m2() {
13   VTPoint vt = VTPoint.create(); // Assume inlined and NOT allocated
14   return vt; // Escapes m2() -> 'vt' needs to be allocated
15 }
```

Listing 4.7: Pseudo-code of an inlined method without an allocation inside `VTPoint.create()`

```
1 void m() {
2   // Inlined call 'VTPoint vt = VTPoint.create();' -> NOT allocated
3   VTPoint vt = __MakeDefault VTPoint();
4
5   // ...
6 }
```

line 4. Therefore, the value type needs to be allocated before the actual method call. Similarly, the created (but not allocated) value type instance on line 13 needs to be allocated before escaping over the `return` statement on line 14. Calling an inlined method such as the `create()` method from Listing 4.1 embeds the entire method body directly inside the method body of the caller as shown in pseudo-code Listing 4.7. In this case, the value type instance returned from `create()` does not escape and does not need to be allocated. Similarly, calling an inlined method with a value type argument does not need an allocation of the value type instance.

In case (iii) the value type must be stored in a field. It is either allocated and stored as a reference or stored in a flattened representation (i.e. the fields of the value type are inlined into the holder class). However, this is not supported in this thesis and is left for future work (see Section 7.1). Thus, an allocation is only inserted for case (i) or (ii).

Each value type instance contains a flag that indicates if a value type is allocated or not. This flag is part of the HIR node `NewValueTypeInstance`. Whenever a value type instance is allocated, the flag is set to `true`[3]. The decision if a value type instance is allocated or not gets

[3]When a new value type instance is created from an allocated instance with `vwithfield`, then a new `NewValueTypeInstance` node is created which has its flag set to `false` again as it is not allocated.

more complex if the control flow of a method is split and merged again. The presented approach needs to be improved and is discussed in the next section.

4.3.3 Branches with Phi Nodes

Assigning different value type instances in different branches to a local variable requires not only a merge of the value type instance but also a mechanism to merge all stored field values from one entry in the buffer with those entries from the other branches. The correct value type instance to use is decided with the help of a phi function for the value type instance variable at runtime (see Section 2.6)[4]. The *i-th* operand represents the value type instance that was assigned to the variable at the end of the *i-th* predecessor block. This idea is reused to also create a phi function for each individual field of a value type in the HIR (referred to as *field-phi* to distinguish from a normal value type instance phi). The *i-th* operand of such a field-phi function of a value type instance phi $p_{instance}$ corresponds to the field value in the buffer-entry for the value type of the *i-th* operand of $p_{instance}$ (i.e. in the value type instance at the end of the *i-th* predecessor block).

An example to illustrate this concept of field-phis is shown in Figure 4.4. A new value type instance is created in block *B2* (for simplicity the term *block* is used throughout all the examples when actually meaning the SSA term *basic block*) by assigning the value 3 to *vt.x* while another value type instance is created in block *B3* with the value 4 for *vt.x*, respectively. To assign the correct value to *i* in block *B4* and remove the value type allocations for *vt* completely, a phi function has to be introduced for each field of the value type variable *vt*. The resulting field-phi function for *vt.x* takes the stored values in the buffer entries from the instances of the predecessor blocks as operands and is finally assigned to *i* in block *B4*. The field-phi node for *vt.y* is removed since the values of both branches are the same.

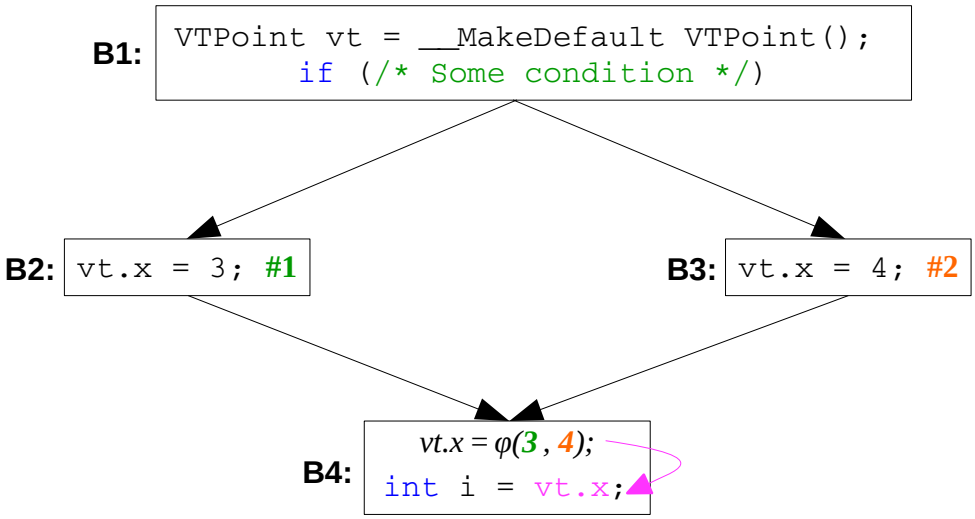


Figure 4.4: Example of merging different branches with different value type instances for the same variable with statically known field values

[4]Each local variable in the C1 compiler automatically gets a phi function at a merge point.

If a field has a statically unknown value from an allocated value type, the buffer provides the *LOAD_REQUIRED* constant. This indicates that a field load is required. A field-phi uses this constant as an operand value for all fields that need a load. An example is shown in Figure 4.5. The field value *vt.y* is unknown in block *B2* while the field value *vt.x* is unknown in block *B3*. Therefore, the resulting field-phi nodes in block *B4* use the *LOAD_REQUIRED* constant as operand value for those unknown field values.

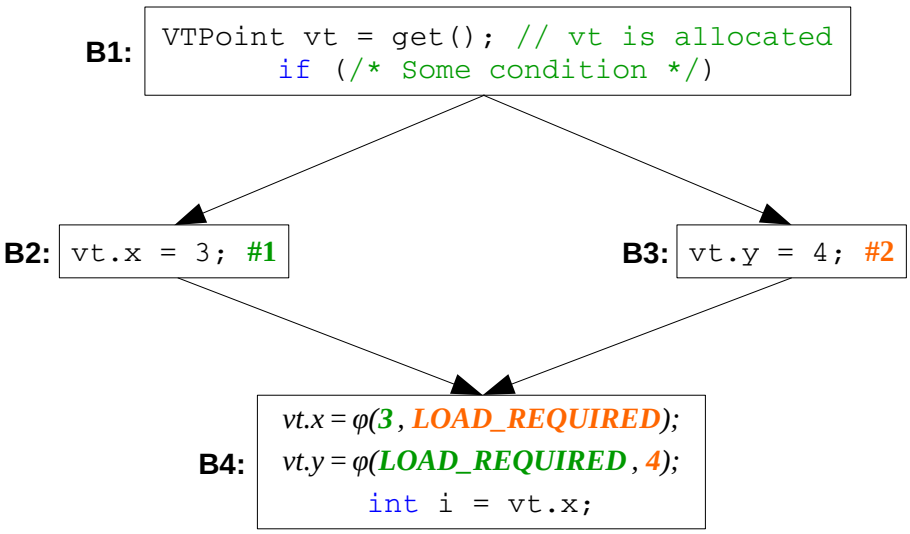


Figure 4.5: Example of merging different branches with different value type instances for the same variable including statically unknown field values

The created field-phi nodes are not stored in a local variable as they do not refer to an actual variable in the code. They only exist as part of a value type instance in this allocation and field load removal optimization. Therefore, the field-phi nodes are directly stored as field values in the buffer. This leaves the question which key should be used to fetch those field-phi values. The answer is to use the additionally created phi functions for the value type instance variables[5]. Those add a new buffer-entry since they represent a new instance in the HIR. The phi nodes can then be used to store and look up field values represented by own field-phi functions.

The example shown in Figure 4.6 illustrates this concept together with the state of the buffer. Both assignments in block *B2* and *B3*, respectively, create a new value type instance and add a new entry (i.e. a field value array) to the buffer with the corresponding value for *vt.x*. Block *B4* then sets up a phi function for *vt* and also stores it as a new entry in the buffer. It fills its field value array with the additionally created field-phi function for the field *vt.x* accordingly. This field-phi is afterwards fetched from the buffer and assigned as value to *i* in block *B4*. This allows the successful removal of all possible allocations of *vt* in block *B1*, *B2*, and *B3*. Both operands for the field-phi for *vt.y* would contain the value 0 and thus is not shown. Moreover, the *vt.y* values in the other two entries are removed due to the lazy extension of the

[5]All other normal variables in the code also get automatically a phi function in the C1 compiler.

buffer shown in Section 4.3.1. In the following, the term *phi* or *phi node* generally refers to phi functions for value type instances (if not otherwise clear from the context) while the term *field-phi* or *field-phi node* is used for phi function of the fields of a value type instance to avoid confusion.

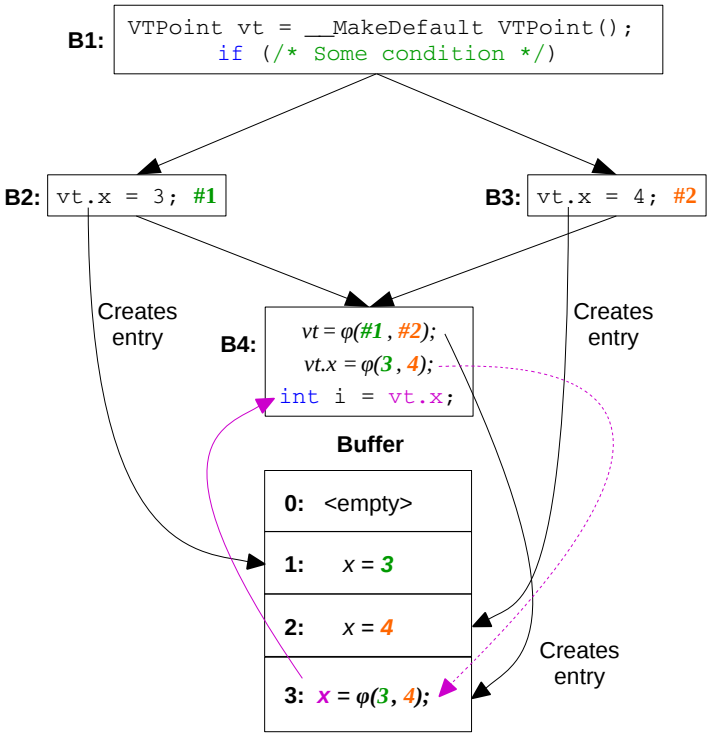


Figure 4.6: Extended example of merging different branches with different value type instances for the same variable including the buffer

The same approach is also applied for loops as shown in Figure 4.7. Analogously to the previous example, a phi node is created for *vt* and stored in a new buffer-entry containing the field-phi for *vt.x* in block *B2*. This field-phi is then read from the buffer and used to assign the variable *i* in block *B4*. However, the implementation itself is more difficult for loops. The field values from a loop back-branch (i.e. from block *B3* in Figure 4.7) are not yet known while parsing the condition block in the C1 compiler. Thus, loops need some special care in the design (see Section 4.3.5 and 4.3.6) and later in the implementation (see Chapter 5).

4.3.4 Value Type Phi Allocation

As described in the last section, a value type instance can be replaced by a phi node if the control flow is split and later joined again. Deciding if such a phi node represents an instance that is actually allocated (or needs to be allocated) requires a more profound analysis. Some operands could have an allocated version, while others do not. The phi node itself is not a real instruction in the source code. It only merges values from different branches. Therefore, a value type instance phi node needs to decide its allocation status on the basis of its operands. It can only be treated as completely allocated if all operands (i.e. the value types of each branch) are allocated since it

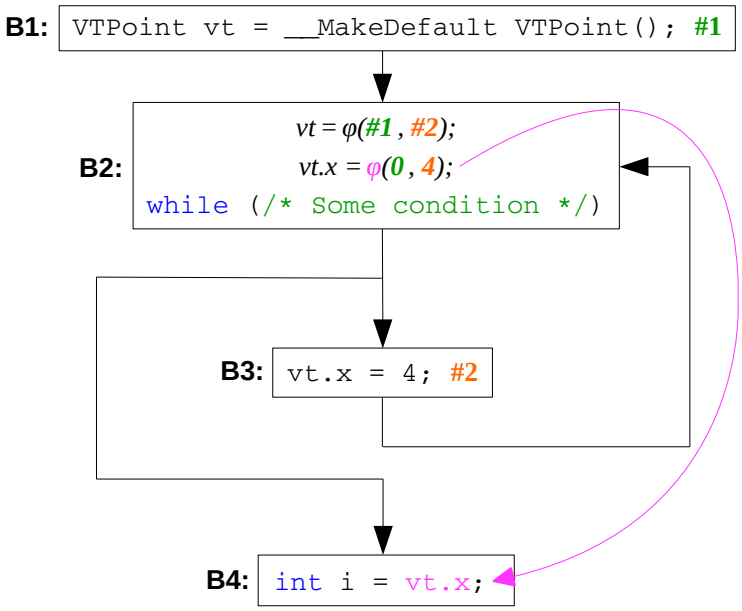


Figure 4.7: Example of merging different branches of a loop with different value type instances for the same variable

is not known from which branch the merge point is reached at runtime. If none of the operands is allocated, then the entire phi node can obviously be seen as not allocated. However, there are situations where one branch could provide an allocated value type instance while others provide an unallocated version. The phi node represents a mixture of a "partly-allocated" instance. An example of this is shown in Listing 4.8. The phi node created for *vt* at the merge point just before line 9 contains an allocated operand from line 4 and an unallocated operand from line 2 from the `else`-branch which does not allocate *vt*. Depending on the condition on line 3, an allocation for line 9 is only needed if it evaluates to `false`. In that case, the phi node would choose the unallocated version of *vt* from the `else`-branch at runtime[6].

There are two possible ways to solve the problem of setting the allocation status of a new phi node at its insertion point: (i) allocate all operands immediately (eagerly) or (ii) allocate the operands lazily upon an escape of the phi node. While the first approach is straightforward and easier, it is not efficient. For example, the code in Listing 4.8 could be modified such that the escape of *vt* on line 9 is removed (see Listing A.2 in Appendix A.5 which shows the changed code for this example). Furthermore, assume that the condition on line 3 is never evaluated to `true`. If this modified method is executed many times, then the eager approach would allocate *vt* on the `else`-branch in each invocation even though *vt* does not escape the method scope and hence would not require an allocation. Therefore, the lazy approach (ii) is chosen for more efficiency.

[6]This also requires that a `NewValueTypeInstance` node is copied later in the implementation upon an allocation to differentiate between allocated and unallocated versions in different branches (see Chapter 5).

Listing 4.8: Merging branches of an allocated and unallocated instance with a required allocation

```
__ValueFactory static VTPoint m() {
  VTPoint vt = __MakeDefault VTPoint();
  if (/* some condition */) {
    uninlined(vt); // Escapes m() -> 'vt' needs to be allocated
  } else {
    /* Some code, NO allocation of 'vt' */
  }
  // Create phi node for vt, is it marked allocated?
  uninlined(vt); // Allocation?

  // ...
}

// Method NOT inlined
void uninlined(VTPoint vt) {
  /* Some code */
}
```

A possible design for the lazy approach is to insert an allocation for each operand that is not allocated if the phi node requires to be allocated. The allocation is pushed to the end of the block of the corresponding operand. Such an example is shown in Figure 4.8. The value type instance *vt* remains unallocated until it reaches block *B4* where it escapes over the `return` statement. At this point, an allocation is inevitable. The variable *vt* is represented by a phi node in block *B4* due to the merge of both blocks *B2* and *B3*. Since no operand is allocated, an allocation needs to be inserted at the end of both predecessor blocks. On the other hand, if *vt* would have already been allocated in one of the blocks, the corresponding allocation push can be removed. If all operands are already allocated, then an allocation push can even be completely omitted.

Improved Allocation Rules: Even though this approach is correct, it is not very efficient in terms of code size and for the explicit desire to allocate as late as possible to keep registers free. If all operands are unallocated, it would be sufficient to allocate the instance directly in the block which lets the phi node escape since all field values are already present as field-phi nodes in the buffer. Therefore, the approach of the lazy allocation of phi nodes is slightly modified and improved:

- If *all* operands are allocated, then no allocation is needed.
- If at least *one* operand (but not all) is allocated, then an allocation is pushed up for all unallocated operands.
- If *no* operand is allocated, then an allocation is directly inserted in the block where it is needed and replaces the phi function as local variable inside the block.

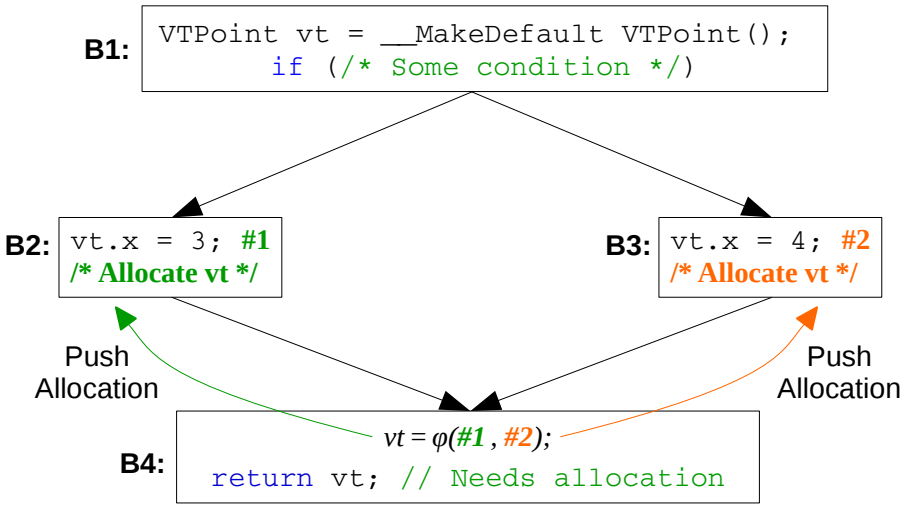


Figure 4.8: Example of pushing up an allocation for all operands of a phi function to the corresponding end of a block

4.3.5 Recursive Value Type Phi Allocation

It is not evidently clear how to apply the improved allocation rules presented at the end of Section 4.3.4 if a phi node for a value type instance contains another phi node as an operand or even itself (in case of loops). Therefore, the presented approach needs to be extended to work also recursively. For that purpose, each value type instance phi node gets an own *allocation-flag* describing the state of its operands:

af1: A phi node is marked as being *allocated* if *all* of its operands are allocated or marked as *allocated*.

af2: A phi node is marked as being *partly-allocated* if *one* (but not all) of its operands is allocated or marked as *allocated* or at least one operand is already marked as being *partly-allocated*.

af3: A phi node is marked as being *not-allocated* if *no* operand is allocated and none is marked as being *partly-allocated* or *allocated*.

af4: A phi node in a loop-header block is marked as being *partly-allocated* on parsing the loop-header and once the last operand is available, it is *completely* reevaluated and marked according to the other rules above. If the last operand refers to itself recursively, then it is ignored in the reevaluation.

If a new phi node is inserted with all its operands, the allocation-flag needs to be set accordingly. This is achieved by an iteration over all operands. An operand is allocated if it represents an allocated value type or if the operand itself is a phi node that is marked as *allocated*. If one of the operands is a phi node that is itself *partly-allocated*, then the new phi node is also marked as *partly-allocated* since there is at least one allocation needed. A non-phi value type instance is always either allocated or not allocated. The additional *partly-allocated* status is only needed for phi functions.

A value type instance phi node is automatically marked as *partly-allocated* when parsing a loop-header block since it is not clear whether the last missing operand is allocated, partly-allocated, or not allocated at all. Thus, the rule is conservatively and sets it to *partly-allocated* to ensure correctness during the parsing stage of the loop-body together with the rules introduced in the following paragraph. Once all operands are known at the end of the loop-body, the phi node is completely reevaluated and set to one of the three states. If the last operand refers the field-phi itself recursively, then this operand is ignored for deciding the new status of the field-phi (due to a cycle). This information is now complete and can be used in successor blocks after the loop.

Recursive Allocation Rules: Once a value type instance needs to be allocated and is represented by a phi node, the improved allocation rules shown in Section 4.3.4 are adapted and extended to apply them recursively with the introduced allocation-flags and the thoughts presented in the previous paragraphs. Upon a command to allocate a value type instance, the following recursive rules are applied:

A1: A phi node marked as being *allocated* does not need to insert an allocation.

A2: A phi node marked as being *partly-allocated* pushes an allocation to each non-phi operand which is not already allocated and an allocation command to each phi operand which is marked as being *partly-allocated* or *not-allocated* recursively.

A3: A phi node marked as being *not-allocated* directly inserts an allocation in the corresponding block and replaces the phi function as local variable inside the block.

A4: During the allocation process, the *partly-known* phi nodes are marked as being *allocated* and newly inserted allocations replace the old operands of the phi function[7].

A5: If the phi node in a loop-header block is marked as *allocated* but the last operand turns out to be not at the reevaluation, then an allocation is pushed towards the block of the last operand (i.e. to the loop-body).

A phi node which is marked as *not-allocated* (all operands unallocated) is directly replaced by an allocated `NewValueTypeInstance` in the current block. Since a phi node is also stored as local variable in the block, it needs to be overridden by the new instance to use it correctly (instead of the phi node) in possible successor blocks.

Special care has to be taken for loops. They can contain phi nodes which refer to itself recursively as shown in Figure 4.9. Assume that block *B3* is not reassigning *vt*. Therefore, the phi function created in block *B2* refers to itself over the last operand. During the parsing stage, the phi function is incomplete at the first visit of the loop-header block. The loop-body is not yet known. Thus, the phi function is never directly marked as *allocated* or *not-allocated* on the first visit. It is not clear if the last missing operand needs an allocation or not. Hence, the phi node is marked as

[7] This last point is automatically done by the already existing implementation of phi functions in the C1 compiler and does not need an additional effort.

partly-allocated (rule af4). If such a phi node needs to be allocated inside the loop-body, it pushes an allocation to all currently known predecessor blocks before the loop-header (due to rule A2 for *partly-allocated* phi nodes). Once the entire loop is parsed, the remaining operand is filled in and the final decision is made if the phi node is marked as *not-allocated*, *partly-allocated* or *allocated* as described above. This approach also has a performance advantage of pushing an allocation out of the loop-body instead of allocating it directly upon a usage in a method call to escape the method scope from the loop-body.

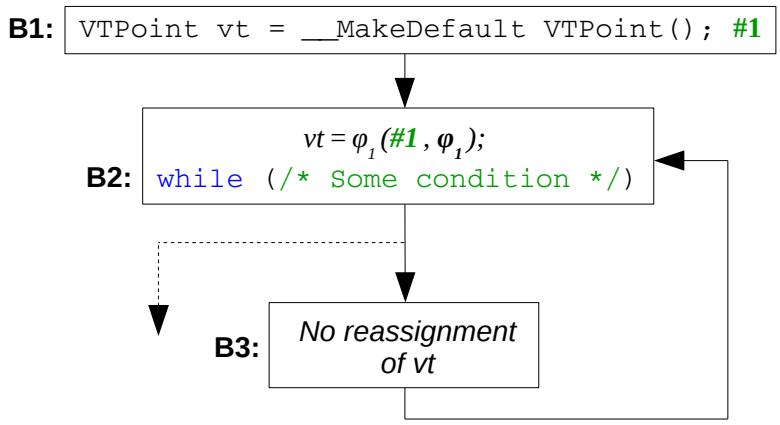


Figure 4.9: Example of a recursive phi function for value type instance *vt* in the context of a loop

A loop-header phi node can escape the method scope from inside a loop-body which triggers an allocation process (rule A2) and marks the phi node as *allocated* (rule A4). However, a situation can occur at the reevaluation step of the allocation-flag of a loop-header phi node (after the loop-body was parsed) which has its flag already set to *allocated* but the last operand turns out to be not marked as *allocated*. This means that the allocated instance that escaped was replaced in the meantime by another value type instance in the loop-body that is not properly allocated (i.e. not allocated or not marked as *allocated* in case of a phi node). This instance needs an additional allocation because it might be reused again in the next iteration which assumes an instance that is marked as *allocted*. Otherwise, the compiler uses the wrong information of a completely allocated instance which could result in a runtime error[8]. An allocation is therefore pushed to the block represented by the just newly setup last phi operand of the loop-header phi node (rule A5). Only now this information is complete and can be later used for successor blocks after the loop correctly.

An example of this situation is shown in Figure 4.10. The created value type instance *vt* is first unallocated in block *B1*. While parsing the block *B2*, the phi function is set to *partly-allocated* according to the allocation-flag rules from above. The last operand is not yet known at this point. When parsing the block *B3* the value type instance *vt*, represented by a phi function, escapes through an uninlined method call. Thus, an allocation is pushed to block *B1* according to the rule for partly-allocated phi nodes (rule A2) and the node itself is marked as *allocated* (rule A4). Just after this call, *vt* gets a new value type instance assigned. Thus, it is set to *not-allocated*. Once

[8] An unallocated instance could be accessed, which does not exist on the heap. This could result in a potential `null` pointer exception or a similar error.

the loop-body is completely parsed (i.e. block *B3*), the phi function in block *B2* is revisited. It is already marked as *allocated* but the last operand is not. Therefore, an allocation is pushed to the end of the loop-body in block *B3*. Otherwise, the next iteration uses the wrong assumption that the value type instance is already (completely) allocated.

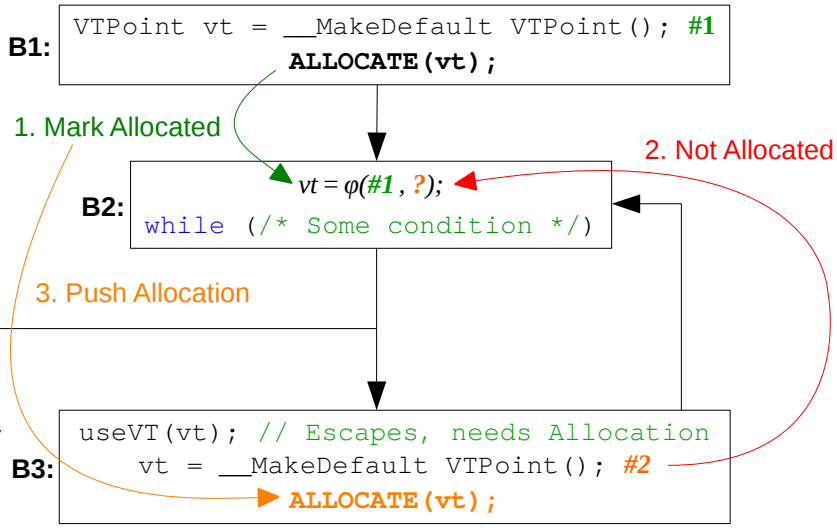


Figure 4.10: Example of the need to push an allocation to the block of the last loop-header phi operand if it escapes inside the loop-body and gets reassigned by an unallocated value type instance

An extended example of pushing up allocations and replacing phi functions with the recursive allocation rules from above is shown in Figure 4.11. There is a value type instance *vt* of type `VTPoint` involved that is assumed to be created at the start of the method (i.e. accessible in each block) and is not modified outside of the shown blocks. Block *B9* returns *vt*, which therefore needs to be allocated properly (see Section 4.3.2). The value type *vt* was never allocated in a block before except in block *B2* which contains an unlined call with *vt* as an argument and consequently is already allocated in this block. Therefore, the created phi function φ_1 merges an unallocated instance from block *B1* with an allocated instance from block *B2*. According to the allocation-flag rules it is marked as *partly-allocated* (rule af2). The phi functions φ_2 and φ_3 merge two unallocated instances and thus are marked as *not-allocated* (rule af3). Finally, the created phi function φ_4 in block *B9* merges a partly-allocated instance with a not-allocated instance and hence is marked as *partly-allocated* (rule af2). According to the partly-allocated rule (rule A2), the phi function φ_4 requires to push up allocation commands recursively to block *B3* and *B8*. Block *B3* itself is partly-allocated and thus pushes an allocation up to *B1*. There is nothing to do in block *B2* since the instance is already allocated. The phi node φ_3 in block *B8* is not-allocated. According to the not-allocated rule (rule A1) the allocation can directly be inserted in this block. The local variable that holds φ_3 is replaced by the new instance. There is nothing else to do for any predecessor blocks. The φ_3-operand of φ_4 in block *B9* is automatically replaced by the newly allocated instance. During this process, each partly-allocated phi function that was involved contains now allocated operands. Therefore, they are marked as *allocated* due to rule A4[9].

[9]If the method would instead of the `return` statement use an uninlined method call to let *vt* escape and contain

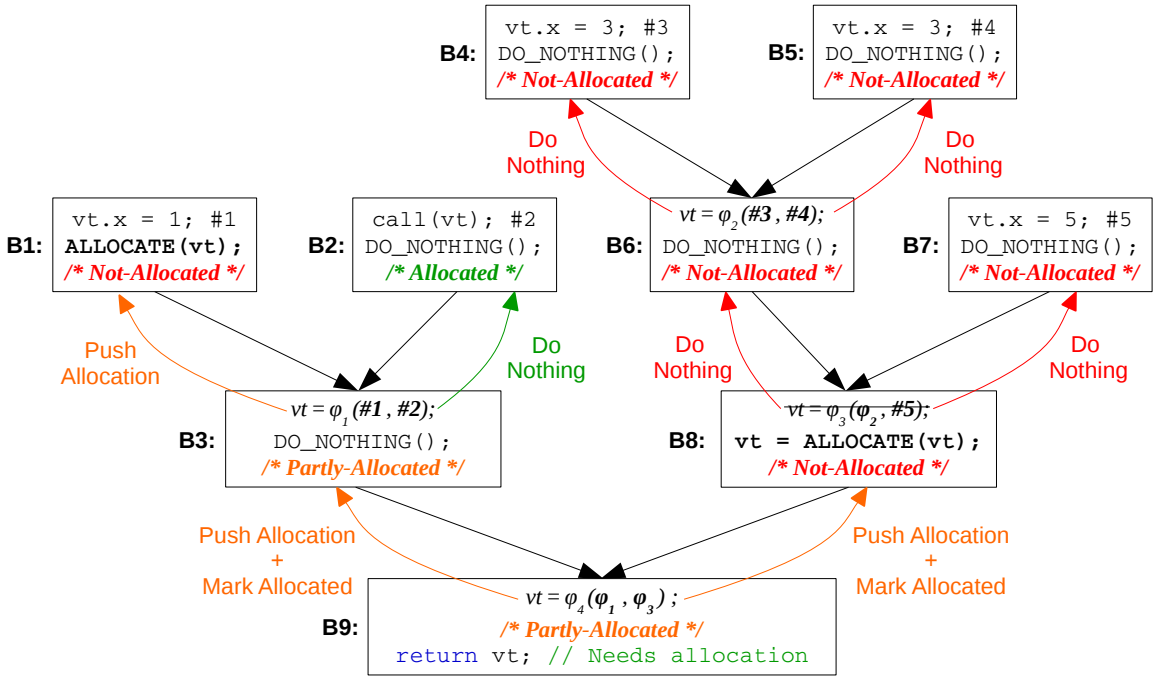


Figure 4.11: Extended example of pushing up allocations and allocation commands once an allocation of *vt* is required in block *B9*

4.3.5.1 Avoiding Allocation Pushes to Loop-Bodies

The presented approach in Section 4.3.5 works correctly but can create a significant unnecessary performance impact for partly-allocated phi functions inside loop-header blocks. Pushing up an allocation could mean that an allocation ends up in a loop-body, even though the allocation is not needed there. An example of this problem is shown in Figure 4.12. The variable *vtArg* was passed to the method as an argument and is assumed to be allocated in block *B1* (`m()` was not inlined). In the loop-body, represented by block *B3*, *vtArg* is set to a new instance due to a field assignment through `vwithfield`. Thus, at the end of the loop-body, *vtArg* is not allocated. Therefore, the created phi function in block *B2* for *vtArg* is marked as *partly-allocated* at the reevaluation (rule af2 and af4). This value type instance phi node is finally returned in block *B4* and thus needs an allocation. The phi function for *vtArg* pushes an allocation to all unallocated operands (i.e. not marked as *allocated*) according to the rule for partly-allocated phi nodes (rule A2). Since *vtArg* is already allocated in block *B1*, the allocation is only pushed to block *B3* into the loop-body. If the loop ends up being executed many times, the allocation is performed in each iteration, even though it conceptually belongs to block *B4*. This is a major performance impact that could be avoided.

However, simply allocating every time in the block after the loop when the loop-header phi is marked as *partly-allocated*, is not a good option either as shown in Figure 4.13. Block *B3* and

more blocks after block *B9*, then the information that φ_4 is properly allocated can be reused after the uninlined method call in block *B9* and possible successor blocks. φ_4 does not need to perform any additional allocation insertions.

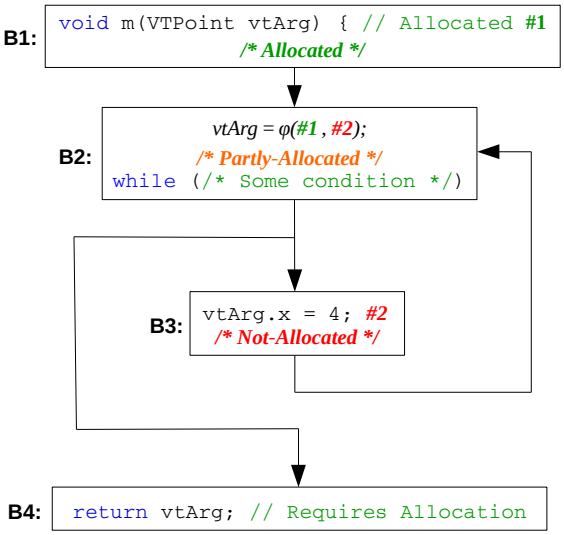


Figure 4.12: The problem of pushing allocations inside a loop-body, even though it is not used there

B6 are assumed to not reassign *vtArg* (i.e. the same instance as before). Block *B4* creates the phi function φ_1 for *vtArg* which is marked as *partly-allocated* since it merges a not-allocated and an allocated instance from block *B2* and *B3*, respectively (rule af2)[10]. The phi function φ_2 for *vtArg* in block *B5* is set to *partly-allocated* at the parsing stage of the block (rule af4). After the last operand is available, the reevaluation of the allocation-flag sets it again to *partly-allocated* due to the partly-allocated operand φ_1 (the last operand is ignored since it refers to itself recursively). Applying the same logic as in the previous example in Figure 4.12 would add an allocation in block *B7*. However, this would result in an unnecessary double allocation if the path over block *B3* is taken. The idea is to avoid an allocation push into the loop-body which would not have been done with the normal rule for partly-allocated phi nodes (rule A2)[11]. Thus, this additional allocation in block *B7* is redundant and should be avoided in this case.

Taking into account those thoughts allows the removal approach to be extended to include this loop-optimization with three additional rules to those already stated in the recursive allocation rules for phi functions:

[10]The C1 compiler additionally inserts block *B4* instead of directly merging block *B2, B3*, and *B6* into the loop-header block *B5*.

[11]Rule A2 would only insert an allocation in block *B3* for φ_1. The operand φ_2 refers to itself and thus is already processed by the allocation steps of φ_1. The implementation has to be careful to detect these cases and avoid an infinite loop by inspecting the last operand φ_2 in a cycle.

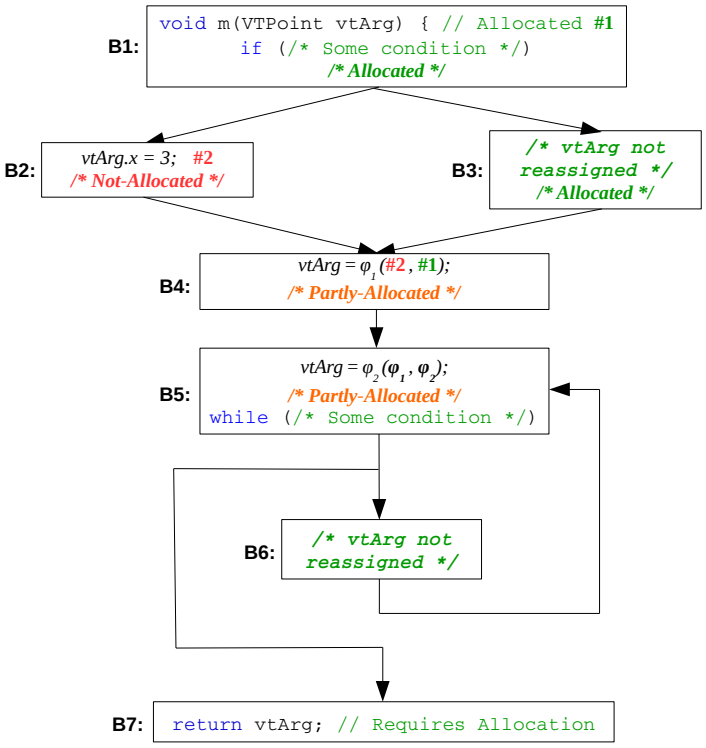


Figure 4.13: The problem of always inserting allocations after the loop if the loop-header phi function is partly-allocated

A6: If a phi function in a loop-header block is marked as *partly-allocated* and the last operand (i.e. the one from the loop-body) refers the phi node itself recursively (i.e. no new instance created for the variable), then the normal allocation insertion rule for *partly-allocated* phi nodes (rule A2) is applied.

A7: If a phi function in a loop-header block is marked as *partly-allocated* but does not refer the phi node itself recursively (i.e. a new instance is created in the loop-body), then an allocation is directly inserted in the block after the loop where it is required *if* the instance in the loop-body is not properly allocated (i.e. an unallocated instance or a phi function not marked as *allocated*).

A8: While the last operand of a phi function in the loop-header block is not yet known, this optimization cannot be applied (i.e. not applied at parsing the corresponding loop-body).

If a value type instance phi node is partly-allocated, then the allocation would normally (without those new rules) be pushed either to the loop-body or the other predecessor blocks of the loop-header. Rule A7 improves this by directly inserting an allocation in the block after the loop instead of pushing it into the loop-body. If the instance is already properly allocated inside the loop-body, then there is no need to insert an allocation directly after the loop since one of the predecessor blocks already contains an allocation (due to being partly-allocated which could result in a double allocation). Rule A6 avoids the double allocation problem shown in Figure 4.13. Obviously this approach does not work and does not make sense while parsing a loop-body since the last operand is not yet known. Thus, this optimization is only applied for completely parsed loops (rule A8).

4.3.6 Field-Phi Load Insertion

Reading the *LOAD_REQUIRED* constant from the buffer results in a direct insertion of a `FieldLoad` node to the HIR in the specified block. When setting up field-phi operands, it is beneficial to load the corresponding field values lazily instead of eagerly for the same reasons as stated in Section 4.3.4. However, this still leaves the question how a field load is inserted if the *LOAD_REQUIRED* constant is found as an operand of a field-phi function that is returned from the buffer. Before the field-phi can be used correctly, all its operands must have replaced all *LOAD_REQUIRED* constants recursively by actual field loads in the corresponding blocks[12]. The idea is to use the same approach as for the value type instance phi allocation in Section 4.3.5. A field-phi gets an *operands-known-flag* similar to the allocation-flag for value type instance phi nodes with the following states and rules:

of1: All non-phi operands and non-value-type phi operands are considered as being marked *all-known*.

of2: The *LOAD_REQUIRED* constant is considered as being marked *no-operand-known*.

of3: A field-phi node is marked as *all-known* if *none* of its operands contain (neither directly or recursively) the *LOAD_REQUIRED* constant (i.e. all operands are marked *all-known*).

of4: A field-phi node is marked as *partly-known* if *one* (but not all) of its operands contains (directly or recursively) the *LOAD_REQUIRED* operand (i.e. one operand but not all are marked *all-known* or at least one operand is already marked *partly-known*).

of5: A field-phi node is marked as *no-operand-known* if *all* operands contain (directly or recursively) the *LOAD_REQUIRED* constant (i.e. all operands are marked as *no-operand-known*).

of6: A field-phi node in a loop-header is marked as *partly-known* on parsing the loop-header and once the last operand is available, it is *completely* reevaluated and marked according to the other rules above. If the last operand refers to itself recursively, then it is ignored in the reevaluation.

As for the allocation-flag, the operands-known flag is set at the creation of new a field-phi node. The rules are very similar to those for the allocation-flag. A field-phi is again automatically marked as *partly-known* in a loop-header block since it is not clear whether the last missing operand represents a *LOAD_REQUIRED* constant or not. Thus, the rule is conservatively and sets it to *partly-known* to ensure correctness during the parsing stage of the loop-body together with the rules introduced in the next paragraph. Once all operands are known at the end of the loop-body, the field-phi is reevaluated again and set to one of the three states. If the last operand refers the field-phi itself recursively, then this operand is ignored for deciding the new status of the field-phi (due to a cycle). This information is now complete and can be used in successor blocks after the loop.

[12] Otherwise, the compiler cannot use the *LOAD_REQUIRED* constant as a field value in the further execution since it refers to a `FieldLoad` instruction. Accessing it as such results in an error.

Field Load Insertion Rules: The rules for inserting the field loads correctly are also similar to the ones for inserting a value type allocation in Section 4.3.5 upon a request to load a field value:

F1: A field-phi node marked as *all-known* does not need to insert a field load.

F2: A field-phi node marked as *partly-known* pushes a field load insertion command to each field-phi operand which is marked as *partly-known* or *no-operand-known* recursively and inserts a `FieldLoad` instruction for each operand containing the *LOAD_REQUIRED* constant in the corresponding block.

F3: A field-phi node marked as *no-operand-known* directly inserts a `LoadField` instruction in the current block if the instance to load from is the same on all possible incoming paths (i.e. on all operands) and is otherwise treated as a *partly-known* field-phi.

F4: During the field load insertion process, the involved *partly-known* field-phi nodes are marked as *all-known* and all newly inserted `FieldLoad` instances replace the old operands of the corresponding field-phis.

F5: If the field-phi node in a loop-header block is marked as *operands-known* but the last operand turns out to be not at the reevaluation, then a field load insertion is pushed towards the block of the last operand (i.e. to the loop-body).

The key difference to the allocation rules in Section 4.3.5 is that in rule F3 a field load is only directly inserted (instead of recursively inserting field loads) if the field load is performed on the *same* instance. Otherwise, it would be incorrect to directly insert a field load since the field-phi depends on different value type instances to load from. In this case, the field-phi is treated as being marked *partly-known* and the field loads are inserted accordingly for each operand. Another difference to the allocation rules is that a directly inserted field load with rule F3 does not need to update the local variables. The field-phi nodes are only present within the buffer. The other rules are analogously to the recursive allocation rules in Section 4.3.5.

Once again the situation can occur that a field-phi is already marked as *operands-known* due to a usage of the field inside a loop-body and thus marked as *operands-known* during the field load insertion process (rule F4) but the last operand is not known yet. This is very similar to the situation for setting up the last operand of an instance phi node discussed in Section 4.3.5. If the last operand at the reevaluation step of the operands-known flag is itself not marked *operands-known* but the field-phi itself is, then a field load must be inserted in the block represented by the last operand (i.e. the last block of the loop-body) for the same reasons.

An example similar to that for the allocation process in Figure 4.10 is shown for field loads in Figure 4.14. The method `m()` provides an already allocated value type instance *vtArg* (`m()` is not inlined). The field-phi created in block *B2* for *vt.x* sets its first operand to the *LOAD_REQUIRED* constant (abbreviated by "L_R") since the field value is unknown. The operand from the loop-body is not yet known at this stage of the parsing process. Therefore,

the field-phi is marked *partly-known* according to the operands-known-flag rule of4. In block *B3*, the field *vt.x* is used and thus triggers a field load insertion towards block *B1* due to rule F2 for partly-known field-phis. The *LOAD_REQUIRED* constant operand is replaced by the `FieldLoad` result and the field-phi is marked *operands-known*. Finally, *vtArg* is reassigned by a new value type instance that is assumed to be allocated over the uninlined call to `getAllocatedVTPoint()` (i.e. statically unknown field values). Once the loop-body is completely parsed, the field-phi function in block *B2* is reevaluated. It is already marked as *operands-known* but provides a *LOAD_REQUIRED* constant (i.e. no-operand-known according to rule of2) from the new assignment in block *B3*. Therefore, a field load is inserted at the end of the loop-body in block *B3* to provide a valid usage of *vt.x* in a possible next loop iteration.

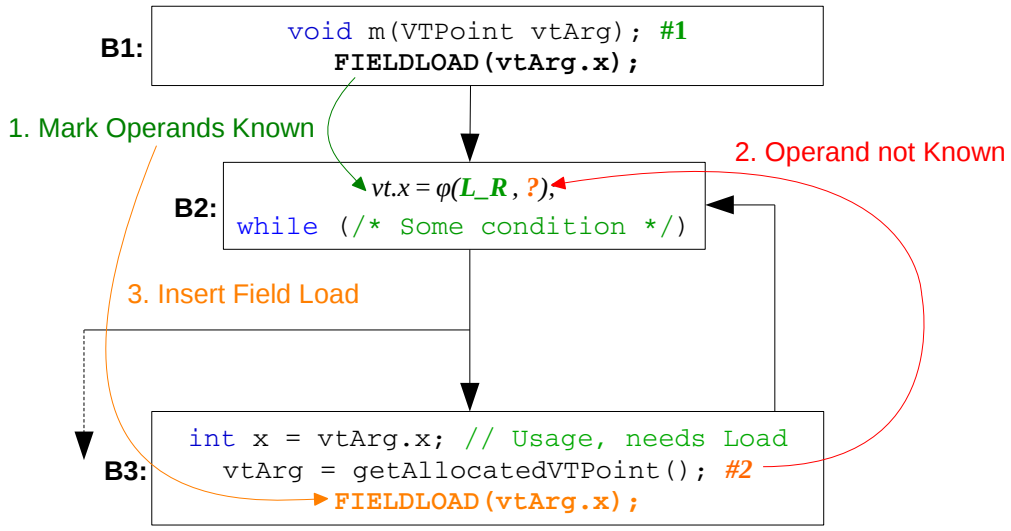


Figure 4.14: Example of the need to push a field load insertion to the block of the last loop-header phi operand if the field is used inside the loop-body

The extended example presented in Figure 4.11 works similarly for field-phi nodes with its rules from above and is shown in Figure 4.15. The value type pointed to by *vt* of type `VTPoint` is assumed to be passed as an argument to the unlined method and the variable itself is unmodified outside of the shown blocks. The only place where *vt* is modified is in block *B2*. A new value type is created with the field value 2 for *x*. Block *B3* creates the field-phi function φ_1 for *vt.x*. The first operand needs to be loaded from block *B1* and is represented by the *LOAD_REQUIRED* constant (abbreviated in the figure as "L_R"), while the other one is just a constant from block *B2* and implicitly known (i.e. treated as *all-known* due to rule of1). According to the operands-known-flag rule of4 this field-phi function is thus marked *partly-known*. Block *B6* merges block *B4* and *B5* which both need a load for *vt.x*. Hence, both operands of the phi function φ_2 contain the *LOAD_REQUIRED* constant and therefore it is marked *no-operand-known* (rule of5). Similarly, the phi function φ_3 is marked *no-operand-known*. Finally, in the block *B9*, the field *vt.x* is accessed and needs to be fully loaded. The buffer returns the field-phi φ_4 as a value which is marked *partly-known* due to rule of4 (φ_1 is marked as *partly-known*). Thus, a field load command is pushed up recursively to block *B3* and *B8*. The block *B3* only needs to insert a `FieldLoad` instruction in block *B1* (rule F2) since the first operand contains the *LOAD_REQUIRED* constant and the other operand is already a known

constant (i.e. treated as all-known). The phi function φ_3 in block *B8*, on the other hand, is marked as *no-operand-known*. According to rule F3, the `FieldLoad` instruction is directly inserted in block *B8* since both operands depend on the same value type instance represented by *vt*. Finally, the φ_4 field-phi can be used to assign x in block *B9*. In a last step, the φ_3-operand of φ_4 is replaced by the new `LoadField` instruction. During this process, each partly-known field-phi function that was involved contains now known operands. Therefore, they are marked as *all-known* due to rule F4. If the field *vt.x* is used again afterwards, it can take advantage of the information that φ_4 is already completely loaded. φ_4 does not need to perform any additional field load insertions.

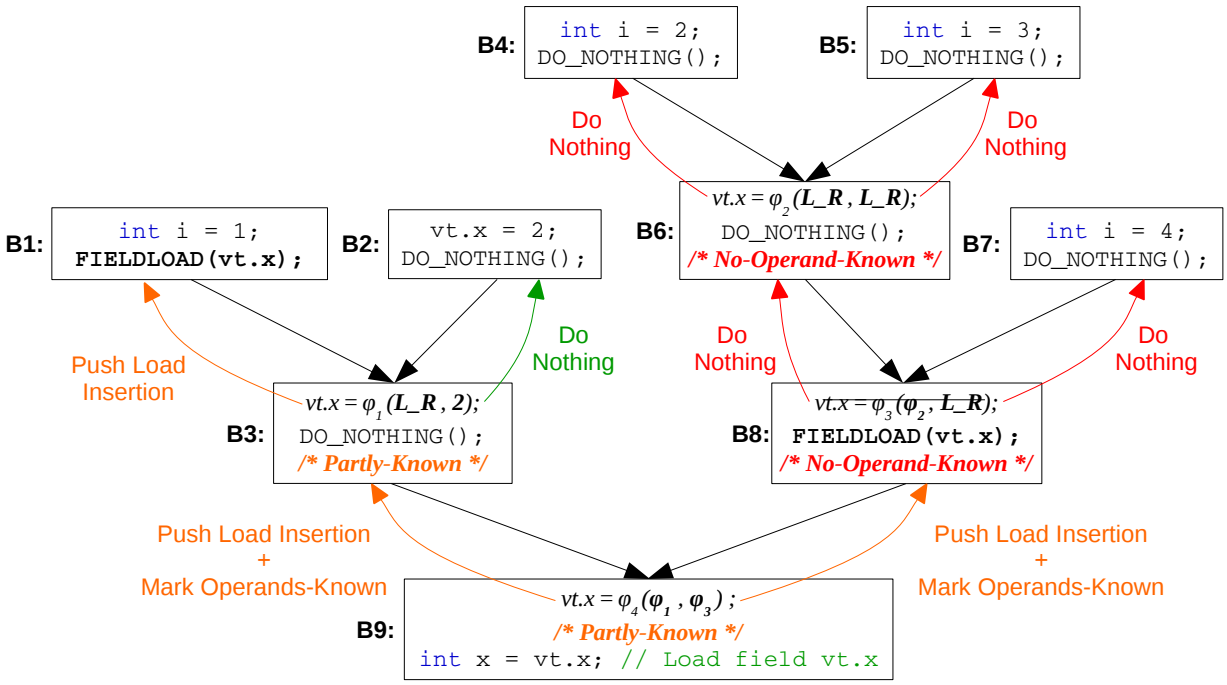


Figure 4.15: Example of pushing up field load commands and inserting loads once the field *vt.x* is used in block *B9*

4.3.6.1 Avoiding Field Load Pushes to Loop-Bodies

The optimization to avoid allocation pushes to loop-bodies that are not necessary, as described in Section 4.3.5.1, can also be applied similarly to field loads for the same reasons. However, a key difference is that this optimization can only be applied if all field loads would be performed on the same value type instance. Moreover, if there is a statically known field value in an operand directly or recursively, then the optimization cannot be applied since the statically known value needs to be used over the field-phi function and cannot be skipped and replaced by a direct field load[13]. The field load insertion approach is therefore extended to include this loop optimization with three additional rules to those already stated in the field load insertion rules for field-phi functions:

[13]This might raises the question if this optimization is needed at all. However, it can still be beneficial if a field-phi was marked as *no-operand-known* originally but was updated in the meantime to *all-known* due to a field value usage. Merging it with other field-phi functions marked as *no-operand-known* could result in a partly-known field-phi in a loop-header block. Using the field-phi after the loop can then benefit from this optimization by omitting an expensive field load insertion into the loop-body.

F6: If a field-phi function in a loop-header block is marked *partly-known* and the last operand (i.e. the one from the loop-body) refers to itself recursively (i.e. no new value assigned for the field), then the normal field load insertion rule for *partly-known* field-phis (rule F2) is applied.

F7: If a field-phi function in a loop-header block is marked as *partly-known* but does not refer to itself recursively and none of its operands contains a statically known value directly or recursively, then a field load is directly inserted in the block after the loop where it is required *if* all field loads would be performed on the same value type instance.

F8: While the last operand of a field-phi function in the loop-header block is not yet known, this optimization cannot be applied (i.e. not applied at parsing the corresponding loop-body).

Finding the correct instance to load from and checking if all operands depend on the same value type is not trivial if there are loops involved. The implementation also has to handle cases where an operand of a phi refers to itself. This is further explained in Chapter 5.

4.4 General Field Load Removal

Section 4.3 introduced a method-global buffer for field values to remove value type allocations and value type field value look ups from the heap. Nevertheless, there are situations where a field load is inevitable since the field value is statically unknown to the method. Some field load insertions can be avoided by the approach showed in Section 4.3.6, especially by moving a field load out of the loop-body. The idea is to also buffer the actual result of a field load to later reuse it. This was first approached by reusing the method-global buffer in the same way. However, it was later redesigned to use a separate buffer for each block that directly and only stores `FieldLoad` nodes. This removes possible update chains of many value type instance entries in the method-global buffer that rely on this result. An example is shown in Listing **??**. All three value type variables point to *vtArg*, which has unknown field values. They all update their x field resulting in three new value type instances. Finally, the field *vtArg.y* is used on line 9 which creates a `FieldLoad` instance. This new information now has to be propagated to all method-global buffer entries that depend on this value to update them. In this example, all three instances *vt1, vt2* and *vt3* need an update. This gets even more complex with additional field-phi nodes involved.

However, the additional optimization to insert field loads directly in a block and skipping the recursive field-phi load insertion process (see Section 4.3.6 for no-operand-known and Section 4.3.6.1 for partly-known field-phis as part of the loop optimization) does not work directly with the method-global buffer. Replacing the field-phi field entry in the method-global buffer for a value type instance by the new `FieldLoad` instance is erroneous in certain situations. For example, this replacement could be done in a block on one branch and afterwards another block on a completely different branch tries to access this field value of the very same value type instance. The method-global buffer provides the `FieldLoad` result which should have been executed already. However, the block in which this `FieldLoad` was created and buffered was not part of the current

Listing 4.9: Example for an update chain for the buffer for a new `FieldLoad`

```
1 // NOT inlined
2 __ValueFactory static VTPoint m(VTPoint vtArg) {
3   VTPoint vt1 = vtArg;
4   VTPoint vt2 = vtArg;
5   VTPoint vt3 = vtArg;
6   vt1.x = 3;
7   vt2.x = 4;
8   vt3.x = 5;
9   int y = vtArg.y;
10  // ...
11 }
```

execution path. This leads to an error. Therefore, the choice was made to use a completely separate `FieldLoad` buffer for each block (called *block-buffer*) instead of reusing the method-global buffer[14].

A block-buffer stores all `LoadField` nodes that were appended to the HIR for each field of a value type instance in the current block. This avoids wrong uses in other blocks that are not part of an execution path. Each time a field value of a value type is loaded for the first time from the heap, a new entry is created for that field of that instance. This book-keeping process is similar to the one in the global-method buffer. If a field is requested to be loaded a second time from the heap, the block-buffer can directly provide the result without performing another field load (i.e. adding another `LoadField` node).

However, this limits the usage in cases where a branch is involved as shown in Listing 4.10. The field load buffered on line 3 cannot be reused on line 5 since the `if`-branch is part of a new block. Therefore, the approach is improved to allow blocks to copy the buffered `LoadField` nodes from predecessor block-buffers if they meet the following conditions:

- If a block B has only one predecessor, then all `FieldLoad` nodes from the predecessor block-buffer can be copied into the block-buffer of B.
- If a block B has multiple predecessors, then the *conjunction* of all entries of all predecessor block-buffers are copied to the block-buffer of B.

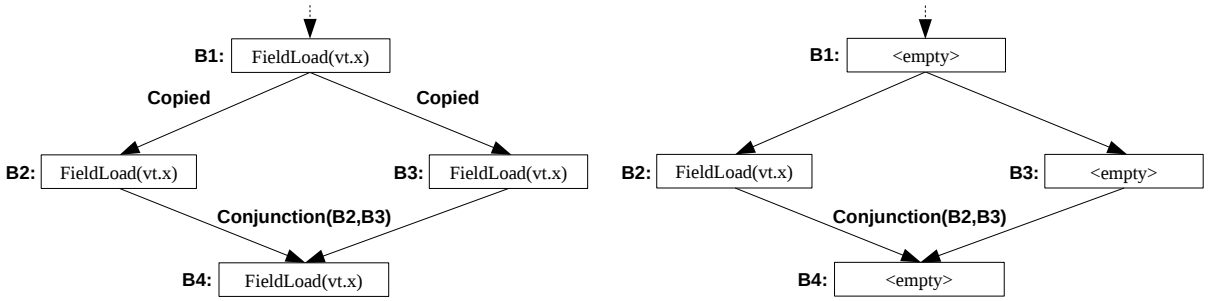
The *conjunction* of entries is a merge process that checks if all predecessor block-buffers of a block B contain the `LoadField` node for a specific field (i.e. the `LoadField` node exists on every execution path). If that is the case, the `LoadField` node is copied to the block-buffer of B for that field. Otherwise, an entry for a field is not copied. An example is shown in Figure 4.16. In Figure (a), the field value *vt.x* is assumed to be loaded in block *B1*. Therefore it is buffered

[14]The method-global buffer sets a new entry for a value type instance with all its field values (together with possible *LOAD_REQUIRED* constants) at the creation of the value type once and for all. They are not modified anymore later and thus all uses in subsequent blocks use the correct information. Replacing an already stored *LOAD_REQUIRED* constant by a `FieldLoad` node later can break this implicit property.

Listing 4.10: Example for the need to extend the block-buffer approach to pass information to successor blocks

```
1 // NOT inlined
2 __ValueFactory static VTPoint m(VTPoint vtArg) {
3   int x = vtArg.x; // Insert into block-buffer
4   if (/* some condition */) {
5     int x2 = vtArg.x; // New block, nothing buffered, perform load
6   }
7   // ...
8 }
```

as `FieldLoad(vt.x)` in the block-buffer. Since block *B2* and *B3* have only one predecessor, the entry for *vt.x* is copied to both block-buffers. Block *B4* has multiple predecessors. Thus, it takes the conjunction of the block-buffer entries of block *B2* and *B3*. Since the `FieldLoad(vt.x)` node exists in both predecessors block-buffers the conjunction function also copies the entry to the block-buffer of block *B4*. In Figure (b), on the other hand, the field load is only performed in block *B2*. Therefore, the conjunction over the block-buffer from block *B2* and *B3* is empty and nothing is copied to the block-buffer of block *B4*.



(a) Field load in block *B1*

(b) Field load in block *B2*

Figure 4.16: Two different outcomes of taking the conjunction of buffered `FieldLoad` nodes from block *B2* and *B3*

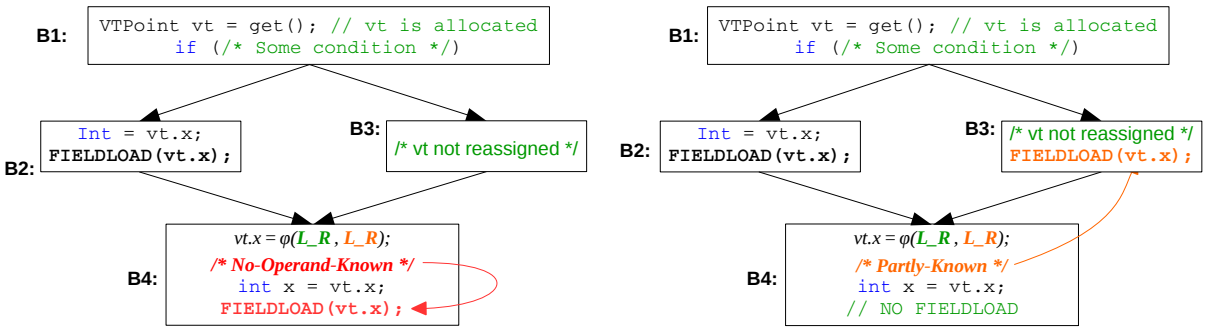
Merging block-buffers with the conjunction function is necessary, even though field-phi functions are present. Normally, a field load automatically uses the field-phi function. However, due to the optimization to skip the recursive field-phi load insertion process, as described in Section 4.3.6 for no-operand-known and in the loop-optimization in Section 4.3.6.1 for partly-known field-phis, a field load request in a block could possibly reuse an earlier buffered result from a predecessor block which was copied over the conjunction function to the current block-buffer.

The method-global buffer of the allocation and field load removal approach might be later merged with these block-buffers. This exploration is left as future work (see Section 7.1).

4.4.1 Adaptation for No-Operand-Known

The approach presented in Section 4.3 performs a field load for a read *LOAD_REQUIRED* constant from the method-global buffer. However, it does not update the buffer with the field load result due to the reasons explained earlier. Thus, a field-phi could contain an operand with a *LOAD_REQUIRED* constant while the corresponding block has already performed a field load. This is not a problem if the field-phi function is marked as *partly-known* since the normal field load insertion process (see rule F2 in Section 4.3.6) ensures that no redundant field loads are inserted with the help of the new block-buffers. However, a field-phi could be marked as *no-operand-known* which means that all operands contain the *LOAD_REQUIRED* constant (rule of5 in Section 4.3.6). As a consequence, the corresponding field load insertion rule F3 (see Section 4.3.6) directly inserts a field load in the current block (i.e. skips the recursive field load insertion process for partly-known field-phis). This might result in a repeated field load if it was already performed in one of the predecessor blocks. Therefore, the process for setting the operands-known-flag for a field-phi additionally queries the block-buffer of each block of an operand. If one of them contains a buffered field load, then the field-phi is marked *partly-known* to apply the normal recursive field load insertion rule F2 for partly-known field-phis (see Section 4.3.6). This avoids the additional field load insertion for *no-operand-known* marked field-phis. However, the field-phi cannot be directly marked as *all-known* since the field-phi still contains a *LOAD_REQUIRED* constant.

An example of this optimization is shown in Figure 4.17. Block *B1* contains an allocated value type *vt* with unknown field values which is assumed to not be reassigned in block *B3*. The normal approach without block-buffers, shown in Figure (a), marks the field-phi in block *B4* as *no-operand-known* (both operands contain the *LOAD_REQUIRED* constant) and thus applies the rule to directly insert a field load into block *B4* (rule F3 in Section 4.3.6). However, if the condition in block *B1* is true, a field load is unnecessarily performed twice in block *B2* and *B4*. With the optimization described above, shown in Figure (b), the field-phi in block *B4* is marked as *partly-known* since there is already a field load result available in the block-buffer of block *B2*. Thus, the normal rule for partly-known field-phis is applied (rule F2 in Section 4.3.6) and pushes a field load into block *B3* and avoids a repeated field load in block *B4*.



(a) Two field loads on the path over block *B2*

(b) Only one field load on each path

Figure 4.17: Exploiting the new block-buffer concept to avoid unnecessary field loads for field-phis marked as *no-operand-known*

4.5 On-Stack-Replacement

The C1 compiler can compile a method which contains a loop that exceeded its back-branch counter. A special OSR-entry block is created which is directly executed after the end of a loop iteration in the interpreter and once the method is OSR-compiled. The necessary information about the last state in the interpreter is setup for the compiler in the OSR-entry block (e.g. local variables, state of the stack etc.). Afterwards, the execution continues in the loop-header block which evaluates the condition of the loop. If the method is called again, it enters the normal standard entry block. The OSR-entry block is only used at the transition from the interpreter to the C1 compiler. There are a few things which have to be considered for the value type approach:

- An additional phi operand has to be added for the field-phi functions at the loop-header block.
- All value type instances from the OSR-entry block are allocated.
- All field values from the OSR-entry block are unknown and need to be loaded from the heap.

When a method is OSR-compiled, it is usually also scheduled for normal compilation afterwards. The OSR version is used as long as the method is executed and can be replaced by the normally compiled version (if already available) once the method is finished and later invoked again.

5 Implementation

This chapter presents the concrete implementation of the value type design decisions presented in Chapter 4. The introduced concepts are mapped to code parts and are explained in more detail together with involved classes and methods. The evaluation of the code follows in Chapter 6. The introductory Section 5.1 first gives a brief overview of how the C1 compiler compiles a method and which classes and methods are invoked. This is important to understand the context and to follow the explanations of the contributed code for the value type implementation in the subsequent sections. The structure of this chapter follows the one used in Chapter 4 with additional explanations of cleanup steps in Section 5.6 and about the exploration of deoptimization support with value types in the C1 compiler in Section 5.8.

The code for the HotSpot™ JVM is located inside the `hotspot/src` folder. All references to code files in the following sections use this prefix. All changes of this thesis can be found in the provided patch[1] to the code of changeset 13528:f017570123b2 in the HotSpot™ JVM mercurial repository[2] from 21. August 2017 which is called *baseline* or *baseline version*. In the development process, the special Eclipse version for C++ developer was used together with gdb and the c1visualizer for debugging (see Section A.2 for a description of those tools).

5.1 Overview of the C1 Compiler

This first section gives an overview of how the C1 compiler compiles a method and presents the location where the value type implementation starts to take action in the context of the existing code state right before the start of the thesis (i.e. the baseline version). Important existing classes for the value type implementation are quickly introduced. However, concrete code details about value types together with a description of newly added classes and methods are explained later in subsequent sections.

The HotSpot™ JVM provides several `CompilerThread`[3] instances which are used for compilation. Each compiler thread contains a separate `CompileQueue`[4] to enqueue incoming compile requests

[1] Link: `http://cr.openjdk.java.net/~thartmann/thesis_hagedorn/webrev/hotspot.patch`

[2] Repository: `valhalla/valhalla10-old/hotspot`; Changeset link: `http://hg.openjdk.java.net/valhalla/valhalla10-old/hotspot/rev/f017570123b2`

[3] Defined in `/share/vm/runtime/thread.hpp`

[4] Defined in `/share/vm/compiler/compileBroker.hpp`

represented by the `CompileTask`[5] class. The compiler threads are constantly polling for new compilation tasks in the `CompileBroker::compiler_thread_loop`[6] method as long as the JVM is running. Once a task is ready it is dequeued and delegated to the corresponding compiler specified by the task. A C1 task is processed by the `Compiler::compile_method`[7] method. The `Compilation`[8] class finally compiles the Java method inside the `Compilation::compile_java_method` method. This is the location where the methods for the HIR and the LIR creation are called. At the end of the method, assembly code is emitted based on the LIR.

The folder `/share/vm/ci` represents a compiler interface which defines sharable information between the compilers, such as class or method information. Since the C2 compiler already supports value types, there are reusable structures defined for value types in general. The folder `/share/vm/c1` contains the platform-independent implementation of the C1 compiler. The machine-dependent code for generating specific LIR instructions and assembly code with the C1 compiler is located in a subfolder of the `/cpu` folder[9]. The relevant files for the C1 compiler are prefixed by "`c1_`".

5.1.1 HIR

The HIR is created inside the `Compilation::build_hir`[10] method and is represented by the `IR`[11] class which consists of an `IRScope`[12]. This scope can refer to all inlined methods which are themselves `IRScope` instances. However, there is only one scope without any inlining. The `IRScope` contains all the required information for the compilation and triggers the control flow graph creation performed inside the `GraphBuilder`[13] class. The `GraphBuilder` also represents the core class for the entire value type implementation described later in this chapter. It is responsible to setup all basic blocks by iterating over all bytecodes. A block is represented by a linked list of HIR instruction. Those are defined as separate subclasses of the super class `Instruction`[14]. A block always starts with a `BlockBegin`[15] instance and ends with a `BlockEnd`[16] instance (both are subclasses of `Instruction`) once the block is completely parsed. Whenever the control flow is split or merged by a bytecode instruction, a new block is created by defining a new `BlockBegin` instance as a start node. The `GraphBuilder::iterate_bytecodes_for_block` method is responsible to read all bytecodes instructions from the bytecode stream of the method and process them accordingly. The requested HIR instructions are linked into the currently active block. The switch statement over the read bytecodes in the `GraphBuilder::iterate_bytecodes_for_block` method

[5] Defined in `/share/vm/compiler/compileTask.hpp`
[6] Defined in `/share/vm/compiler/compileBroker.hpp`
[7] Defined in `/share/vm/c1/c1_Compiler.hpp`
[8] Defined in `/share/vm/c1/c1_Compilation.hpp`
[9] This thesis only supports the x86-64 architecture and thus modifications were done to the `/cpu/x86/vm/` folder.
[10] Defined in `/share/vm/c1/c1_Compilation.hpp`
[11] Defined in `/share/vm/c1/c1_IR.hpp`
[12] Defined in `/share/vm/c1/c1_IR.hpp`
[13] Defined in `/share/vm/c1/c1_GraphBuilder.hpp`
[14] All defined in `/share/vm/c1/c1_Instruction.hpp`
[15] All defined in `/share/vm/c1/c1_Instruction.hpp`
[16] All defined in `/share/vm/c1/c1_Instruction.hpp`

is the starting point of the value type implementation. The new value type bytecodes are added as new case statements together with the code to create HIR instructions for them.

During the parsing stage of bytecodes, the `GraphBuilder` keeps track of the current block (field `_block`) and the current *state* (field `_state`). A state is described by the `ValueStack`[17] class which contains information about the current local variables, the expression stack (i.e. equivalent to the bytecode operand stack), and locks. This information is stored in HotSpot™ specific resizable arrays defined by the `GrowableArray`[18] class. A state can be *split* by various `StateSplit`[19] instructions. They modify the state and therefore make a copy of the current state and assign it to the `_state` field of the instruction. Those splits are important to keep track of different snapshots of the state inside the block to get the correct values at a specific point later, for example, in case of a deoptimization and the associated transfer back to the interpreter. The `BlockBegin` and the `BlockEnd` instructions are both subclasses of `StateSplit`.

Each HIR instruction has an associated type defined in the `ValueType`[20] class and a reference to the next instruction. Moreover, each `Instruction` instance contains a unique instruction id and an `LIR_Opr` field that is later filled with LIR specific information (i.e. an LIR instruction operand). After the HIR is created, several optimizations, such as null check elimination, are applied from the `Optimizer`[21] class before the control goes back to the `Compilation::compile_java_method` method.

5.1.2 LIR and Assembly Code

The `Compilation::compile_java_method` calls the `Compilation::emit_LIR` method to build the LIR after the HIR was created. The `LIRGenerator`[22] generates the LIR from the HIR with virtual registers (also called *LIR operands*). The phi functions are removed and replaced by pushing instructions to the predecessor blocks accordingly. Some HIR instructions, such as creating an object, require machine dependent LIR nodes which are specified in a machine specific source file inside the `/cpu` folder[23]. The LIR operands are afterwards assigned to physical registers by the `LinearScan`[24] class. Finally, the `Compilation::compile_java_method` calls the `Compilation::emit_code_body` method to emit machine specific assembly code.

[17] Defined in `/share/vm/c1/c1_ValueStack.hpp`
[18] Defined in `/share/vm/utilities/growableArry.hpp`
[19] Subclass of `Instruction` defined in `/share/vm/c1/c1_Instruction.hpp`
[20] Defined in `/share/vm/c1/c1_ValueType.hpp` and not to be confused with actual value types which have nothing to do with the `ValueType` class directly
[21] Defined in `/share/vm/c1/c1_Optimizer.hpp`
[22] Defined in `/share/vm/c1/c1_LIRGenerator.hpp`
[23] For the x86-64 architecture used in this thesis the source file `/cpu/x86/vm/c1_LIRGenerator_x86` defines the required methods.
[24] Defined in `/share/vm/c1/c1_LinearScan.hpp`

5.2 Changes to the Intermediate Representations

As described in Section 4.1, a new HIR instruction `NewValueTypeInstance`[25] is introduced. It is a subclass of `StateSplit` similarly to the `NewInstance`[26] class for objects. It contains a flag `_allocated` to mark the instance to be either allocated or not and a field `_depends_on` to have a reference to an allocated instance with statically unknown field values from which a load must be performed if the instance was created from it with `vwithfield` (see Section 4.3.1). The lookup might need to be recursive for many applications of `vwithfield` for a variable. Thus, an additional `NewValueTypeInstance::depends_on` method can be used to perform this task. The instance points to itself if it does not depend on another instance (i.e. no field load must be performed from the heap).

The `Phi`[27] class is used for all phi nodes in the HIR. Value type specific modifications are directly added[28]. To differentiate between normal phi functions, value type instance phis, and value type field-phis, two flags `_value_type_instance` and `_value_type_field_phi` are set accordingly[29]. As for the `NewValueTypeInstance` class, the `Phi` class contains an `_allocated` flag which can be set to the different states described in Section 4.3.5 in case of a value type phi instance. This is done through the internal `enum Allocation` in the `Phi` class. If a `Phi` instance represents a field-phi, the `_operands_known` field is used to indicate its state depending on the phi operands described in 4.3.6. This is done through the `enum OperandsKnown`, which is also part of the `Phi` class.

The only needed changes for the LIR is the mapping of the new `NewValueTypeInstance` class to the methods that already exist for the `NewInstance` inside the `LIRGenerator`[30] class and the replacement of the different implementation of value type field-phi functions (see Section 5.4.2). The assembly generator, which processes the LIR, maps everything related with value types to objects.

5.3 Unoptimized Value Type Implementation

The first unoptimized value type approach described in Section 4.2 treats all value types as normal objects without any optimizations. As shown in Section 5.1, the switch statement inside the `GraphBuilder::iterate_bytecodes_for_block` method is the entry point of the value type implementation. Additional case statements for the value type bytecodes `vdefault, vwithfield, vload, vstore` and `vreturn` are added together with the corresponding methods `GraphBuilder::new_value_type_instance, GraphBuilder::vwithfield, GraphBuilder::vload, GraphBuilder::vstore`, and `GraphBuilder::vreturn`, respectively, in-

[25] Defined in `/share/vm/c1/c1_Instruction.hpp`
[26] Defined in `/share/vm/c1/c1_Instruction.hpp`
[27] Defined in `/share/vm/c1/c1_Instruction.hpp`
[28] At a later stage the `Phi` class might be split in a class for normal phis, value type instance phis and value type field-phis (see Section 7.1).
[29] Those could be later merged together to a single field which uses an `enum` since a phi is always either an instance phi or a field-phi but never both at the same time. This could avoid some unintentional wrong uses (see Section 7.1).
[30] Defined in `/cpu/x86/vm/c1_LIRGenerator_x86`

side the GraphBuilder class. The existing method GraphBuilder::access_field for the getfield bytecode case statement is adapted for value types to call the new GraphBuilder::vgetfield method for them. Except for the vwithfield bytecode, the implementations for the object bytecodes are mostly reused.

To differentiate between the unoptimized implementation and the one with optimizations, a JVM flag OptimizeValueTypes (see Appendix A.3 for an explanation of its usage) is used as a guard in those new methods. For the GraphBuilder::vgetfield and GraphBuilder::vwithfield method, there is an additional GraphBuilder::vgetfield_unoptimized and GraphBuilder::vwithfield_unoptimized version, respectively, for the unoptimized approach. All these methods for the unoptimized implementation are described in the following:

- **new_value_type_instance:** This method is very similar to the object method GraphBuilder::new_instance. It creates a NewValueTypeInstance instance (instead of a NewInstance) and appends it to the current block with GraphBuilder::append_split. It calls the GraphBuilder::append_with_bci method internally which further sets up the state, copies the current state to a StateSplit instruction to be appended and ensures, with the help of a hash table, that a node is not added again if it is already linked. At last, the NewValueTypeInstance instance is pushed back on the stack represented by the ValueStack class described in Section 5.1.
- **vload, vstore and vreturn:** Those methods simply call the methods already used for the other load, store and return bytecodes: GraphBuilder::load_local, GraphBuilder::store_local and GraphBuilder::method_return, respectively.
- **vgetfield_unoptimized:** This method is an adaptation and simplification for value types of the _getfield case statement inside the GraphBuilder::access_field method for general field accesses.
- **vwithfield_unoptimized:** This is the only method in the unoptimized implementation that needs a different implementation since there is no similar bytecode that creates an object, assigns a field, and copies the other field values from another instance. The idea is to loop over all field values, create a LoadField instruction for each field and append it with the GraphBuilder::append method to the HIR. The LoadField instance is directly used to create a StoreField instruction[31] which is also appended directly with GraphBuilder::append. After the loop, the specified field read from the bytecode stream field_modify gets the value val assigned that was popped from the stack at the start of the method. This is again done with a StoreField instruction.

Throughout the entire unoptimized implementation, the type instanceType is used instead of the new valueTypeType[32] since the value types are entirely treated as objects. For an improved readability, the expressions "NewValueTypeInstance" and "NewInstance" are used instead of the

[31] Both defined in /share/vm/c1/c1_Instruction.hpp

[32] Both defined in /share/vm/c1/c1_ValueType.hpp while the latter one is used in the optimization approach for value types

confusing but technically correct versions "NewValueTypeInstance instance" and "NewInstance instance" in the following.

5.4 Allocation and Field Load Removal

This section maps the design decisions presented in Section 4.3 to the implemented code and explains why they were made. The same methods GraphBuilder::new_value_type_instance, GraphBuilder::vload, GraphBuilder::vstore, and GraphBuilder::vreturn as for the bytecodes presented in Section 5.3 are used but are guarded with the OptimizeValueTypes flag for this optimization. For the getfield and vwithfield bytecode, the optimization specific method versions GraphBuilder::vgetfield and GraphBuilder::vwithfield are implemented. The basic functionality of those methods is the same as in the unoptimized implementation but they contain additional code for the new buffer concepts presented in this section for the lazy allocation (see Section 5.4.3) and the lazy insertion of field value loads (see Section 5.4.4) together with the block-buffer presented in Section 5.5. The additional changes to the bytecode processing methods[33] are mentioned throughout the discussion of those features in the following sections and paragraphs.

5.4.1 Buffering Field Values

The existing C1 implementation provides a (method-global) MemoryBuffer _memory instance and FieldBuffer[34] class to perform some basic store and load instruction elimination for object fields. However, its support is very limited. Those two classes are reused and extended to provide a method-global buffer, introduced in Section 4.3.1, for the value type field values. For simplicity, the following sections will just refer to the *buffer* or to the MemoryBuffer when actually meaning the _memory field of the GraphBuilder class.

The MemoryBuffer keeps track of all newly created value type instances in a GrowableArray _newvaluetypes of Instruction. The same index of such an instance is used as an index to the second GrowableArray _newvaluetypefields of FieldBuffer that stores the corresponding field value array for a value type instance. The FieldBuffer class itself provides a GrowableArray _values of Instruction to buffer the field values. The offset of a field is used as a unique index.

5.4.1.1 Insert New Buffer Entries

A new buffer-entry (i.e. a new FieldBuffer instance) has to be added whenever a new value type instance is created by a vdefault or a vwithfield bytecode. The corresponding methods GraphBuilder::new_value_type_instance and GraphBuilder::vwithfield to process them have an additional statement to store the value type into the buffer compared to their unoptimized implementation (see Section 5.3). The GraphBuilder::new_value_type_instance method calls the simplified equally named MemoryBuffer::new_value_type_instance method

[33] GraphBuilder::new_value_type_instance, GraphBuilder::vload, GraphBuilder::vstore, GraphBuilder::vreturn, GraphBuilder::vgetfield and GraphBuilder::vwithfield

[34] Both defined in /share/vm/c1/c1_GraphBuilder.hpp

which only creates an empty `FieldBuffer` as part of the lazy allocation approach described in Section 4.3.1.1. The `GraphBuilder::vwithfield` method, on the other hand, uses the additional `MemoryBuffer::store_value_type_field` and `MemoryBuffer::load_on_value_type` methods to fetch field values from the buffer of the old instance and store them to the entry of the new instance together with the new specified field value in the `vwithfield` bytecode.

5.4.1.2 Read Buffered Field Values

A field value request to the `MemoryBuffer` is handled by the `MemoryBuffer::load_on_value_type` method which expects a value type instance and a field descriptor (`ciField`[35] class). First, it checks if the `_newvaluetypes` array contains the value type instance. If that is not the case, the value type was not newly created in this method (i.e. already allocated with unknown field values as shown in Section 4.3.1.2). Thus, the constant `Memory::LOAD_REQUIRED`[36] is returned, which is defined inside the `MemoryBuffer` class, indicating that nothing is buffered and a field load must be performed with `FieldLoad`. Otherwise, the index returned by the lookup of the value type instance is used to query the `_newvaluetypefields` array which provides the buffered value. To support lazy buffering (see Section 4.3.1.1), the value `NULL` read from a `FieldBuffer` returns a new default value instruction `Constant`[37] according to the field type instead. The `GraphBuilder::vgetfield` method inserts the returned buffered `Instruction` into the HIR or creates and appends a `LoadField` instance for the `MemoryBuffer::LOAD_REQUIRED` result accordingly (or fetches an already buffered `FieldLoad` from the new `BlockBuffer`[38] class for the approach shown in Section 4.4).

5.4.2 Branches with Phi Nodes

5.4.2.1 Adaptations for Field-Phi Nodes

While the value type instance phis can reuse the existing implementation of phi functions in the C1 compiler, the field-phis cannot. The reason is that a phi function for a local variable is defined in such a way that the operands always correspond to the value found at the same index in the local variable array inside the state (i.e. `ValueStack` instance) of the last `BlockEnd` instruction of the predecessor blocks. However, a field-phi (represented by a `Phi` instance) is not stored in the local variable array but instead in the `MemoryBuffer`. Therefore, the `Phi` class is extended by a custom `GrowableArray _value_type_field_operands` to store all operands of a field-phi. The associated `Phi::operand_at` and `Phi::operand_count` methods are adapted to use the information from this new array for field-phis accordingly.

The existing approach to remove the phi functions later in the LIR generation in the `LIRGenerator` class needs to be adapted for field-phis, too. The operands need to be fetched directly from the

[35] Defined in `/share/vm/ci/ciField.hpp`

[36] This constant is implemented as a `LoadField` pointer pointing to a constant value and thus is not a real instance. It should not be dereferenced as it would result in an a runtime error. This could be improved to point to a dummy instance later (see Section 7.1).

[37] Defined in `/share/vm/c1/c1_Instruction.hpp`

[38] Defined in `/share/vm/c1/c1_GraphBuilder.hpp`

`Phi` instance (i.e. the `_value_type_field_operands` array) instead of the predecessor blocks. Therefore, the new method `LIRGenerator::move_to_value_type_field_phi` is created to move the necessary instructions to the corresponding predecessor blocks. This method is called for all field-phis which have their `_operands_known` field not set to `OperandsKnown::operand_known` (see Section 5.2) or have its special flag `_field_phi_used` not set (see Section 5.6.3)[39]. Otherwise, the field-phi is never used (i.e. dead) and does not need to process its operands values. The method is called from the `LIRGenerator::move_to_phi` method which performs all move operations for the other phis (including those for value type instances). These are the only significant modifications to the LIR together with the ones mentioned in Section 5.2.

5.4.2.2 Normal Phi Setup

The existing implementation of the C1 compiler copies the state (i.e. `ValueStack` instance) of a `BlockEnd` instruction to the initially empty state of a `BlockBegin` instance which it points to in the parsing process. A new phi node is then created whenever a `BlockEnd` instruction points to a `BlockBegin` instance which already has an existing state. The local variables and stack values in the `ValueStack` instance are replaced by phi functions (i.e. `Phi` instances). This needs to be adapted for value type instance phis and field-phis since they have special flags assigned (see Section 4.3.5 and Section 4.3.6) and the field-phis are treating the operands completely differently.

Before the bytecodes of a new block are iterated by the `GraphBuilder::iterate_bytecodes_for_block` method, the phi nodes for the value types are set up by the new `GraphBuilder::setup_value_type_phis` method. This is done when a block has two or more predecessors or constitutes a loop-header. All local variables are iterated and processed if they represent a value type instance[40]. The local variables are passed and merged from the predecessor block states at the end of the block (i.e. `BlockEnd` instruction) inside the existing `GraphBuilder::try_merge` method executed after a block is fully parsed by the `GraphBuilder::iterate_bytecodes_for_block` method. All kinds of dead local variables are removed in this process.

In the first step of executing the `GraphBuilder::setup_value_type_phis` method, a `Phi` instance is created for the value type instance itself with its `_value_type_instance` flag set. All operands are iterated to properly update the `_allocated` flag of the `Phi` instance, if needed, to `Allocation::partly_allocated` or `Allocation::allocated` (see Section 5.2). Otherwise, it is left at its default value `Allocation::not_allocated` assigned at the creation of the phi node. If the current block is a loop-header, then the flag is automatically set to `Allocation::partly-allocated` for reasons explained in Section 4.3.5 since the last operand is not yet known.

[39] Normal no-value-type phi nodes have their `_operands_known` flag set automatically to `OperandsKnown::all_known` at their creation.

[40] This process should be later extended to also support this setup for stack values. However, with the usage of `javac` everything is stored in variables since it would not be accessible from the code otherwise. This extension is left as future work as described in Section 7.1.

In the second part of the method, the field-phis are created. Each field of the value type instance needs a new `Phi` instance with its `_value_type_field_phi` flag set. To fill the operands, the predecessor blocks have to be iterated. The value type instance found there is used as an index into the `FieldBuffer` array in the `MemoryBuffer` to fetch the appropriate field values with the `MemoryBuffer::load_on_value_type` method. If the `MemoryBuffer::LOAD_REQUIRED` constant is returned, then it is also directly stored as an operand value. The convention is to set the field value from the *i-th* predecessor block as the *i-th* operand of the new field-phi. After this setup, the field-phi is marked as either `OperandsKnown::all_known`, `OperandsKnown::partly_known`, or `OperandsKnown::no_operand_known` for normal blocks and as `OperandsKnown::partly_known` for loop-header blocks as described in Section 4.3.6. Additionally, if an operand gets the `MemoryBuffer::LOAD_REQUIRED` constant assigned but the corresponding `BlockBuffer` (see Section 5.5) already contains a `FieldLoad` instruction for that field (i.e. already loaded), then the field-phi is marked as `OperandsKnown::partly_known` with the help of the `BlockBuffer::check_if_exists` method[41] (see Section 4.4.1).

As a last step, the new field-phi instances are stored in the `MemoryBuffer`-entry (i.e. `FieldBuffer` array) for the newly created value type instance `Phi` node. The field-phis are also pushed to the state of the `BlockBegin` instance which represents this block[42]. They are not part of the local variables and thus have an own `GrowableArray _value_type_field_phis` inside the `ValueStack` class for an easier access later to process them all[43].

5.4.2.3 Setup the Last Operand

The last operand of a phi function in a loop-header block is not yet known at the time when the block is first visited and parsed. This decision can only be made once the entire loop-body is processed. At the end of the `GraphBuilder::iterate_bytecodes_for_block` method, when all bytecodes are processed in the current block, the states are merged with all successor blocks. If a successor is an already parsed loop-header block and has some unfinished value type phis (block marked with the new `BlockBegin::has_pending_vt_phi_operands` flag which is set in the `GraphBuilder::setup_value_type_phis` method), then the new method `GraphBuilder::setup_remaining_operands` is called to set up the last missing operands.

The structure is similar to the one used in the `GraphBuilder::setup_value_type_phis` method. The local variables already contain the value type instance phi nodes created in the first visit of the loop-header block. They work in a similar way to the normal phi functions and thus only need to update their `_allocated` flag with the new information of the last available operand. As described in Section 4.3.5, the flag is completely reevaluated. In this process, the last operand is excluded if it refers itself recursively. This is done with the operations including the local `depends_on_itself` flag. If the `Phi` instance, however, was already marked as

[41] An explanation of the method follows in Section 5.5.

[42] The `BlockBegin` instruction is always the first instruction in the block and at this stage the only existing instruction in a block anyways, as the block is only about to be parsed.

[43] Instead of iterating through all `MemoryBuffer` entries

`Allocation::allocated` but the reevaluation of the phi node reveals that it should be marked as `Allocation::partly_allocated`, then the last operand needs to be allocated (see rule A5 in Section 4.3.5). The `Phi` instance escapes the method scope in the loop-body and was replaced by another instance in the meantime. Thus, an allocation needs to be inserted in the block of the last operand (i.e. the loop-body) with the new `GraphBuilder::insert_value_type_instance_at_end` method (see Section 5.4.3) to ensure a correct usage on the next iteration (see Section 4.3.5). In all other cases, the phi node is marked with the result from the reevaluation step to either `Allocation::allocated`, `Allocation::partly_allocated`, or `Allocation::not-allocated`. Additionally, if an operand contains the `MemoryBuffer::LOAD_REQUIRED` constant but the corresponding `BlockBuffer` (see Section 5.5) already contains a `FieldLoad` instruction for that field (i.e. already loaded), then the field-phi is marked as `OperandsKnown::partly_known` with the help of the `BlockBuffer::check_if_exists` method[44] (see Section 4.4.1).

In the next step, all field-phis of the value type instance phi are processed. The field values are loaded from the last loop-body block with the help of the `MemoryBuffer` and inserted as last operand. Afterwards, the field-phi is completely reevaluated by looping over all operands. The last operand is again ignored, with the operations including the `depends_on_itself` flag, if it refers itself recursively. Similarly, as for the value type instance phis, a field load insertion is pushed to the block of the last operand if the reevaluation would not mark the field-phi as `OperandsKnown::all_known` but the entire field-phi is already marked as `OperandsKnown::all_known`. This is done with a call to the new `GraphBuilder::insert_load_at_end` method (see Section 5.4.4). In all other cases, the reevaluation result is used to set the field-phi to either `OperandsKnown::all_known`, `OperandsKnown::partly_known`, or `OperandsKnown::no_operand_known` accordingly.

5.4.3 Value Type Instance Allocation

An allocation of a value type instance is required each time it escapes the current scope due to an uninlined method call or a return statement returning the value type inside an uninlined method as described in Section 4.3.2. For that purpose, hooks are inserted in the `GraphBuilder::vreturn` and in the `GraphBuilder::invoke` method. In the latter case, a value type allocation is triggered when the arguments are set up. In both cases, the new method `GraphBuilder::try_append_value_type_instance` is called to properly allocate the value type as described in Section 4.3.4 and 4.3.5.

5.4.3.1 Allocate NewValueTypeInstance

The simpler case is to allocate a `NewValueTypeInstance` inside the `GraphBuilder::try_append_value_type_instance` method. The instance needs to be linked into the current block. However, it cannot be directly appended. The `NewValueTypeInstance` node could be active in another branch which has not allocated it yet. Moreover, the `NewValueTypeInstance` was created in an earlier block and therefore contains a different state

[44] An explanation of the method follows in Section 5.5.

(i.e. different values in the `ValueStack` instance). Thus, updating a state for any HIR instruction is prohibited by the C1 compiler once it is set. Therefore, the straightforward solution is to create a completely new `NewValueTypeInstance` with the current state in the block and its `_allocated` flag set to `true`. The buffered values of the old value type instance in the `MemoryBuffer` are copied with the `MemoryBuffer::replace_value_type_instance` method to a new entry for the newly created instance. The `NewValueTypeInstance` is appended to the graph with the `GraphBuilder::append_split` method (calls internally `GraphBuilder::append_with_bci`) and replaces the old value type in the local variables array.

In the next step, the fields need to be set to the correct values. The field values are read from the `MemoryBuffer` and are directly inserted into `StoreField` operations and appended to the HIR if they do not require a separate field value load. Reading the `MemoryBuffer::LOAD_REQUIRED` constant or a field-phi that has its `_operands_known` flag not set to `OperandsKnown::all_known` requires additional `LoadField` insertion(s). In the first case, a `LoadField` is either newly created or taken from the `BlockBuffer` (see Section 5.5). The latter case requires a recursive insertion of `LoadField` instructions such that the field-phi does not contain any `MemoryBuffer::LOAD_REQUIRED` constant as operands directly or recursively anymore (i.e. properly loaded). This is done with the new `GraphBuilder::try_append_load` method, which is described in Section 5.4.4.

5.4.3.2 Allocate Phi Instance

The situation is more complex if a value type phi instance escapes the scope and needs to be allocated inside the new `GraphBuilder::try_append_value_type_instance` method. The phi node could have its flag set to either `Allocation::allocated`, `Allocation::not_allocated`, or `Allocation::partly_allocated`:

- `Allocation::allocated`: The `Phi` instance can directly escape the scope since it is already properly allocated. Thus, the method returns immediately.
- `Allocation::not_allocated`: The phi function can directly be replaced by a `NewValueTypeInstance` as described in Section 4.3.5. In this case, it uses the same approach as described in Section 5.4.3.1.
- `Allocation::partly_allocated`: This case needs some additional effort to correctly allocate the phi node with allocation command pushes to predecessor blocks. The new method `GraphBuilder::insert_value_type_instance_at_end` is called recursively on all operands. This process can also be skipped by inserting an allocation directly in a block after the loop by the optimization described in Section 4.3.5.1.

If the `Phi` node is marked as being `Allocation::partly_allocated`, then the `GraphBuilder::try_append_value_type_instance` method first checks if the phi belongs to a loop-header block. If that is the case, a check is performed to see if the phi node refers to itself recursively. If that is not the case (`has_cycle` variable is `false`), then there was a reassignment of the

value type instance variable inside the loop-body. Therefore, an allocation is directly inserted after the loop where it is needed as described in Section 4.3.5.1 and the case is mapped to the same processing as if the phi is marked as `Allocation::not_allocated`, which is described later. Otherwise, the `GraphBuilder::insert_value_type_instance_at_end` is called recursively for each operand to allocate the instance properly. The `GraphBuilder::insert_value_type_instance_at_end` method also performs the same loop-header cycle check if the operand to process is marked `Allocated::partly_allocated` and moves the allocation into the corresponding block after the loop if it meets the same conditions as described before.

If the phi function is marked as `Allocation::allocated`, there is nothing to do and `GraphBuilder::try_append_value_type_instance` returns immediately as does the `GraphBuilder::insert_value_type_instance_at_end` method for such an allocated operand. The last possible case is that the phi function is completely unallocated (i.e. marked as `Allocation::not_allocated`). The phi node is directly replaced through an allocated `NewValueTypeInstance` according to rule A3 in Section 4.3.5 and is explained in the next paragraph.

A new `NewValueTypeInstance` has to be created with a copy of the `MemoryBuffer`-entry of the old instance due to the same reasons explained in Section 5.4.3.1. Afterwards, the instance needs to be inserted at the end of the block. The main reason for differentiating between the `GraphBuilder::try_append_value_type_instance` and the separate `GraphBuilder::insert_value_type_instance_at_end` method, even though they intend to do the same thing, lies within the way the C1 compiler is designed. The existing C1 implementation does not intend to allow an insertion of HIR nodes during the parsing stage anywhere else than at the end of the currently parsed block. Thus, the provided `GraphBuilder::append_with_bci` method (see Section 5.3), which automatically sets everything up (including the state), cannot be reused, unfortunately. Therefore, the `GraphBuilder::insert_value_type_instance_at_end` method needs to adapt the behavior of the `GraphBuilder::append_with_bci` method manually to insert the instance at the end of a specific block that was already parsed. On the other hand, the `GraphBuilder::try_append_value_type_instance` method can directly use the `GraphBuilder::append_with_bci` method to append to the right place in the currently parsed block without an additional effort.

Since a block is only a singly linked list (without references to previous nodes), the `GraphBuilder::insert_value_type_instance_at_end` method needs to traverse the block from the beginning to the end to find the second last instruction. The new value type instance needs to be inserted just after that instruction because the last instruction is always a `BlockEnd` instance. In the last step of the method, the fields have to be initialized to the correct values. This is done with `StoreField` instances which get the field values either from the `MemoryBuffer` or from possible `LoadField` instances in a similar way as described in Section 5.4.3.1. The only difference is that they need to be appended again manually without the help of the `GraphBuilder::append_with_bci`

method (as it is done in the `GraphBuilder::try_append_value_type_instance` method). Additionally, the created value type instance needs to replace the old value type instance in the local variable array of the state of the last `BlockEnd` instruction in the block. Otherwise, the wrong instance is fetched later as operand for a phi function. Once all fields are properly set with appended `StoreField` instructions (together with possible `LoadField` instructions), the last newly added `Instruction` sets its link to the last `BlockEnd` instruction in the block and the `GraphBuilder::insert_value_type_instance_at_end` method returns.

5.4.4 Field-Phi Load Insertions

The `MemoryBuffer` can return the `MemoryBuffer::LOAD_REQUIRED` constant directly or a field-phi function that contains it recursively in one of its operands (i.e. the field-phi is marked `OperandsKnown::partly_known` or `OperandsKnown::no_operand_known` as described in Section 4.3.6) upon a field value request from a value type instance. Those values can be further used but are required to be replaced in certain situations:

- A `vgetfield` bytecode needs to replace a `MemoryBuffer::LOAD_REQUIRED` value directly and, in case of a field-phi, all `MemoryBuffer::LOAD_REQUIRED` values recursively.
- A value type instance allocation needs to properly assign its fields and thus needs to replace all `MemoryBuffer::LOAD_REQUIRED` values directly and, in case of a field-phi, recursively to ensure a correct initialization of all fields.

The first requirement is triggered in the `GraphBuilder::vgetfield` method which processes the `vgetfield` bytecode. The second requirement is triggered in the `GraphBuilder::try_append_value_type_instance` and the `GraphBuilder::insert_value_type_instance_at_end` method. In both cases, a field-phi is delegated to the `GraphBuilder::try_append_load` method which loads the field value properly by processing it according to the state of its `_operands_known` flag:

- `OperandsKnown::all_known`: The field-phi can directly be used and the method returns immediately.
- `OperandsKnown::no_operand_known`: The field-phi is replaced by a direct `LoadField` instance in the corresponding block if all operands depend on the same instance.
- `OperandsKnown::partly_known`: This case needs some additional effort to correctly insert a `LoadField` in predecessor blocks. The method `GraphBuilder::insert_load_at_end` is called recursively on all operands. This process can also be skipped by inserting a field load directly in a block after the loop by the optimization described in Section 4.3.6.1 if all operands depend on the same instance and do not contain a statically known value directly or recursively.

The structure of `GraphBuilder::try_append_load` and `GraphBuilder::insert_load_at_end` is very similar to that used for `GraphBuilder::try_append_value_type_instance` and `GraphBuilder::insert_value_type_instance_at_end`, respectively (hence the same naming scheme). But the field value load methods have to do an additional check if all operands of a field-phi depend on the same value type instance to load from. However, getting the right value type instance on which a field depends on (i.e. the field load has to be performed on) and comparing it with all other instances found during the process is not trivial. A value type instance might be nested inside many field-phis and possibly contain cycles (field-phi operands referring to the field-phi itself). Therefore, a special helper method `GraphBuilder::get_load_instance` is implemented to achieve this task. It visits all operands recursively and tries to find the correct value type instance on which a field load would be performed. When an operand represents a `NewValueTypeInstance`, then its `_depends_on` field is used to get to the actual instance in the heap to load from (which could be again a `Phi` instance that needs to be processed recursively). To prevent getting stuck in an infinite loop in this process due to field-phi operands referencing the field-phi node itself recursively, an additional new class `VisitedArray`[45] is used to keep track of the already visited field-phi instances. The `GraphBuilder::get_load_instance` method returns the unique instance on which a value type depends on or `NULL` if the operands depend on different instances. If there is an instance mismatch, then the case for `OperandsKnown::no_operand_known` is mapped to the recursive field load insertion process for field-phis marked as `OperandsKnown::partly_known`. Moreover, the loop-optimization from Section 4.3.6.1 to insert a field load after the loop cannot be applied for field-phis marked as `OperandsKnown::partly_known` in case of an instance mismatch or if they contain a statically known value directly or recursively[46]. Otherwise, the direct field load insertion of Section 4.3.6 (and the loop optimization of Section 4.3.6.1) can be applied as shown in the next paragraph.

The following process for a field-phi marked as `OperandsKnown::no_operand_known` is only applied if all its operands depend on the same value type instance (checked with the `GraphBuilder::get_load_instance` method). Otherwise, the field-phi is treated and processed as being marked as `OperandsKnown::partly_known` which is described in the following paragraph. The `GraphBuilder::try_append_load` method distinguishes two cases for a field-phi marked as `OperandsKnown::no_operand_known`: the method is either (i) called directly in the currently active parsing block or (ii) called from the `GraphBuilder::insert_value_type_instance_at_end` method. In case (i), a newly created or buffered `FieldLoad` instance from the `BlockBuffer` (see Section 5.5) can directly be appended with the `append_with_bci` method. The correct instance to load from can be retrieved by the `GraphBuilder::get_load_instance` method. In case (ii), the currently active parsing block does not match the block in which the `GraphBuilder::insert_value_type_instance_at_end` inserts the new instance. Therefore, a field load has to be added manually without the help of the `append_with_bci` method for the same reasons explained in Section 5.4.3.2. The `BlockBuffer` can provide an already created

[45] Defined in `/share/vm/c1/c1_GraphBuilder.hpp`

[46] This is also checked with the `GraphBuilder::get_load_instance` method which queries the `MemoryBuffer` for statically known field values and returns `NULL` if such a value is found.

and buffered FieldLoad instance for the value type instance that is again found with the help of the GraphBuilder::get_load_instance method. The FielLoad instruction is then inserted before the last BlockEnd instruction in the block in the same manner as the allocation insertion in the GraphBuilder::insert_value_type_instance_at_end method. Before returning, the BlockBuffer is updated with the new LoadField instance.

The process for a field-phi that is marked as OperandsKnown::partly_known or as OperandsKnown::no_operands_known but was delegated to this process due to an instance mismatch calls the GraphBuilder::insert_load_at_end method for each operand which differentiates between three cases: (i) the operand is neither a phi nor the MemoryBuffer::LOAD_REQUIRED constant, (ii) the operand is a MemoryBuffer::LOAD_REQUIRED constant, or (iii) the operand is itself a field-phi function. In case (i), nothing has to be done and the method returns immediately. In case (ii), a LoadField instruction is inserted manually at the end of the corresponding block in the same way as described in the previous paragraph. In case (iii), the operand itself is a field-phi function and thus calls the GraphBuilder::insert_load_at_end method again for each of its operands recursively if they are marked as OperandsKnown::partly_known or as OperandsKnown::no_operand_known in case of an instance type mismatch to load from. Before the call, the field-phi is marked as OperandsKnown::all_known. This prevents an infinite loop in the process of a field-phi operand in a loop-header block that possibly refers itself recursively. If all operands of a loop-header field-phi depend on the same value type instance and do not contain a statically known value directly or recursively, then the loop-optimization from Section 4.3.6.1 can be applied by directly inserting the field load after the loop as in case (ii) for field-phis marked as OperandsKnown::no_operand_known.

It can be seen that the approach for field load insertions is very similar to the one for recursive allocations presented in Section 5.4.3.2. Due to all removals of already created HIR Instruction instances, some states need to be cleaned up accordingly later (see Section 5.6.1).

5.5 General Field Load Removal

This section describes the approach of buffering FieldLoad instances for value type field loads from the heap as discussed in Section 4.4. For that purpose, the MemoryBuffer class contains an additional GrowableArray _blockbuffers of BlockBuffer instances. The BlockBuffer[47] class has the same structure as the method-global MemoryBuffer in association with the FieldBuffer class. It provides a GrowableArray _value_types for all value type instances that have performed a field load earlier. The index of such a value type instance is used as an index in the actual GrowableArray _value_type_field_loads that stores the FieldLoad instances for a value type. Each time a BlockBuffer is requested for a field load, the method BlockBuffer::value_type_field_load_at_create is called. It checks if the requested field of a value type instance has already a buffered FieldLoad. If that is the case, it is returned.

[47] Defined in /share/vm/c1/c1_GraphBuilder.hpp

Otherwise, a new `FieldLoad` instance is automatically created, stored in the `BlockBuffer`, and returned to be used at the call site. An additional `BlockBuffer::check_if_exists` method is provided which checks if a field load is already buffered. It is used in the two phi setup methods `GraphBuilder::setup_value_type_phis` and `GraphBuilder::setup_remaining_operands` while evaluating the `_operands_known` flag of a field-phi to apply the optimization of Section 4.4.1.

The `BlockBuffer` entries are copied and merged from the predecessor blocks in the new `GraphBuilder::setup_value_type_loads` method just before the call of the `GraphBuilder::setup_value_type_phis` method inside the `GraphBuilder::connect_to_end` method. There are two types of methods to copy the content of a `BlockBuffer` depending on the number of predecessor blocks as explained in Section 4.4:

- **One Predecessor:** In this case, the `BlockBuffer::copy_loads` method is called which copies all buffer entries from the only predecessor block to the current block.
- **Multiple Predecessors:** In this case, the conjunction of all entries for a field are taken of each predecessor block by calling the `BlockBuffer::set_conjunction_for_value_type_loads` method. If a `FieldLoad` instance for a field of a value type instance is present in each block (i.e. available on all paths), the entry is copied. This ensures that no dead or unavailable field loads on a path are used in successor blocks. At the end of the last block in a loop-body, the `GraphBuilder::setup_value_type_loads` method is called again (right after the call of `GraphBuilder::setup_remaining_operands`) which invokes the `BlockBuffer::set_conjunction_for_value_type_loads` method one more time and overrides the previous conjunction result since the loop-body has to be considered, too.

A new `BlockBuffer` is created before one of the copy methods is called (if it does not already exist, which is the case for revisiting a loop-header block). The `MemoryBuffer` can be accessed through the `MemoryBuffer::get_block_buffer` method throughout the entire parsing process of all blocks.

5.6 Cleanup the HIR

This section describes the cleanup steps performed once all blocks are parsed and a first version of the HIR is completely built on return of the `GraphBuilder::iterate_all_blocks` method. The process of creating the LIR later has some conditions that need to be met, for example, that each existing value in the local variable array in a state (i.e. a `ValueStack` instance) also needs to have a corresponding origin in the HIR. With the new value type implementation and its optimizations, there are possibly many dangling unused HIR nodes due to the allocation and field load removal approach shown in Section 5.4 which have to be removed in each state. Furthermore, there are situations where a default value is requested to be stored with a `StoreField` instruction for a `NewValueTypeInstance` instruction in the HIR. This is redundant since the `NewValueTypeInstance` instruction already sets its fields automatically to their default values. The following paragraphs explain the necessary steps to fix these problems.

5.6.1 States

During the process of the allocation and field load removal, many `NewValueTypeInstance` are created that are not going to be allocated or were replaced by other `NewValueTypeInstance` nodes by adding them lazily as seen in Section 5.4.3. After all blocks are parsed, the `ValueStack` instances, as part of each instruction of the created HIR describing their state, need to remove these dead references from the local variables and stack arrays. For that purpose, a new class `VTStateInvalidator`[48] which is called from the new `GraphBuilder::eliminate_dead_value_types` method, is created. The `VTStateInvalidator` iterates over all blocks and processes the associated states of each instruction. The method `VTStateInvalidator::invalidated_dead_value_type_local` removes all unallocated value type instances, while the `VTStateInvalidator::invalidated_dead_value_type_stack` does the same for all stack values. The methods check if a local variable or stack value, respectively, is either a `NewValueTypeInstance` or a `Phi` instruction and then consider its `_allocated` flag. If it is not set to `true` for a `NewValueTypeInstance` or to `Allocation::allocated` for a `Phi` instance, the instruction is removed from the state.

The LIR generation process operates on all created `Phi` instances later. However, due to the lazy insertion of field loads (see Section 5.4.4), some field-phi functions might still contain a `MemoryBuffer::LOAD_REQUIRED` constant if the field-phi is never used throughout the method. Those field-phis are dead and thus should be removed for the LIR generation. This is especially important since the `MemoryBuffer::LOAD_REQUIRED` is just set to a constant which does not even represent a valid `LoadField` instance and might be accessed in the LIR generation process[49]. This removal is implemented in a new separate `VTFieldPhiRemover`[50] class, which is called from the `GraphBuilder::eliminate_redundant_phis` method. The `VTFieldPhiRemover` iterates over each field-phi function for each value type instance in each block. If an operand contains the `MemoryBuffer::LOAD_REQUIRED` constant, the field-phi is invalidated inside the `GrowableArray _value_type_field_phis` of the `ValueStack` state of the block[51]. The implementation needs to be careful for field-phi functions that refer itself recursively over its operands inside a loop-header block. Therefore, the already introduced `VisitedArray` class (see Section 5.4.4) is used to avoid an infinite loop.

[48] Defined in `/share/vm/c1/c1_GraphBuilder.cpp`

[49] As described earlier, the `MemoryBuffer::LOAD_REQUIRED` constant could be later improved to point to a dummy `LoadField` instruction that is designed specifically for this task to avoid runtime errors if it is accidentally accessed directly (see Section 7.1).

[50] Defined in `/share/vm/c1/c1_GraphBuilder.cpp`

[51] All created field-phis for a block are stored in the state of the `BlockBegin` instance at the start of the block, which is also used as reference to a block.

5.6.2 Default Value Stores

The parsing process of all blocks inside the `GraphBuilder::iterate_all_blocks` method can possibly create many redundant phi functions of the form $x = [y, y]$. This happens, for example, when a phi function is created from branches that all contain the same value. The call to the `GraphBuilder::eliminate_redundant_phis` replaces such phi functions by direct HIR nodes with the help of the already existing `PhiSimplifier`[52] class. For example, a phi node $\varphi =$ [`Constant(1)`, `Constant(1)`] is directly replaced by the instruction `Constant(1)`. This simplification works for all kinds of HIR `Instruction` instances. However, this process can also produce `StoreField` instances that would store a default value to a value type instance. Such field stores are redundant and thus are removed with the help of the new `VTStoreRemover`[53] class. All instructions in each block are iterated and each `StoreField` instruction that would assign a default value to a `NewValueTypeInstance` is removed by setting the link of the previous instruction the to one the `StoreField` instance originally pointed to.

5.6.3 Dead Field-Phis

After removing all dead field-phis containing a `MemoryBuffer::LOAD_REQUIRED` constant with the `VTFieldPhiRemover` class and simplifying phis (including field-phis) with the `PhiSimplifier` class, a *live* check is performed for all field-phis. This is done by the new `VTFieldPhiLiveChecker`[54] class which is also called from the `GraphBuilder::eliminate_redundant_phis` method. Each `Phi` instance that represents a field-phi has a special flag `_field_phi_used` set to `false` at its creation. The `VTFieldPhiLiveChecker` loops through all instructions of all blocks and checks all associated values that are part of an instruction together with its state values if a field-phi `Phi` instance is used (i.e. occurs in the HIR). If this is the case, its `_field_phi_used` flag is set to `true` which indicates that the field-phi is not dead (i.e. live). This flag is later checked in the `LIRGenerator::move_to_phi` method. If a field-phi is dead, the `LIRGenerator::move_to_value_type_field_phi` method is not called and no instructions are moved to the predecessor blocks. In this case, the field-phi is invalidated for the associated state. This is also important for various checks later in the linear scanner.

5.7 On-Stack-Replacement

The OSR-entry block has a special flag `BlockBegin::osr_entry_flag` set. During the parsing stage inside the `GraphBuilder::iterate_all_block` method, this condition is checked and the block is setup with the `GraphBuilder::setup_osr_entry_block` method. The only adaptation that has to be made for the value type approach presented in this thesis is to separately initialize the `BlockBuffer` for the OSR-entry block inside this method. The phi setup is automatically done later with the `GraphBuilder::setup_value_type_phi` method. The interpreter passes all local variables and stack values directly to the OSR-entry block. The value type in-

[52] Both defined in `/share/vm/c1/c1_GraphBuilder.cpp`
[53] Defined in `/share/vm/c1/c1_GraphBuilder.cpp`
[54] Defined in `/share/vm/c1/c1_GraphBuilder.cpp`

stance phis created in the successor block are always marked as `Allocation::partly_allocated` or `Allocation::partly_allocated` since the interpreter only works with allocated value types. Field loads are pushed into the OSR-entry block in the same manner as for normal blocks (see Section 5.4.4).

5.8 Deoptimization

The section presents a first exploration of the possibilities and challenges to support deoptimization (see Section 2.5.3) in combination with value types. The implementation is by no means complete but provides a basis that can be used as a starting point for a further development in the future.

A deoptimization is most often applied in combination with the C2 compiler (see Section 2.5.3.2). There are many optimistic optimization that can later turn out to be wrong. This requires a transfer back to the interpreter. However, there are still a few cases where the C1 compiler can trigger a deoptimization as well. All possible reasons are described in the `enum DeoptReason` as part of the `Deoptimization`[55] class. A deoptimization can happen at any time and needs to be handled instantly. Each instruction created for the LIR from the HIR gets a `CodeEmitInfo`[56] instance that contains information about the HIR, such as a `ValueStack` instance describing the state. This information can later be used for a deoptimization to have access to the current values on the stack and for the local variables. The problem with the allocation and field load removal process (see Section 5.4) is that some value type instances are not allocated or not even present anymore but are required to exist and being allocated. Otherwise, the interpreter could possibly run into a *null pointer exception* upon an access. Therefore, an allocation process needs to be triggered for each unallocated `NewValueTypeInstance` or not fully allocated value type `Phi` instance (i.e. not set to `Allocation::allocated`) when a deoptimization is required.

For this case, an additional `GrowableArray _unallocated_value_types` is added to the `ValueStack` class. Each time a new value type instance is stored in a local variable, for example in `GraphBuilder::vstore` or in `GraphBuilder::setup_value_type_phis`, a new entry gets created at the index of the local variable if the instance is not marked as being completely allocated (i.e. an unallocated `NewValueTypeInstance` or a value type `Phi` instance that is not marked as `Allocation::allocated`). The interpreter can only work correctly with fully allocated value types (i.e. allocated on each possible path). Once the LIR is created, the linear scanner (`LinearScan` class) iterates over the LIR and creates a special `IRScopeDebugInfo`[57] instance. Its purpose is to record all debug information for a particular `CodeEmitInfo` instance attached to an instruction. The `IRScopeDebugInfo` instance translates the information inside the `ValueStack` state instance to a different format. All `GrowableArray` of `Instruction` instances are converted to `GrowableArray` of `ScopeValue`[58] instances. The value type instances use the

[55] Defined in `/share/vm/runtime/deoptimization.hpp`
[56] Defined in `/share/vm/c1_IR.hpp`
[57] Defined in `/share/vm/c1_IR.hpp`
[58] Defined in `/share/vm/code/debugInfo.hpp`

`ObjectValue` class and the value type field values one of the appropriate `Constant` subclasses of `ScopeValue`[59]. This new format allows a space-efficient serialization to a binary format later when assembly code is emitted. This is done over the `CodeEmitInfo::record_debug_info` method which calls the `IRScopeDebugInfo::record_debug_info` method that serializes the `GrowableArray` of `ScopeValue` instance to an opaque data structure called `DebugToken` (see `/share/vm/code/debugInfoRec.hpp`).

Finally, the actual execution process of a deoptimization request calls the `Deoptimization::fetch_unroll_info_helper` method. Each compiled frame is represented by a `compiledVFrame`[60] instance and contains a corresponding scope descriptor instance (`ScopeDesc`[61] class). This descriptor encodes the debug information stream from above back to `GrowableArrays` of `ScopeValue` instances. The `Deoptimization::fetch_unroll_info_helper` method eventually calls the `Deoptimization::realloc_objects` and afterwards the `Deoptimization::reassign_fields` method to allocate the value type instance and assign its fields to the format used in the interpreter. The big challenge faced is that the C1 compiler was originally not designed to reallocate objects on a deoptimization. This concept is only implemented for the C2 compiler. Therefore, it is difficult to find the right spots to adapt the code for enabling this functionality for the C1 Compiler. Another non-trivial challenge is to set the field values of a value type instance up correctly. Some fields need to be loaded, where others are already present in buffer entries in the `MemoryBuffer`. Field-phi functions need an additional effort to ensure that each operand is correctly loaded. An idea could be to reuse the `GraphBuilder::try_append_load` in association with the `GraphBuilder::get_load_instance` method to preform the required field loads. However, this assignment process is not yet completed and is left for future work since it would have exceeded the scope of this thesis (see Section 7.1).

[59] All defined as subclasses of `ScopeValue` in `/share/vm/code/debugInfo.hpp`
[60] Defined in `/share/vm/runtime/vframe_hp.hpp`
[61] Defined in `/share/vm/code/scopeDesc.hpp`

6 Evaluation

This chapter shows an evaluation of the presented value type implementation in the C1 compiler. The optimized value types with the allocation removal and field load insertion optimization are compared to the unoptimized value type implementation and an implementation with conventional objects only representing the baseline. Unfortunately, there are no publicly available benchmark suites[1] for value types in Java since they are not part of standard Java SE. Moreover, the implementation is restricted to non-array usages. Therefore, the performance is tested with own benchmarks including typical use cases of value types.

Section 6.1 first briefly explains how correctness tests have been performed during the development of value types in the C1 compiler. It follows a description of the experimental setup for the evaluation together with the used flags and details about the machine in Section 6.2. Before the evaluation was done, some preliminary changes had to be done as described in Section 6.3. The first two benchmarks in Section 6.4 and Section 6.5 evaluate the cost of a simple allocation (with the `vdefault` bytecode) and an allocation with the `vwithfield` bytecode, respectively. The evaluation of the allocation and field load removal approach is presented in Section 6.6. Finally, Section 6.7 evaluates some well-known mathematical applications with value types and illustrates the importance of inlining decisions done by the compiler.

The code of each benchmark method for an evaluation in the following sections is defined in the benchmark class `C1ValueTypeBench`[2]. A mapping from an evaluation to the benchmark methods is shown in Table A.1 in Appendix A.4.

6.1 Correctness Tests

During the development of the value types for the C1 compiler in this thesis, various correctness tests have been performed including some edge cases. They are contained inside the `OwnTests`[3] class. There are 88 tests on almost 2000 lines of code[4]. They have been evaluated with the *c1visualizer* [64] tool. The debug build of the HotSpot™ JVM[5] can be run with a special `-XX:PrintCFGToFile`

[1] State: October 2017

[2] Defined in `/hotspot/test/compiler/valhalla/valuetypes/C1ValueTypeBench.java` in the provided patch

[3] Defined in `/hotspot/test/compiler/valhalla/valuetypes/OwnTests.java` in the provided patch

[4] The 108 tests written in the benchmarks for the evaluation in the following sections could also be counted as correctness tests which results in 196 tests in total for value types in the C1 compiler.

[5] Special build that provides more debug information together with assertions that are not present in the normal product build.

flag which prints the created HIR and LIR to a file. They can afterwards be inspected with the c1visualizer tool. This has been done for all the tests to check the expected results of the optimizations such as removed allocations or removed field loads. However, they are not very meaningful to use as performance tests. Therefore, some typical use cases of value types are evaluated with new benchmarks and are described in the following sections.

6.2 Experimental Setup

Value types have been implemented and tested throughout the thesis on a Linux x86-64 architecture only. Nevertheless, the code could be ported to different architectures in the future without a big effort since most parts, including the optimizations, are done within the platform-independent HIR (see Section 7.1). The entire evaluation of this thesis is performed with the following machine setup:

- **Architecture:** 64-bit x86-64 Linux architecture with 16 GB RAM
- **Processor:** Intel® Core™ i7-4790K CPU @ 4.00GHz with 4 physical and 8 virtual cores
- **Operating System:** Ubuntu 16.04.3 LTS (xenial)
- **JVM Build:** Product build[6] of the provided patch to the changeset 13528:f017570123b2 of project Valhalla from 21. August 2017 and built with GCC version 4.7.4[7]

The evaluation is done with the *JMH* benchmark tool [59] which is specifically designed by the OpenJDK team for testing the HotSpot™ JVM. A `jar` file of the benchmark class is built with the *Maven* build tool [69] which calls the experimental version of `javac` of the project Valhalla to generate value type bytecodes. JMH treats specifically annotated methods (with a `@Benchmark` annotation) as benchmark methods and executes them iteratively. It automatically generates the average time per method execution together with the standard deviation as an error metric. The HotSpot™ JVM is run with the following fixed flags:

- **-Xbatch:** Stops the thread, which caused the method to be complied, until the compilation is complete (i.e. avoids background compilation).
- **-noverify:** Disables bytecode verification since the value type bytecodes have no verifier support yet[8].
- **(*) -XX:-ValueTypePassFieldsAsArgs:** Disables the optimization to only pass value type fields to a method to bypass an allocation (see Section 7.1).
- **(*) -XX:-ValueTypeReturnedAsFields:** Disables the optimization to only return value type fields from a method to bypass an allocation (see Section 7.1).
- **-XX:+EnableValhalla:** Enables the use of Valhalla specific features, such as value types.

[6] Normal Java build without additional debug information support as in the debug build (which is also slower as the product build).

[7] Patch link: `http://cr.openjdk.java.net/~thartmann/thesis_hagedorn/webrev/hotspot.patch`
Changeset link: `http://hg.openjdk.java.net/valhalla/valhalla10-old/hotspot/rev/f017570123b2`

[8] State: October 2017

- **-XX:TieredStopAtLevel=1:** Restricts the JVM to only go from interpretation (level 0) to C1 compilation at level 1 without gathering profiling information (see Section 2.5.4).
- **-XX:BigValueTypeThreshold=1:** This flag is used for a workaround due to a bug in the interpreter for value types in the working changeset 13528:f017570123b2. This, however, does not affect the evaluation presented in this chapter.

The two flags marked with (*) are enabled by default and were originally not available in the product build. This was changed for this thesis as these flags represent optimizations for value types that are not yet supported in the C1 compiler and thus need to be explicitly disabled. Additionally specified flags for benchmark tests are mentioned in the corresponding setup descriptions.

Each benchmark method (annotated with `@Benchmark`) is executed with the same measurement settings summarized in Table 6.1. A JMH *fork* creates a completely new instance of the JVM and lets the entire benchmark run with the same warm-up and measurement setting again. The JMH environment automatically generates the average time of executing a method once in nanoseconds over all measurement periods of all forks (excluding the warm-up phase). It additionally provides the standard deviation as an error metric, which is provided in each graph throughout this chapter.

Table 6.1: Benchmark setup for all measurements of all evaluations

Property	**Setting**
Measurement Unit	Average Time [ns] / Method Execution
Error Metric	Standard Deviation
Warm-Up	2x 10s
Measurement	5x 20s
Forks	3

6.3 Preliminary Changes

Before any benchmark test for this thesis could have been run, a few modifications to the existing code had to be done. The problem is that the current version of the HotSpot™ JVM aligns the very first field of a value type class to a long offset (i.e. eight bytes for a 64-bit system). This is an optimization for copying a value type which has to be done a lot in the interpreter. However, this impacts the results of an evaluation of the value types in the C1 compiler for this thesis. This field alignment problem is also described in the official bug report JDK-8182473[9] which suggests to fix the alignment to the largest field type of the involved class. However, its status is still open[10] and therefore a custom fix is done in the `ClassFileParser`[11] and the `ValueKlass`[12] class to avoid this problem temporarily for the evaluation of this thesis. The changed code parts are marked with a comment containing the keyword "C1 implementation" in these two classes.

[9] Link: `https://bugs.openjdk.java.net/browse/JDK-8182473`
[10] State: October 2017
[11] Defined in `/share/vm/classFile/classFileParser.cpp`
[12] Defined in `/share/vm/oops/valueKlass.cpp`

An additional adaptation has to be done for JMH itself. The benchmark methods are called from an internal JMH method which afterwards consumes the return value. The problem is that it can only handle objects as return values but not value type instances. Therefore, additional `useOT` and `useVT` methods are used instead of `return` statements which are explicitly marked for the compiler to not be inlined[13]. These should simulate an escape from the current method scope (not only for a `return` statement) to trigger a value type allocation for the optimized value types benchmarks.

6.4 Cost of an Allocation

The first simple benchmark evaluates the impact of a single allocation with the object baseline and the unoptimized value types compared to the case of removing it completely with the optimized value types version (i.e. simulating the `vdefault` bytecode). This is done with a varying number of fields (1, 2, 4, 8, and 16). The fields are automatically set to their default values as part of the underlying allocation call of the JVM. Nevertheless, a normal class needs an additional constructor to initialize its `final` fields. To get a better performance, the fields could be directly initialized by a statement `final int i1 = 0` to omit the constructor assignments. However, this would make the class itself useless as it could only create instances with its values set to their default values. Therefore, a constructor is used to get a usable class in practice.

A separate class is defined for a different number of fields for both value types and objects. They use the convention `VT_#` and `OT_#`, where `#` is replaced by the number of fields. For example, the two classes for eight fields are named `VT_8` and `OT_8` for value types and objects, respectively. To have comparable results, all fields are of type `int`. The benchmark calls a simple method that creates either a value type for a value type class or an object for an object type class. The method for value types uses the special `__MakeDefault` modifier, whereas the object version uses the normal `new` keyword. An example of the benchmark method for four fields is shown in Listing A.5 in Appendix A.5. A creation method is called from the actual JMH benchmark method which inlines the call due to its small size. The additional layer is necessary since value types can only be created inside a value type class.

Due to the inlining of the method, the allocation removal optimization for value types removes the value type and its allocation completely since it does not escape. This situation is directly compared to the method for objects which allocates the instance each time. An additional run is performed for unoptimized value types which are treated as objects. Since value types, compared to objects, do not call an additional constructor the performance should be slightly better. Moreover, `final` fields of a normal class need to be additionally set by a constructor which should also have an influence on the performance. For an increasing number of fields, an allocation needs more space and should need more time to initialize the fields properly.

[13] This is done in JMH with the annotation `@CompilerControl(CompilerControl.Mode.DONT_INLINE)` which is an equivalent to the earlier used flag `-XX:CompileCommand=dontinline,SomeClass::SomeMethod`.

The results are shown in Figure 6.1 together with the standard deviation. The x-axis shows the different number of fields (i.e. the different classes with the corresponding number of fields) and the y-axis how much time that is needed for executing a method once on average (this is unchanged for all the remaining evaluations in this chapter except for Section 6.7 and is not mentioned again). Since the allocation is completely removed for the optimized value type version, the method body is empty and therefore has the same low average time for any number of fields. The result for the unoptimized value types and the object baseline are similar for few fields. However, with a growing number of fields the influence of calling the constructor and initializing the fields again for objects results in a worse performance.

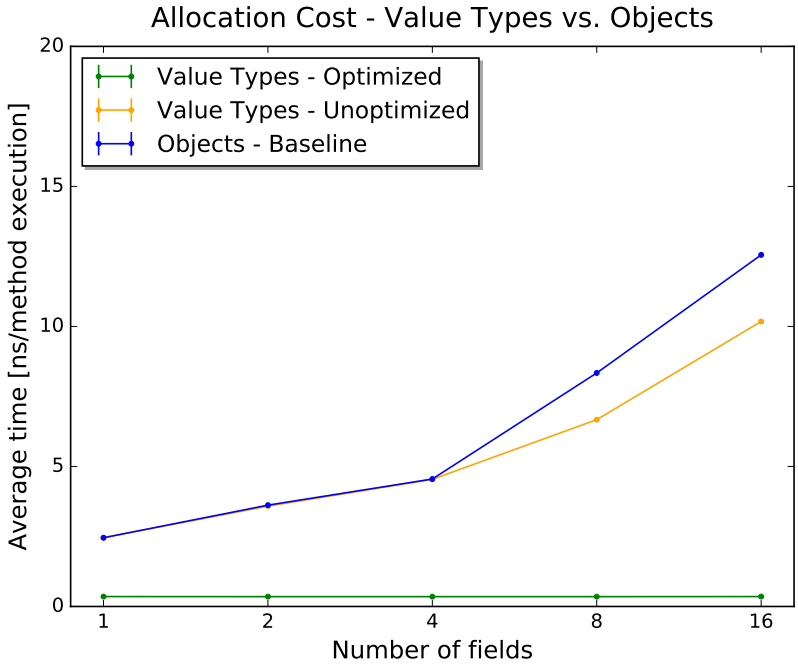


Figure 6.1: Results of the benchmarks of Section 6.4

These results show that the actual size of an allocation has a certain influence on the performance. Not only is an allocation of an object slower with more fields but also garbage collection is invoked sooner due to an increased heap consumption for larger objects which additionally impacts the performance. However, garbage collection does not influence this evaluation much since the amount of allocations is small. It becomes a more significant factor in the evaluations with many allocations inside loops in later sections. Therefore, a complete removal of a value type allocation for a growing number of fields becomes even more beneficial. For one field, the optimized value type version, which removes the allocation completely, is 6.9 and for 16 fields over 35.4 times faster than the object baseline, which only performs a single allocation to simulate the `vdefault` bytecode.

6.5 Cost of "vwithfield"

This benchmark is similar to the one used in Section 6.4. The only difference is that the method call does not perform a default creation (as for the `vdefault` bytecode) but instead executes the actions of a `vwithfield` bytecode. This means that a new instance is created by updating one field while copying the fields of the old instance. The benchmark method is implemented in a straightforward way for value types by reassigning a field in a value factory[14]. However, it needs an additional effort to compare them to the baseline with objects which should perform the same copying steps. This is simulated for objects in two ways: (i) simply calling a constructor with an argument for each field while setting one to the new value and the others to the original field values, or (ii) calling a constructor with the field to be set together with another instance to copy from. The first approach should be slower since an additional argument has to be pushed for each field upon calling the constructor compared to the fixed two-argument constructor. However, the second approach suffers from an inflexibility that it can only be called to update a single specific field. To update other fields, a separate constructor must be declared accordingly.

This benchmark sets the first field to a new value for a varying number of fields reusing the same classes as in Section 6.4. An example of such a benchmark method for four fields is shown in Listing A.4 in Appendix A.5 for value types and objects. The object class provides two setter methods for the two approaches (i) and (ii) explained in the previous paragraph. All involved methods in this benchmark are automatically inlined due to their small sizes and are executed for value types with and without optimizations and for both object setter methods with the two constructors (i) and (ii) resulting in four different measurement settings.

The results are shown in Figure 6.2. They are very similar to those from Section 6.4. The unoptimized value types perform again better than both object versions due to saving an additional constructor call. The object constructor (i) with an argument for each field clearly becomes worse with a growing number of fields. The arguments do not fit in a register anymore and have to be pushed onto the stack eventually. The optimized value type version completely removes the allocation and the field assignments of the value type and thus performs similarly well compared with the benchmarks in Section 6.4. It is between 4.8 (for one field) and 23.9 times (for 16 fields) faster than the object baseline with constructor (ii) and up to 30.9 times (for 16 fields) faster than the object baseline with constructor (i).

6.6 Allocation and Field Load Removal

This section evaluates some typical situations of value type usages. They are compared with and without optimizations to a similar baseline implementation with objects. The evaluation uses a class `VTPoint` and `OTPoint` for value types and objects, respectively, which represents a point with an `int` x- and y-component. As for the previous benchmarks, there are also additional classes

[14] The method is marked with the `__ValueFactory` modifier.

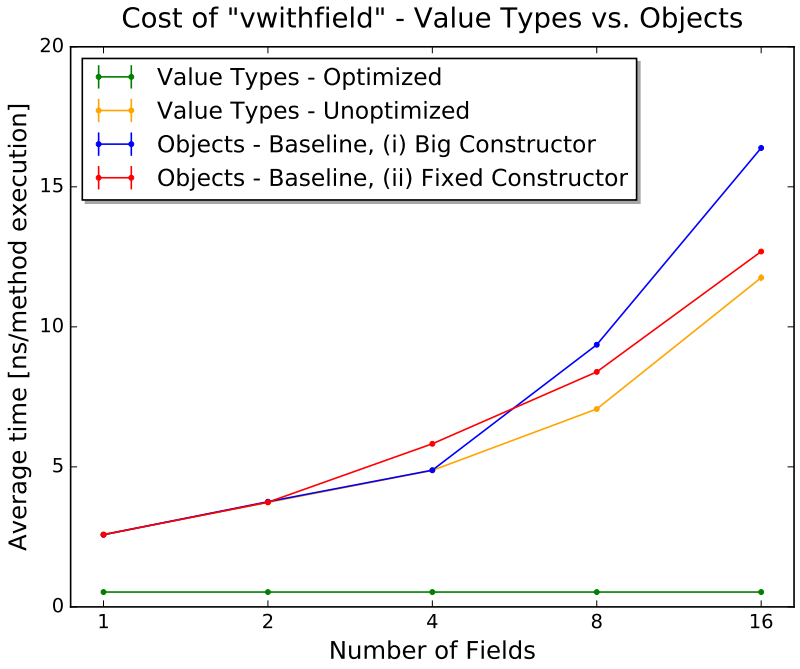


Figure 6.2: Results of the benchmarks of Section 6.5

containing more `int` fields to compare the difference in performance depending on the number of fields. Each benchmark in this section is executed for 2, 4, 8, and 16 fields. The class naming convention for 4, 8, and 16 fields is very similar: `VTPoint_#` and `OTPoint_#`, where `#` is replaced by the number of fields. Each class provides a creation method which initializes a point with two provided arguments for the x- and y-component, respectively. The other fields (if present) are set to their default values. Furthermore, each method contains a setter method for the x- and y-component which copies all values of one instance to a *newly created* instance and set the requested field to the new provided value. These methods are always inlined due to their small sizes. An example for the two classes with four fields is shown in Listing A.5 in Appendix A.5. All benchmarks method presented in this section use these methods without explaining them again.

6.6.1 Simple Branch

The first benchmark evaluates a simple `if-else` statement shown in pseudo-code in Listing 6.1 (a). A point $p = (3,4)$ is created and its x-component is set to the value of the y-component in the `if`-branch and vice versa in the `else`-branch. The condition for the branch is statically unknown. Otherwise, the compiler would figure out its value and remove the block for the branch which never gets executed. The benchmark is evaluated twice. Once letting p escape (with the red marked, uninlined call on line 13) and once without (without executing the red marked statement). The performance impact of this small method call should be negligibly small for objects and unoptimized value types. However, it should have a bigger influence for the optimized value type version since an allocation is inserted due to the escape.

The `Point` class and `use` method are placeholders for the actual `VTPoint` and `OTPoint` classes and uninlined `useVT` and `useOT` methods (see Section 6.3) for value types and objects, respectively. This is also the case for the evaluations shown in the proceeding sections and is not repeated again.

Listing 6.1: Benchmark method in pseudo-code (a) and the generated pseudo-HIR for optimized value types (b) for the benchmarks in Section 6.6.1

(a) Method in pseudo-code

```
1  // Block 1
2  Point p = Point.createSet(3,4);
3  int x = p.x;
4  int y = p.y;
5  if (/* unknown condition */) {
6    // Block 2
7    p = Point.setX(p,y); // x = y
8  } else {
9    // Block 3
10   p = Point.setY(p,x); // y = x
11 }
12 // Block 4
13 use(p) // Escape!
```

(b) Pseudo-HIR for optimized value types

```
1  // Block 1
2  x = Constant(3) // int x = p.x
3  y = Constant(4) // int y = p.y
4
5  /* ... */
6
7  // Block 4
8  φx = [y,x]
9  φy = [y,x]
10 p1 = Allocate Value Type Instance p
11 p1._x := φx // StoreField
12 p1._y := φy // StoreField
13 invokestatic(p1)
```

The results are shown in Figure 6.3 together with the standard deviation. Without an escape, the optimized value types only assign constants to both variables x and y in the HIR shown in pseudo-code in Listing 6.1 (b). Those are even removed later in the register allocator since they are not used (i.e. dead). Thus, the performance for optimized value types, shown as green graph in Figure 6.3, is equally good compared to the performance of the optimized value types in the benchmarks of Section 6.4 and 6.5. However, additionally executing the red marked `use` statement triggers an allocation. This is also shown in red in Listing 6.1 (b) in the last block. Block 2 always provides the value of y for x and y due to the setter method and block 3 the value of x for the same reason. These are then merged with the phi functions shown in Block 4. The value type instance is unallocated on both branches and thus an allocation with proper field assignments is directly inserted in block 4 after the `if-else` block according to the recursive allocation rule A3 from Section 4.3.5. The statements to execute are very similar to those for allocating and setting a field for the unoptimized value type version in Section 6.5. The resulting performance shown as orange graph in Figure 6.3 is therefore similar to the one for the unoptimized value types in Figure 6.2.

The performance for the unoptimized value type version and the object baseline are almost identical in both method versions with and without escape (<0.9ns) and thus showing both results would overlap themselves. Moreover, it would not add value to the purpose of the discussion. Therefore, the results without the red marked call in Listing 6.1 (a) are omitted in Figure 6.3 for

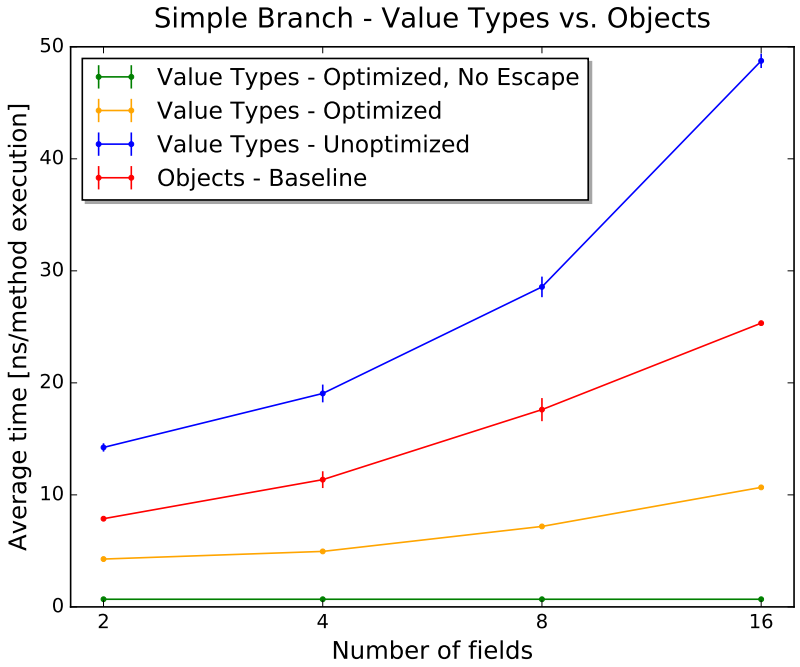


Figure 6.3: Results of the benchmarks of Section 6.6.1

the unoptimized value types and the object baseline. The object version needs an allocation at the start of the method to assign p and one in either branch for calling the setter. The situation is even worse for the unoptimized value types. Calling the initial setter requires an allocation of three value types: a default, one for setting x and one for setting y, respectively. This is a downside compared to objects which can directly initialize all their `final` fields at once at the creation of the instance, whereas value types first need a `vdefault` and then a `vwithfield` for each field[15].

In summary, the non-escaping and escaping optimized value type version need zero and one allocation, respectively, the object baseline two, and the unoptimized value type version four allocation (one with `vdefault` and three with `vwithfield`). This fact can also be seen in the results, for example for 16 fields in Figure 6.3. The difference of optimized value types with no escape to objects and from objects to unoptimized value types is almost equal. This corresponds to the same relative difference in terms of number of allocations. The same observation can be made for the relation between optimized value types with no escape and the version with an escape as well as between the optimized value type version with an escape and the object baseline. As before, garbage collection has only a small influence on the results with those few allocations.

6.6.2 Loops

The next benchmark executes a simple `for`-loop shown in pseudo-code in Listing 6.2. In each iteration p is reassigned with a copy of p (i.e. a new instance) that has its x-component updated. As in the evaluation in Section 6.6.1, the method is executed twice, once with the red marked

[15]This is no problem with the allocation and field load removal optimization since it can omit those additional allocations.

escape statement and once without. For the same reasons, only the results of the optimized value type versions are shown for both benchmark executions. The results for the unoptimized value type version and object baseline are only shown for the benchmark that executed the additional red marked escape statement.

The loop-body is executed 100 times and requires an allocation on each iteration for the unoptimized value type version and the object baseline[16]. However, the optimized value types can omit these allocations. This should have a significant performance benefit compared to the unoptimized value types and the object baseline version. The optimized value types either skip an allocation completely (in case of no escape) or allocate only once (in case of an escape) after the loop due to the recursive allocation rule A3 from Section 4.3.5 for phi nodes marked as *not-allocated*.

Listing 6.2: Benchmark method in pseudo-code (a) and the generated pseudo-HIR for optimized value types (b) for the benchmarks in Section 6.6.2

(a) Method in pseudo-code

```
1  // Block 1
2  Point p = Point.createSet(3,4);
3  int x = p.x;
4  int y = p.y;
5  for (int i = 0; i < 100; i++) {
6    // Block 2
7    p = Point.setX(p,x+i);
8  }
9  // Block 3
10 use(p) // Escape!
```

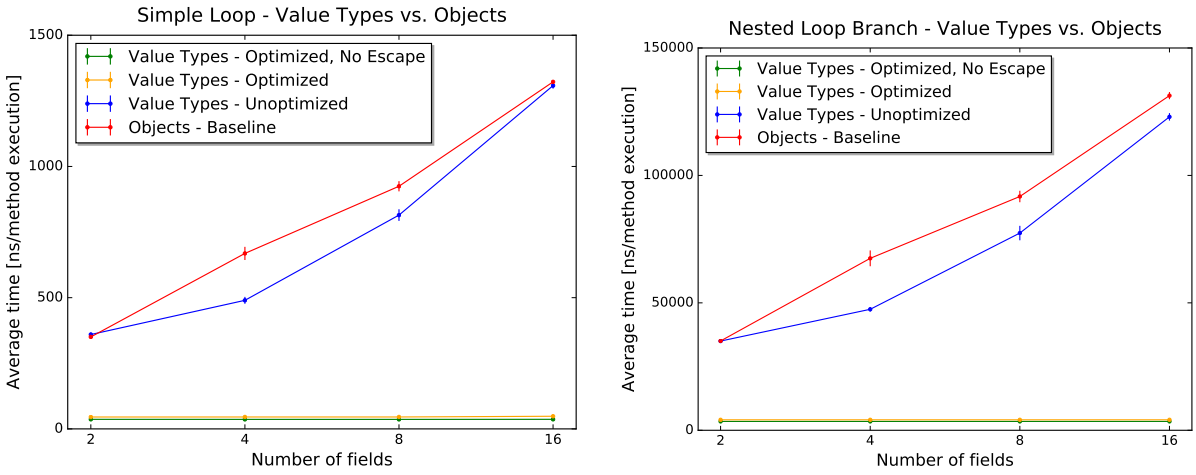
(b) Pseudo-HIR for optimized value types

```
1  // Block 1
2  x1 = Constant(3) // int x = p.x
3  y1 = Constant(4) // int y = p.y
4  i1 = Constant(0) // int i = 0
5
6    // Block 2
7    φi = [i1,i3]
8    φx = [x1,x2]
9    i2 = Constant(1)
10   i3 = ArithmeticOp(φi+i2) // i++
11   x2 = ArithmeticOp(x1+φi) // x+i
12
13 // Block 3
14 p1 = Allocate Value Type Instance p
15 p1._x := φx // StoreField
16 p1._y := y1 // StoreField
17 invokestatic(p1)
```

The results are shown in Figure 6.4 (a) together with the standard deviation. Without an escape, the optimized value types only do some simple arithmetic operations inside the loop-body as shown in pseudo-code in Listing 6.2 (b). These are repeated 100 times but are still very fast compared to additionally performing an allocation each time. This can be seen by the huge performance difference in the graph compared to the unoptimized value types and the object baseline version.

[16] As described earlier, the `setX` method for objects also creates and allocates a new instance due to the fields of `OTPoint` being `final`. Thus, it cannot just override them and skip the allocation in each iteration. This simulates the behavior of a value type.

The optimized value type version with an escape in the end requires only one allocation since both branches provide an unallocated value type instance for block 3 as discussed above and is shown in red in Listing 6.2 (b). The field y can directly be used as it is not modified[17]. For setting up the field-phi functions, some additional arithmetic operations have to be performed which, however, are relatively fast. Thus, both optimized value type versions perform equally good in comparison to the unoptimized value types and the object baseline version. Even when substituting the inlined `Point::createSet` method on line 2 in Listing 6.2 (a) by an allocated instance with unknown field values, it still requires only one allocation in block 3 due to the optimization to avoid allocation pushes to loop-bodies for partly-allocated value type instance phis (see Section 4.3.5.1). It would only need two additional field loads from the heap for $p.x$ and $p.y$ which does not take up a more than few nanoseconds. This is still equally fast compared to the performance gap to the unoptimized value types and the objects baseline version and thus is not additionally shown in Figure 6.4 (a).



(a) Simple Loop

(b) Nested Loop

Figure 6.4: Results of the benchmarks of Section 6.6.2

For two fields, the object baseline performs slightly better than the unoptimized value type version since the time spent for the call to the constructor is negligibly small compared to the additional allocations in the `Point::createSet` method required for the value types. However, this impact of additionally calling a constructor becomes noticeable for four fields but then converts towards the performance of the unoptimized value types for an increasing number of fields since the actual copying mechanism for all fields becomes the limiting factor. The performance difference is between -2.5% (for two fields) and 36.5% (for four fields).

These results show the real benefit of value types by omitting allocations inside loop-bodies. Moreover, garbage collection can be avoided which additionally impacts the performance. The gap towards using objects is enormous and pays off significantly for many loop iterations. This is further evaluated by replacing the loop shown in Listing 6.2 by a nested-while loop shown in

[17]This is done by the `PhiSimplifier` class which detects that the phi function for y contains the value y for both operands and therefore replaces it directly by y.

Listing 6.3, resulting in 100*100 iterations. The results are shown in Figure 6.4 (b). The relative performance gap is, as one would expect, the same as in Figure 6.4 (a) but the absolute gap is tremendous. Compared to the non-escaping version for optimized value types, the absolute difference to the escaping version is noticeable due to the additionally required operations for using the field-phi functions due to many more iterations. The escaping version is around 700ns slower.

Listing 6.3: Nested loop replacing the loop in Listing 6.2 (a)

```
for (int i = 0; i < 100; i++) {
  p = Point.setX(p, i+x);
    for (int j = 0; j < 100; j++) {
      p = Point.setY(p, j+y);
    }
  }
}
```

Listing 6.4: Benchmark method in pseudo-code (a) and the generated pseudo-HIR for optimized value types (b) for the benchmarks in Section 6.6.3

(a) Method in pseudo-code

```
// Block 1
Point p = Point.createSet(3,4);
int x = p.x;
int y = p.y;
for (int i = 0; i < 100; i++) {
  // Block 2
  p = Point.setX(p,x+i);
  use(p); // Escape!
}
```

(b) Pseudo-HIR for optimized value types

```
// Block 1
x1 = Constant(3) // int x = p.x
y1 = Constant(4) // int y = p.y
i1 = Constant(0) // int i = 0

  // Block 2
  φ_i = [i1,i3]
  φ_x = [x1,x2]
  φ_p = [p1,p2]

  i2 = Constant(1)
  i3 = ArithmeticOp(φ_i+i2) // i++
  x2 = ArithmeticOp(x1+φ_i) // x+i
  p2 = Allocate Value Type p
  p2._x := φ_x // StoreField
  p2._y := y1 // StoreField
  invokestatic(p2)

// Block 3
invokestatic(\varphi_p)
```

6.6.3 Loop-Body Allocations

This section evaluates the situation where a value type instance is both newly created and afterwards escaping inside the loop-body from the method. Therefore, it also needs an allocation for optimized value types in each loop iteration. The structure of the code is very similar to that used in Section 6.6.2 with the only change that only one `use` statement lets the point instance escape from the loop-body. The pseudo-code is shown in Listing 6.4 (a). The pseudo-HIR for optimized value types is presented in Listing (b) which shows the required allocation in each iteration in block 2 on line 14.

A second benchmark method is executed which is similar to the first benchmark method in Listing 6.4 except for an additional guard for the `use` method call inside the loop-body. It only lets the point instance escape if the loop-counter is even (i.e. in each second iteration). The pseudo-code for this second benchmark method is shown in Listing 6.5. The results are shown in Figure 6.5 together with the standard deviation. The orange, blue, and red graphs visualize the results of the first benchmark and are discussed first.

Listing 6.5: Second benchmark method in pseudo-code for the evaluation in Section 6.6.3

```
1 Point p = Point.createSet(3,4);
2 int x = p.x;
3 int y = p.y;
4 for (int i = 0; i < 100; i++) {
5   p = Point.setX(p,x+i);
6   if (i % 2 == 0) {
7     use(p); // Escape!
8   }
9 }
```

All methods of the first benchmark have the same number of allocations except for the two additional allocations in the start for the unoptimized value type version. However, these are negligible compared to the total number of 100 allocations. The results for the unoptimized value types and the object baseline are similar. The additional object constructor call has a minor impact compared to the expensive method call in each iteration. However, despite the fact of equally many allocations, the optimized value types still perform better (especially for 4, 8, and 16 fields). The reason is that even though the value type escapes and is allocated now, it does not need to perform any field loads from the heap. The values of the fields for the value type are statically known and thus can be used directly from the buffer to allocate the escaping value type instance properly. This advantage becomes even more significant with an increasing number of involved fields which require more loads from the heap during an allocation for objects and unoptimized value types. Therefore, not only the number of allocations is important for the performance but also the benefit of not needing to load a field from the heap. The optimized value type version performs between 6.1% (for two fields) and 46% (for four fields) better than the object baseline.

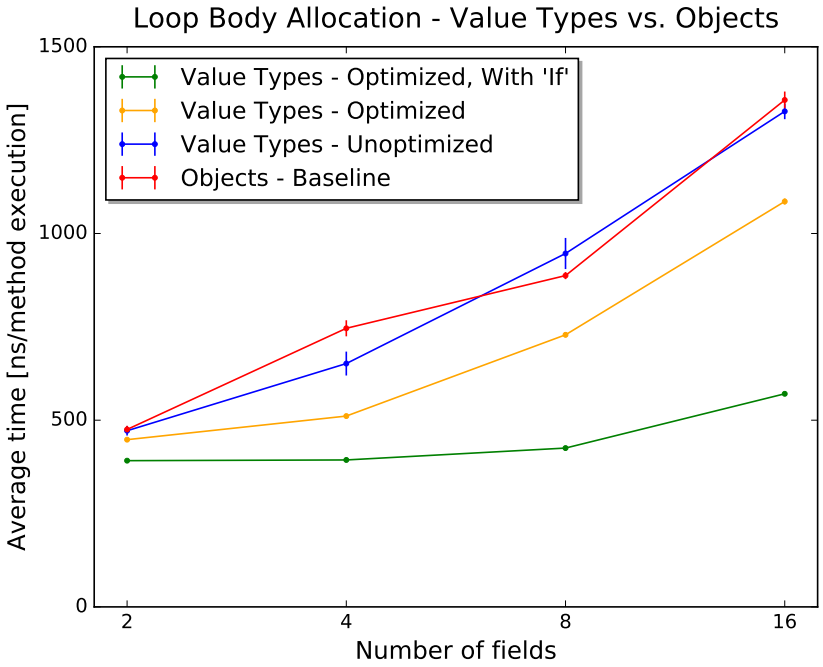


Figure 6.5: Results of the benchmarks of both method versions from Section 6.6.3

The second benchmark only lets the point instance escape in every second iteration. The results for the unoptimized value types and the object baseline are similar to those of the first benchmark and thus are omitted. However, the situation for the optimized value types is different. An allocation is only performed by calling the `use` method in every second iteration. This benchmark result is visualized as a green graph in Figure 6.5. The performance is very similar for 2, 4, and 8 fields but starts to get worse with 16 fields. The number of allocations is exactly half of those used in the first benchmark. Therefore, it could have been expected that half of the time is needed. However, since a lot of objects get allocated on repeatedly executing the benchmark method, garbage collection starts to become a dominant factor compared to the time spent for an allocation. This is additionally tested by running the second benchmark only for a short amount of time. The results matched the original expectation of spending half the time compared to the first benchmark. Nevertheless, at 16 fields, the average time spent suddenly starts to increase and corresponds to a value which is approximately twice as fast compared to the average time spent for allocating every time with 16 fields (visualized in the orange graph). This indicates that time spent for an allocation starts to dominate over the factor of garbage collection with 16 fields.

These results reveal how significant field loads from the heap are even when allocations cannot be removed. This is also further evaluated in Section 6.6.4. The second benchmark additionally shows the impact of garbage collection as an important factor for performance. Even though half of the allocations could have been removed with two fields, the performance gain was small compared to the dominating factor of garbage collection time. This can also be compared to the result of only one allocation as seen in the benchmark for Section 6.6.2. The gap between allocating once and 50 times is huge compared to the gap of allocating 50 and 100 times as shown in Figure 6.5.

6.6.4 Field Load Removal

This section evaluates the benefit of the allocation and field load removal approach to push a field value load out of a loop-body and reusing the immutable result instead. The benchmark consists of a loop that loads all available fields of a class and assign them to two local variables defined outside of the loop. Those are added together in the `return` statement afterwards[18]. The benchmark is again executed with 2, 4, 8, and 16 fields which corresponds to 2, 4, 8 and 16 different field loads in the loop-body, respectively. While the benchmark method for two fields just assign its variables to one of the local variables, the other benchmark methods use additions to assign both variables with the same amount of addends. The benchmark code for two fields is shown in Listing 6.6 (a) and an example of the benchmark code for eight fields is shown in Listing A.7 in Appendix A.5. The loop is executed 1000 times.

Listing 6.6: Benchmark method in pseudo-code (a) and the generated pseudo-HIR for optimized value types with two fields (b) for the benchmarks in Section 6.6.4

(a) Method in pseudo-code

```
1  // Block 1
2  Point p = Allocated.p;
3  int a = 0;
4  int b = 0;
5  for (int i = 0; i < 1000; i++)
6  {
7    a = p.x;
8    b = p.y;
9  }
10 return a + b;
```

(b) Pseudo-HIR for Optimized Value Types

```
1  // Block 1
2  p1 = <get> // p = Allocated.p
3  x1 = p1._x // LoadField
4  y1 = p1._y // LoadField
5  i0 = Constant(0) // a = b = i = 0
6
7    // Block 2
8    φi = [i0,i2]
9    φa = [i0,x1]
10   φb = [i0,y1]
11
12   i1 = Constant(1)
13   i2 = ArithmeticOp(φi+i2) // i++
14
15 // Block 3
16 ireturn φa + φb
```

The results are shown in Figure 6.6 together with the standard deviation. The graph for the unoptimized value types is omitted since no allocation is involved and thus it matches the performance for objects. The pseudo-HIR for the optimized value types is shown in Listing 6.6 (b) for two fields. The field loads are moved out of the loop (compared to the object baseline which performs a load on each iteration) due to the field load optimization described in Section 4.3.6[19]. Even though no allocations are involved, these field loads have a significant influence on the performance for

[18] This avoids that the variables are considered dead.
[19] The benchmark methods for 4, 8, and 16 fields also move all their field loads out of the loop.

an increasing number of field loads as shown in Figure 6.6[20]. The performance difference between optimized value types and the object baseline is between 18.1% (for two fields) and 69.4% (for four fields). Omitting field loads is clearly beneficial and should not be underestimated in long running loops. They are an important factor which was also shown in the evaluation in Section 6.6.3.

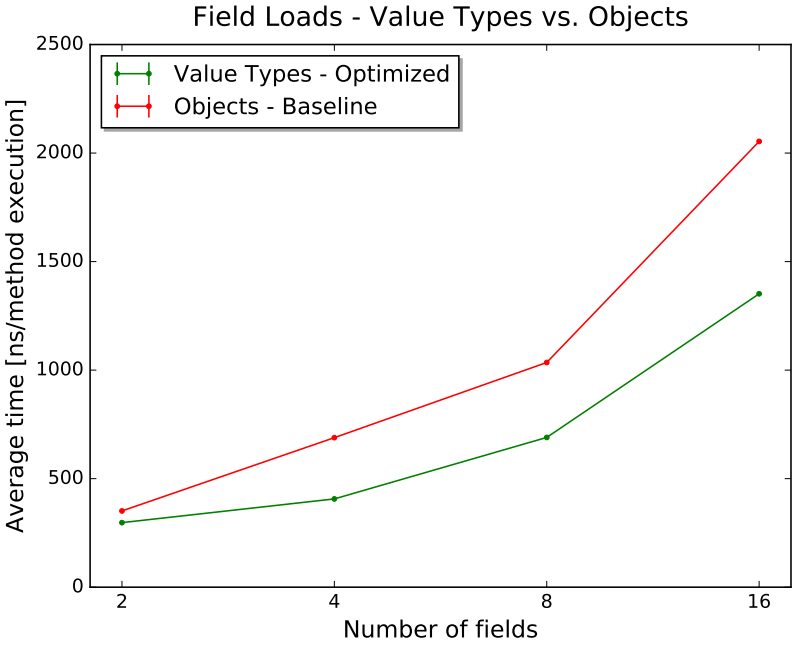


Figure 6.6: Results of the benchmarks of Section 6.6.4

6.7 Mathematical Applications and Inlining

This section evaluates two classic mathematical applications implemented with value types and the object baseline. These are only run with the optimized value type version (without the unoptimized version) to compare them directly to possible implementations found in the current Java SE 8 or 9 version with objects only. The benchmarks also show the importance of being aware of the internal inlining decisions made by the compiler.

6.7.1 Absolute Value of a Complex Number

The first benchmark creates a complex number and computes its absolute value. This is done with a class that contains two `double` fields for the real and imaginary part and a method to create such a complex number. Additionally, it provides a method for computing the absolute value of a complex number $z = x + yi$ with the following formula [10]:

$$|z| = |x + yi| = \sqrt{x^2 + y^2}$$

[20]The number of fields equals the number of field loads in the loop-body.

The benchmark pseudo-code is shown in Listing 6.7. A complex number c is created an then passed to the `abs` method to compute the absolute value with the formula above. The code uses the `Math::sqrt` method provided in the standard `java/lang/Math` class of the JDK. The benchmark is executed for the optimized value type implementation and for the object baseline version. The class `ComplexNumber` is a placeholder for the corresponding class for value types and objects.

Listing 6.7: Method for calculating the absolute value of a complex number

```
1 public void testAbs() {
2   ComplexNumber c = ComplexNumber.create(2.3, 3.4);
3   ComplexNumber.abs(c);
4 }
5
6 class ComplexNumber {
7
8   /* ... */
9
10  public static double abs(ComplexNumber i) {
11    return Math.sqrt(i.real*i.real + i.imaginary*i.imaginary);
12  }
13 }
```

6.7.2 Distance Between Two Integer Points in the Plain

The second benchmark creates two integer points in the plain and computes their distance from each other. This is done with the already used `VTPoint` and `OTPoint` class for value types and objects, respectively. An additional method computes the distance d between two integer points in the plain (x_1, y_1) and (x_2, y_2) with the following formula simply derived from the well-known Pythagorean theorem:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The benchmark pseudo-code is shown in Listing 6.8. Two points *p1* and *p2* are created and then passed to the `distance` method to compute the distance between both points with the formula above. The class `Point` is a placeholder for the corresponding class for value types and objects.

Listing 6.8: Method for calculating the distance of two integer points in the plain

```
1 public void testDistance() {
2   Point p1 = Point.createSet(3,4);
3   Point p2 = Point.createSet(5,6);
4   Point.distance(p1, p2);
5 }
6
7 class Point {
8   /* ... */
9
10  public static double distance(Point p1, Point p2) {
11    return Math.sqrt((p2.x - p1.x)*(p2.x - p1.x)
12                     + (p2.y - p1.y)*(p2.y - p1.y));
13  }
14 }
```

6.7.3 Results

The results for both mathematical benchmarks of Section 6.7.1 and 6.7.2 is shown in Figure 6.7. The mathematical `Math::sqrt` function is fast compared to an allocation. Therefore, it is not surprising that the computation of the absolute value with optimized value types is a lot faster than the object version. However, the results for the distance computation is equally fast for both. The problem with this test is that the compiler has decided not to inline the call to `distance` due to its too large size[21]. Thus, both points are allocated in the optimized value type version and then passed as arguments in the uninlined call of the `distance` method. The object version also creates both points with two allocations and thus performs equally well to value types. A third benchmark is performed which explicitly forces the compiler to inline the `distance` method[22]. The result is shown as an orange bar in Figure 6.7. The compiler has removed the allocations and thus the performance is, as expected, equally good as for the benchmark result of the absolute value with value types. The inlined optimized value type versions are 13.8 times faster for computing the absolute value of a complex number and 22 times faster for computing the distance of two points compared to the baseline versions with objects.

These results not only show that value types can be very beneficial for mathematical computations but also the importance of understanding how the compiler works internally. Without considering the inlining mechanism by the compiler, the results are confusing. Inlining decisions can suddenly change the performance significantly. This is a general problem which is intensified for value types especially inside a loop-body, by performing allocations due to unexpected uninlined method calls.

[21] This can be revealed by using the JVM flags `-XX:+UnlockDiagnosticVMOptions` together with the flag `-XX:+PrintInling`. The compiler prints the message about not inlining the call: "VTPoint::distance (52 bytes) callee is too large".

[22] This is done with the compile command `-XX:CompileCommand=inline,VTPoint::distance`.

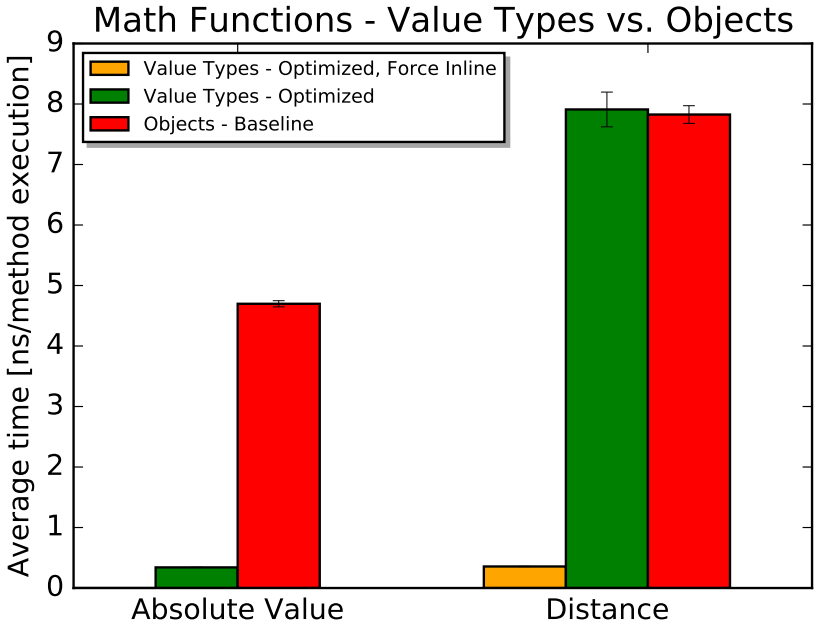


Figure 6.7: Results of the benchmarks of Section 6.7

7 Conclusion

Value types are a promising new type in Java's type system combining the advantages of primitive and reference types. Project Valhalla has been focusing on a value type implementation for the interpreter and the C2 compiler in the HotSpot™ JVM for quite some time. This thesis presented a first value type implementation for the currently missing C1 compiler support that focuses on eliminating value type allocations and field loads from the heap. This is achieved with a buffer concept for the immutable value types with its immutable fields.

The presented implementation adapts to the specific environment of the C1 compiler, which is quite complex. The value type approach directly affects the existing procedure of creating the HIR and later the LIR. The provided patch contains around 3100 changed lines of code in almost 40 files.

The evaluation shows that the allocation and field load removal optimizations are very beneficial for the performance and further avoid time spent for garbage collection compared to the use of objects in the baseline version. With no escape of a value type instance from a method body, the performance is as fast as using primitive types only since the allocations can be completely removed. This is especially useful in combination with many loop iterations as seen in the evaluation of Section 6.6.2 where all allocations could have been omitted in the loop-body and the entire method. Even when a value type escapes, the allocation can often be moved outside of a loop-body which saves a lot of time. The evaluation also shows that avoiding value type field loads from the heap alone, without an involved allocation, improves the execution time. The benchmarks for optimized value types of Section 6.6.3 and 6.6.4 resulted in a performance gain of 46% and 69.4% compared to the object baseline while having the same number of allocations. Moreover, inlining decisions made by the compiler should not be underestimated in association with value types. The choice between inlining and not inlining a method call with value type arguments can significantly influence the resulting program performance by having additional value type allocations, especially inside loop-bodies. This could be mitigated by a future optimization to only pass the value type field values to a method and thus bypassing the allocation (see Section 7.1). But even a method with only a single allocation with 16 fields for objects, the optimized value types are up to 35.4 times faster by removing the allocation as seen in Section 6.4.

The point in time when value types will eventually make it into standard Java SE is not yet clear. However, the ongoing development process of value types suggests that the feature might soon be included into a future release. This thesis enables not only support for value types

in the C1 compiler but also for the missing tiered-compilation with value types. It also serves as foundation and opportunity for a simultaneous continuing development of value types in the C1 compiler next to the interpreter and the C2 compiler, which was not the case up to date. New ideas or optimizations can be considered and applied directly to both just-in-time compilers. Possible opportunities for future work on value types in the C1 compiler are presented in the following section.

7.1 Future Work

There are some opportunities for future work which can also be seen as limitations of this value type implementation for the C1 compiler:

- **Missing Else-Block:** During the parsing stage of the C1 compiler, a single `if`-statement does not directly create an additional (empty) block for the missing `else`-statement. This is only done later after the HIR is completely created and the various optimizations have been applied. Changing this design would be beneficial for the value type implementation since there are situations where an allocation is inserted twice but the object version would only need one. An example is shown in Listing 7.1 using the `VTPoint` class from Section 6.6. If the condition is true, *p1* is allocated because it escapes over the uninlined method call on line 5. At line 7 an unallocated version from line 3 and an allocated version from line 5 are merged. An allocation is pushed towards the block containing line 3. However, it would also not make a difference if the allocation is inserted in the block containing line 7. The execution path will allocate twice in either way without an additional available `else`-block. Figure 7.1 additionally shows the bad benchmark results for the same environmental setup used in Section 6.6 for value types and objects executing the code from Listing 7.1. This could be improved to create an empty `else`-block earlier at the parsing stage of the HIR.
- **Value Type Arrays and Boxing:** The implementation for value types in the C1 compiler of this thesis does not support arrays containing value type instances and boxing instructions. This is left as a possible next step for a future development of value types in the C1 compiler.
- **Stack Value Type Phis:** The `javac` compiler stores every value in a variable since they would not be accessible from the code otherwise. However, bytecode can also be written by hand or by other tools or compilers. This requires not only to merge value types instances (and its fields) in variables but also a mechanism to merge different value type instances (with its fields) on the stack from different blocks with instance phi and field-phi functions. This is left for future work but should not create any problems as it is a straightforward extension.
- **Value Types as Fields:** Value types can also be used as a field in a normal class or in a value type class. Thus, the value type is either allocated and stored as a reference or stored in a flattened representation (i.e. the fields of the value type are inlined into the holder class). This is not supported by the implementation for the C1 compiler in this thesis and is left for future work.

- **Avoid Identical Allocations:** There are situations where an allocation is required in each iteration of a loop for the current approach as seen in Section 6.6.3. A future optimization could avoid such an allocation if all values are equal compared to the last iteration. An example is shown in Listing 7.2. A new value type instance is created on line 5 and is then allocated for the escape on line 6. However, the values are exactly the same on each iteration. Since value types have no identity, an optimization could try to take advantage of this fact and remove the additional allocations on each succeeding iteration.
- **Merge Method-Global and Block-Buffers:** The method-global buffer of the allocation and field load removal approach (see Section 4.3) might be later merged with the block-buffers from Section 4.4. This would make the overall design cleaner and more compact. The possibility of such a merging step could be explored in a future development.
- **Deoptimization:** This thesis only sets the basis for deoptimization support with value types. It needs some more effort to implement a working solution. The approach and the involved challenges described in Section 5.8 can be used as a possible starting point for future work.
- **Only Pass Value Type Fields:** This is an optimization that the C2 compiler already supports. A value type only passes its fields to and from an uninlined method call instead of the entire value type to bypass an allocation. The value type fields are treated as normal arguments by using registers or the stack. Adding support for this feature would drastically improve the performance for required allocations (for example, for the benchmark shown in Section 6.6.3). This could directly be implemented on the LIR and thus would mostly work independently of the optimizations done in this thesis on the HIR. However, this approach is only applicable to a certain degree and becomes inefficient for too many fields.
- **Split Phi Class:** The `Phi` class could be split into separate instruction classes for normal phi functions, value type field-phis, and possibly value type instance phis in the future. The current `Phi` class could be used as a superclass to avoid extensive changes in the C1 compiler. However, it could also be directly used for normal phi functions, while the value type phi classes are special subclasses. This would make the design cleaner and prevents wrong uses.
- **Use an enum for Phi Flags:** A minor improvement could merge the two flags `_value_type_instance` and `_value_type_field_phi` of the `Phi` class (see Section 5.2) together in combination with an `enum`. This could prevent wrong uses since a phi function is never an instance phi and a field-phi function at the same time.
- **Implementation of LOAD_REQUIRED:** Another minor improvement could change the implementation of the `MemoryBuffer::LOAD_REQUIRED` constant to point to a valid dummy `FieldLoad` instance which avoids a runtime crash if it is accessed by mistake.
- **Support Other Architectures:** The code contributed in this thesis only targets the Linux x86-64 architecture. Future work could also extend and test this approach on other architectures. This should not cause any problems as most parts of this thesis are done in the platform-independent HIR.

Listing 7.1: Benchmark method that shows the problem of a missing empty `else`-block during the parsing stage in the current C1 compiler implementation

```
1
2 public void limitation() {
3   VTPoint p1 = VTPoint.createSet(3,4);
4   if (/* some condition */) {
5     useVT(p1); // Escapes! Needs Allocation
6   }
7   useVT(p1); // Escapes! Needs Allocation
8 }
```

Listing 7.2: Example of reallocating a value type due to an escape in each loop iteration while it contains the same field values during the entire loop execution

```
1
2 public void avoidAllocation() {
3   VTPoint p1 = VTPoint.createSet(3,4);
4   for (int i = 0; i < 100; i++) {
5     p1 = VTPoint.createSet(3,4);
6     useVT(p1); // Escapes! Needs Allocation
7   }
8 }
```

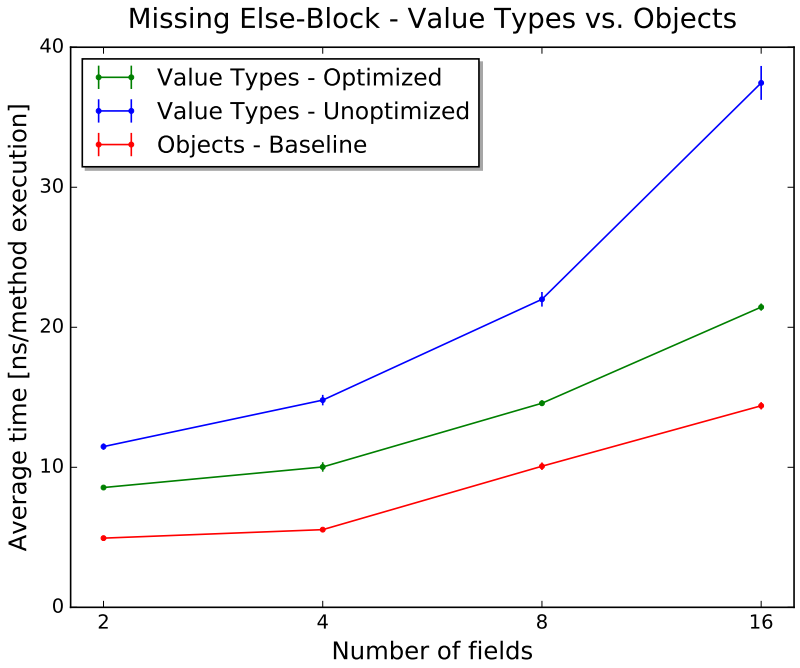


Figure 7.1: Benchmarks based on Listing 7.1 that demonstrates the bad performance of the optimized value types compared to objects due to a missing `else`-block for the allocation and field load removal approach of this thesis (see Section 4.3). This could be improved in future work.

A Appendix

A.1 Thesis Timeline

This section gives on overview of the development steps done in this thesis for the entire value type implementation for the C1 compiler. After the unoptimized implementation, which also served as a process to get familiar with the HotSpot™ JVM environment, the main focus of the thesis lied on the allocation and field load removal optimizations. Various bug fixes were performed along the work of this thesis. Some of them were even unrelated to the implementation of the thesis itself. Value types are an experimental feature which is still in development and thus is prone to errors and bugs. Some could have been avoided by workarounds or different flags but one major bug even required a merge of the entire source code to a newer version which contained a fix for it. The following list summarizes important development steps in chronological order:

- **Unoptimized Value Types:** Getting familiar with the entire HotSpot™ JVM and especially the C1 compiler to provide a simple support for the new value type bytecodes.
- **NewValueTypeInstance:** Introducing the new HIR instruction with an allocation-flag.
- **Method Global Field Value Buffer:** New buffer for storing statically known field values to remove allocations and field loads involving the heap.
- **Lazy Buffer:** Changing the design to support lazy buffering to omit the need to store default values.
- **Statically Unknown Values:** The buffer is extended with the possibility to support required field loads from the heap with the *LOAD_REQUIRED* constant. A field load is inserted every time this constant is returned from the buffer access methods.
- **State Cleaner:** Adding cleaning mechanism to remove unallocated or replaced value type instances from the HIR.
- **Phi and Field-Phi Function Support:** Extending the current phi function design to support value type instances. Additionally, the phi function implementation is adapted to provide a phi function for each value type field (i.e. field-phis). A new array is added in the state class, next to local variables and stack values, to provide an easy access to all field-phis. The current macros for iterating all phi-functions are also extended to process the new field-phi functions for value types.

- **LIR and Field-Phi Functions:** Implementation of additional methods to process the new field-phi functions with a correct replacement for the LIR, which does not have phi-functions.
- **Lazy Value Type Allocation:** Providing methods to insert an allocation in the current block or in predecessor blocks (already parsed) upon an escape of the value type through unlined method calls and return statements.
- **Lazy Field Load Insertions:** Providing a method to insert a field load in the current block upon its usage. An additional method handles insertions in predecessor blocks (already parsed). These guarantee that a field is loaded correctly on each possible path towards its usage.
- **Improved State Cleaner:** Adding additional cleaning methods for dead field-phis.
- **On-Stack-Replacement:** Exploring support for OSR-compilations.
- **Support for Loops:** An additional method handles the setup for the last unknown operand of value type instance phis and field-phi functions in loop-header blocks.
- **Inlining Support:** Excluding value type allocations for inlined method calls.
- **Field-phis and PhiSimpilifier:** Adapting the existing `PhiSimplifier` class to also consider field-phi functions.
- **Replace Optimization:** Directly replacing unallocated phi instances and field-phis marked as not-known in a merge block.
- **Loop-Body Allocation Optimization:** Avoiding an allocation push to loop-bodies in case the value type instance phi is only required to be allocated after the loop but not inside it.
- **Loop-Body Field Load Insertion Optimization:** Avoiding a field load insertion push to loop-bodies if the value type field-phi is only used after the loop but not inside it, all its operands depend on the same value type instance to load from, and they do not contain a statically known value directly or recursively.
- **Supporting Exception States:** Adding support to work with try-catch blocks with special exception states in the C1 compiler.
- **Deoptimization:** Exploration of deoptimization support for the C1 compiler in association with value types.

A.2 Used Tools/Frameworks

- **Eclipse:** The special Eclipse IDE for C/C++ developers [70] is used for the HotSpot™ JVM development.
- **c1visualizer:** The c1visualizer [64] visualizes the HIR and the LIR at different stages during the compilation of a method in the C1 compiler. The debug version JVM flag `-XX:+PrintCFGToFile` provides the input to the tool. This is useful for checking correctness during the development for any kind of changes to the intermediate representations, including those for value types.
- **gdb:** The gdb, GNU Project debugger, [8] was used for debugging the code during the development process of value types over the console.
- **Python:** All plots of this thesis are generated by Python scripts which use Matplotlib [15] as a 2D plotting library.
- **JMH with Maven:** All evaluations are done with the JMH framework [59] in association with the Maven [69] build tool.

A.3 Changes to JVM Flags

New JVM Flags

- **OptimizedValueTypes:** Enable or disable the value type optimizations for the C1 compiler (i.e. use the unoptimized or the optimized value type version). Usage:
  * `-XX:+OptimizedValueTypes` to enable value type optimizations (default)
  * `-XX:-OptimizedValueTypes` to disable value type optimizations (i.e. unoptimized version of value types described in Section 4.2 and 5.3)
- **PrintValueTypeDebugInfo:** Print debug information for the value type implementation in the C1 compiler. Usage:
  * `-XX:+PrintValueTypeDebugInfo` to enable value type debug messages
  * `-XX:-PrintValueTypeDebugInfo` to disable value type debug messages (default)

Changes to existing JVM Flags:

- **ValueTypePassFieldsAsArgs:** Added availability for the product build since the C1 compiler does not support this optimization feature yet.
- **ValueTypeReturnedAsFields:** Added availability for the product build since the C1 compiler does not support this optimization feature yet.

A.4 JMH Benchmark Method Mappings

The following Table A.1 shows which benchmark methods in the benchmark class `C1ValueTypeBench`[1] are used for which evaluation of this thesis. The square braces `[]` denote an optional postfix and the star `*` is replaced by the number of fields.

Table A.1: The JMH benchmark method mappings of each evaluation of this thesis to the benchmark class `C1ValueTypeBench`

Evaluation in	**Benchmark Methods**	**Classes**
Section 6.4: Cost of Allocation	testOT*/testVT*	OT[_*]/VT[_*]
Section 6.5: Cost of "vwithfield"	testOT*Set/testOT*Set2 testVT*Set	OT[_*]/VT[_*]
Section 6.6.1: Simple Branch	testOTPointIf[_*] testOTPointIfAlloc[_*] testOTPointIfAlloc[_*] testVTPointIfAlloc[_*]	OTPoint[_*] VTPoint[_*]
Section 6.6.2: Loops 1. Benchmark	testOTPointWhile[_*] testOTPointWhileAlloc[_*] testVTPointWhile[_*] testVTPointWhileAlloc[_*]	OTPoint[_*] VTPoint[_*]
Section 6.6.2: Loops 2. Benchmark	testOTPointNestedWhile[_*] testOTPointNestedWhileAlloc[_*] testVTPointNestedWhile[_*] testVTPointNestedWhileAlloc[_*]	OTPoint[_*] VTPoint[_*]
Section 6.6.3: Loop-Body 1. Benchmark	testOTPointWhileLoopAlloc[_*] testVTPointWhileLoopAlloc[_*]	OTPoint[_*] VTPoint[_*]
Section 6.6.3: Loop-Body 2. Benchmark	testOTPointIfWhileLoopAlloc[_*] testVTPointIfWhileLoopAlloc[_*]	OTPoint[_*] VTPoint[_*]
Section 6.6.4: Field Load	testOTFieldLoad[_*] testVTFieldLoad[_*]	OTPoint[_*] VTPoint[_*]
Section 6.7.1: Absolute Value	testOTComplexNumberAbs testVTComplexNumberAbs	OTComplexNumber VTComplexNumber
Section 6.7.2: Point Distance	testOTPointDistance testVTPointDistance	OTPoint VTPoint
Section 7.1: Missing Else-Block	testOTPointLimitation[_*] testVTPointLimitation[_*]	OTPoint[_*] VTPoint[_*]

[1] Defined in `/hotspot/test/compiler/valhalla/valuetypes/C1ValueTypeBench.java` in the provided patch in `http://cr.openjdk.java.net/~thartmann/thesis_hagedorn/webrev/hotspot.patch`

A.5 Additional Code Listings, Algorithms and Graphs

Algorithm 1: Algorithm for creating new value types and managing the buffer

```
if creating new value type instance vt then
    Add new empty field array entry to the buffer for vt;
    if created with vwithfield (vt.field = value) then
        Set the referenced field to the specified value in the new buffer-entry for vt;
        if buffer-entry for old instance of vt exists then
            Copy all other field values from the buffer-entry of the old instance to the new entry;
        else
            // First entry for vt
            Set all other field values in the new buffer-entry to the LOAD_REQUIRED
             constant;
        end
    end
end
```

Algorithm 2: Algorithm for accessing a field of a value type instance

```
if field access of value type instance vt (vt.field) then
    if vt instance exist in buffer then
        if buffered value equals LOAD_REQUIRED then
            Insert LoadField in HIR for field;
        else
            Insert buffered value for field in HIR.
        end
    else
        // vt is allocated has unknown field values
        Insert LoadField in HIR for field;
    end
end
```

Listing A.1: Example of a value type class illustrating its constraints together with a corresponding creation of a value type instance

```
__ByValue final class VTPoint { // final, no inheritance
  // Fields must be final
  final int x;
  final int y;
  //final VTPoint bad; // BAD: No cyclic/recursive references

  /* Dummy constructor, completely ignored */
  VTPoint() { this.x = 0; this.y = 0; }

  __ValueFactory static VTPoint create() {
    return __MakeDefault VTPoint(); // Creates value type instance
  }

  __ValueFactory static VTPoint set(VTPoint vt, int x, int y) {
    vt.x = x; // Creates a new value type instance with vt.x = x
    vt.y = y; // Creates a new value type instance with vt.y = y
    return vt;
  }
}

class Test {
  public static void main(String[] args) {
    // Creates a value type with default values x = 0 and y = 0
    VTPoint vt = VTPoint.create();

    // Creates a (new) value type vt2 with x = 3 and y = 4
    VTPoint vt2 = VTPoint.set(vt, 3, 4);
    /* Note: vt is immutable and still contains x = 0 and y = 0  */

    /* FORBIDDEN USAGES:
     *
     * Cannot assign null to a value type:
     * vt = null; // BAD
     * error: incompatible types: <null> cannot be converted to VTPoint
     *
     * No identity and no support for '==' or '!=':
     * if (vt == vt) { } // BAD
     * error: value types do not support ==/!=
     *
     * Value type creation only in value type classes:
     * vt = __MakeDefault VTPoint(); // BAD
     * error: This value factory cannot create values of type VTPoint
     *
     * Value type field assignment only in value factories in
     * value type classes:
     * vt.x = 3; // BAD
     * error: cannot assign a value to final variable x
     */
  }
}
```

Listing A.2: Merging branches of an allocated and unallocated instance *without* a required allocation later

```
void m() {
  VTPoint vt = VTPoint.create();
  if (/* some condition */) {
      uninlined(vt); // Escapes m() -> 'vt' needs to be allocated
  } else {
      /* Some code, NO allocation of 'vt' */
  }
  // Create phi node for vt, is it marked allocated?
  // 'vt' does not escape -> NO allocation needed
}

// Method NOT inlined
void uninlined(VTPoint vt) {
  /* Some code */
}
```

Listing A.3: The `VT` and `OT` class for 4 fields that are used as a basis for the evaluations of Section 6.4

```
__ByValue final class VT_4 {

  final int i1;
  final int i2;
  final int i3;
  final int i4;
  /* ... */

  __ValueFactory static VT_4 createDefault() {
    return __MakeDefault VT_4();
  }

final class OT_4 {

  final int i1;
  final int i2;
  final int i3;
  final int i4;

  /* ... */

  static OT_4 createDefault() {
    return new OT_4();
  }
}
```

Listing A.4: The VT and OT class for 4 fields that are used as a basis for the evaluations of Section 6.5

```
__ByValue final class VT_4 {

  final int i1, i2, i3, i4;

  /* ... */

  __ValueFactory static VTPoint setFirst(VT_4 vt, int a) {
    vt.i1 = a;
    return vt;
  }

final class OT_4 {

  final int i1, i2, i3, i4;

  // Approach (i)
  private OT_4(int i1, int i2, int i3, int i4) {
    this.i1 = i1;
    this.i2 = i2;
    this.i3 = i3;
    this.i4 = i4;
  }

  // Approach (ii)
  private OT_4(OT_4 ot, int a) {
    this.i1 = a;
    this.i2 = ot.i2;
    this.i3 = ot.i3;
    this.i4 = ot.i4;
  }

  // Approach (i)
  static OT_4 setFirst(OT_4 ot, int a) {
    return new OT_4(a, ot.i2, ot.i3, ot.i4);
  }

  // Approach (ii)
  static OT_4 setFirst2(OT_4 ot, int a) {
    return new OT_4(ot, a);
  }
}
```

Listing A.5: The `VTPoint` and `OTPoint` class for 4 fields that are used as a basis for all evaluations done in Section 6.6

```
__ByValue final class VTPoint_4 {

  final int x, y, i3, i4;

  /* All 3 methods are automatically inlined */
  __ValueFactory static VTPoint_4 createSet(int x, int y) {
    VTPoint_4 p = __MakeDefault VTPoint_4();
    p.x = x;
    p.y = y;
    return p;
  }

  __ValueFactory static VTPoint_4 setX(VTPoint_4 p, int x) {
    p.x = x;
    return p;
  }

  __ValueFactory static VTPoint_4 setY(VTPoint_4 p, int y) {
    p.y = y;
    return p;
  }
}

final class OT_4 {

  final int i1, i2, i3, i4;

  /* All 3 methods are automatically inlined */
  static OTPoint_4 createSet(int x, int y) {
    return new OTPoint_4(x,y);
  }

  static OTPoint_4 setX(OTPoint_4 p, int x) {
    return new OTPoint_4(x, p);
  }

  static OTPoint_4 setY(OTPoint_4 p, int y) {
    return new OTPoint_4(p, y);
  }
}
```

Listing A.6: Complete second benchmark method for the evaluation in Section 6.6.3

```
Point p = Point.createSet(3,4);
int x = p.x;
int y = p.y;
for (int i = 0; i < 100; i++) {
  p = Point.setX(p,x+i);
  if (i % 2 == 0) {
    use(p); // Escape
  }
}
use(p); // Escape
```

Listing A.7: Benchmark method in pseudo-code for 8 fields for Section 6.6.4

```
// Block 1
Point p = Allocated.p;
int a = 0;
int b = 0;
for (int i = 0; i < 1000; i++)
{
  // Both additions have equally many addends
  a = p.x + p.i3 + p.i4 + p.i5;
  b = p.y + p.i6 + p.i7 + p.i8;
}
return a + b;
```

Bibliography

[1] Allan Gottlieb, professor in the Computer Science Department within the Courant Institute of New York University. Data Structure. `http://cs.nyu.edu/courses/fall12/CSCI-UA.0102-001/recitation/recitation-02.pdf`. CSCI-UA.102: Data Structures 2012-13 Fall, Accessed: 18.09.2017.

[2] Andrew Binstock, OracleVoice contributer, editor in chief of Java Magazine. Java's 20 Years Of Innovation. `https://www.forbes.com/sites/oracle/2015/05/20/javas-20-years-of-innovation/#332bb25411d7`. Accessed: 20.09.2017.

[3] Brian Goetz. Welcome to Valhalla! `http://mail.openjdk.java.net/pipermail/valhalla-dev/2014-July/000000.html`. OpenJDK Mailing List - valhalla-dev, Accessed: 18.09.2017.

[4] Christian Wimmer. Synchronization. `https://wiki.openjdk.java.net/display/HotSpot/Synchronization`. OpenJDK Wiki, Accessed: 18.09.2017.

[5] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *IR '95 Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pages 35–49, 1995.

[6] cplusplus.com. Value Types (C# Reference). `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types`. Accessed: 10. November.

[7] ETH Professor Thomas Gross. Advanced Compiler Design (263-2810). `http://www.lst.inf.ethz.ch/education/archive/spring-2015/compiler-design.html`. Spring Semester 2016.

[8] Free Software Foundation. GDB: The GNU Project Debugger. `https://www.gnu.org/software/gdb/`. Accessed: 12. November.

[9] golang.org. The Go Programming Language Specification . `https://golang.org/ref/spec`. Accessed: 10. November.

[10] M. Gonzalez. *Classical Complex Analysis*. CRC Press; 1 edition, 1991. Page 19.

[11] Igor Veresov, member of the Oracle HotSpot compiler team. Tiered Compilation. `http://cr.openjdk.java.net/~iveresov/tiered/Tiered.pdf`. Presentation from 2013, Accessed: 14.10.2017.

[12] D. K. Ira Greenberg, Dianna Xu. *Processing, Creative Coding and Generative Art in Processing 2.* Springer Science+Business Media New York, originally published by Kluwer Academic/-Plenum Publishers, 2013.

[13] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[14] R. Jesse. *Static Single Assignment Form.* Book on Demand Pod, 2015. Also available at `http://ssabook.gforge.inria.fr/latest/book.pdf`, Accessed: 15.10.2017.

[15] John Hunter. GDB: The GNU Project Debugger. `https://www.gnu.org/software/gdb/`. Accessed: 12. November.

[16] John Rose. HotSpot Glossary of Terms. `https://wiki.openjdk.java.net/display/HotSpot/Garbage+Collection`. OpenJDK Wiki, Accessed: 18.09.2017.

[17] John Rose, Brian Goetz. Minimal Value Types. `http://cr.openjdk.java.net/~jrose/values/shady-values.html`. April 2017: Minimal Edition (v. 0.4).

[18] John Rose, Brian Goetz, and Guy Steele. State of the Values. `http://cr.openjdk.java.net/~jrose/values/values.html`. April 2014: Infant Edition.

[19] John Rose, Brian Goetz, and Guy Steele. State of the Values. `http://cr.openjdk.java.net/~jrose/values/values.html`. April 2014: Infant Edition; Paragraph: Running example: Point.

[20] John Rose, Brian Goetz, and Guy Steele. State of the Values. `http://cr.openjdk.java.net/~jrose/values/values.html`. April 2014: Infant Edition; Paragraph: Background.

[21] Karen Kinnear. Minimal Value Types. `https://wiki.openjdk.java.net/display/valhalla/Minimal+Value+Types`. OpenJDK Wiki, Accessed: 18.09.2017.

[22] S. Kumar. *Clojure High Performance Programming - Second Edition.* Packt Publishing Ltd., 2015. Page: 80.

[23] P. T. Lorenzo Magnani, Nancy J. Nersessian. *Model-Based Reasoning in Scientific Discovery.* Apress, 1999.

[24] Martijn Verburg. Valhalla_Goals. `https://wiki.openjdk.java.net/display/valhalla/Valhalla_Goals`. OpenJDK Wiki - 1. Align JVM memory layout behavior with the cost model of modern hardware, Accessed: 19.09.2017.

[25] Martijn Verburg. Valhalla_Goals. `https://wiki.openjdk.java.net/display/valhalla/Valhalla_Goals`. OpenJDK Wiki - Goals, Accessed: 19.09.2017.

[26] Martin Odersky, Jeff Olson, Paul Phillips and Joshua Suereth. SIP-15 - Value Classes. `http://docs.scala-lang.org/sips/completed/value-classes.html`. Official Scala Documentation, Accessed: 20.09.2017.

[27] C. C. Michael Paleczny, Christopher Vick. The java hotspot server compiler. In *JVM'01 Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, 2001.

[28] Microsoft. Tutorials. `http://www.cplusplus.com/doc/`. Accessed: 10. November.

[29] Motohiro Kawahito, Hideaki Komatsu and Toshio Nakatani. Effective null pointer check elimination utilizing hardware trap. *ACM SIGARCH Computer Architecture News - Special Issue: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS '00)*, 28(5):139–149, Dec. 2000.

[30] Oracle. 3 JDK and JRE File Structure. `https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/`. Java Platform, Standard Edition Tools Reference, Accessed 10.10.2017.

[31] Oracle. Chapter 4. The class File Format . `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html`. Java Virtual Machine Specification, Accessed 09.10.2017.

[32] Oracle. Chapter 4. Types, Values, and Variables. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html`. Java Language Specification, Accessed: 18.09.2017.

[33] Oracle. Class MethodHandle. `https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html`. Java Platform Standard Ed. 8 documentation, Accessed: 20.09.2017.

[34] Oracle. Class MethodHandle, Interoperation between method handles and Java generics. `https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html`. Java Platform Standard Ed. 8 documentation, Accessed: 20.09.2017.

[35] Oracle. HotSpot Runtime Overview. `http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#Interpreter|outline`. Section "Interpreter", Accessed 10.10.2017.

[36] Oracle. jar-The Java Archive Tool. `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html`. Java SE Documentation, Accessed: 09.10.2017.

[37] Oracle. Java HotSpot^{TM} Virtual Machine Performance Enhancements. `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis`. Accessed: 20.09.2017.

[38] Oracle. Java HotSpot VM Options. `http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html`. Accessed 10.10.2017.

[39] Oracle. Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide. `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html`. Chapter 5, Available Collectors, Accessed: 14.10.2017.

[40] Oracle. Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html. Chapter 6, The Parallel Collector, Accessed: 14.10.2017.

[41] Oracle. Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/concurrent.html. Chapter 7, The Mostly Concurrent Collectors, Accessed: 14.10.2017.

[42] Oracle. Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html. Accessed 14.10.2017.

[43] Oracle. Java Platform, Standard Edition Nashorn User's Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/. Accessed 09.10.2017.

[44] Oracle. Java SE Downloads. http://www.oracle.com/technetwork/java/javase/downloads/index.html. Java SE 8 Edition, Accessed: 20.09.2017.

[45] Oracle. Java SE Technologies. http://www.oracle.com/technetwork/java/javase/tech/index.html. Accessed: 20.09.2017.

[46] Oracle. javac. http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html. Official Oracle Java Documentation, Accessed: 20.09.2017.

[47] Oracle. JEP 193: Variable Handles. http://openjdk.java.net/jeps/193. Accessed: 12. November.

[48] Oracle. JEP 218: Generics over Primitive Types. http://openjdk.java.net/jeps/218. Accessed: 12. November.

[49] Oracle. Oracle Java SE Embedded: Developer's Guide, Codecache Tuning. https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/codecache.htm. Official Oracle Java Documentation, Accessed: 14.10.2017.

[50] Oracle. Package java.util.concurrent.atomic. https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/atomic/package-summary.html. Accessed: 12. November.

[51] Oracle. Primitive Data Types. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html. Official Oracle Java Documentation, Accessed: 18.09.2017.

[52] Oracle. Source Code, src/share/vm/runtime/advancedThresholdPolicy.hpp @ 5820:87ee5ee27509. http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/runtime/advancedThresholdPolicy.hpp. OpenJDK, jdk8, hotspot, Accessed: 14.10.2017.

[53] Oracle. Specification for Value Classes. http://cr.openjdk.java.net/~dlsmith/values.html. June 2017.

[54] Oracle. the Da Vinci Machine Project. http://openjdk.java.net/projects/mlvm/. a multi-language renaissance for the Java™ Virtual Machine architecture, Accessed 09.10.2017.

[55] Oracle. The HotSpot Group. http://openjdk.java.net/groups/hotspot/. Accessed: 09.10.2017.

[56] Oracle. The Java Language Specification. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf. Java SE 8 Edition, Accessed: 20.09.2017.

[57] Oracle. The Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf. Java SE 8 Edition, Accessed: 20.09.2017.

[58] Oracle. Valhalla. https://wiki.openjdk.java.net/display/valhalla/Main. Official OpenJDK Website, Accessed: 18.09.2017.

[59] Oracle - OpenJDK. Code Tools: jmh. http://openjdk.java.net/projects/code-tools/jmh/. Accessed: 28. October.

[60] Oracle - OpenJDK. HotSpot Glossary of Terms. http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html. Accessed: 18.09.2017.

[61] Oracle, Owner: Jon Masamitsu. JEP 122: Remove the Permanent Generation. http://openjdk.java.net/jeps/122. Accessed 10.10.2017.

[62] Oracle, Owner: Vladimir Kozlov. JEP 295: Ahead-of-Time Compilation. http://openjdk.java.net/jeps/295. Accessed 10.10.2017.

[63] Preston Briggs, Keith D. Cooper and L. Taylor Simpson. Effective null pointer check elimination utilizing hardware trap. *Software—Practice & Experience*, 27(6):701–724, June 1997.

[64] Provided on www.openhub.net. Official Website. https://www.openhub.net/p/c1visualizer. Accessed: 28. October.

[65] Roger Smith, prior acquisition editor for Oracle and former senior editor at InformationWeek. Why Java is the most popular programming language. http://www.theserverside.com/feature/Why-Java-is-the-most-popular-programming-language. Accessed: 20.09.2017.

[66] 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 (TOPLAS)*, 13(4):451–490, Oct. 2008.

[67] Simon Ochsenreither. ScalaDays 2015 – Project Valhalla - Part 2: Value Types in the JVM. https://soc.github.io/talks/scaladays-2015.html, 2015. Java Virtual Machine Specification, Accessed: 19.09.2017.

[68] Srikanth Adayapalam. hg: valhalla/valhalla/langtools: Enhancement: Add support for alternate mode of value creation using __MakeDefault. http://mail.openjdk.java.net/pipermail/valhalla-dev/2016-November/002091.html. OpenJDK Mailing List - valhalla-dev, Accessed: 19.09.2017.

[69] The Apache Software Foundation. Apache Maven Project. `https://maven.apache.org/`. Accessed: 28. October.

[70] The Eclipse Foundation. Eclipse IDE for C/C++ Developers. `https://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/marsr`. Accessed: 12. November.

[71] 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 No. 7):1–32, Sept. 2008.

[72] TIOBE. TIOBE Index for September 2017. `https://www.tiobe.com/tiobe-index/`. Accessed: 20.09.2017.

[73] C. Wimmer and M. Franz. Linear scan register allocation on ssa form. In *CGO '10 Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179, 2010.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Value Type Support in a Just-in-Time Compiler

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hagedorn	Christian

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Wädenswil, 29. November, 2017	C. Hagedorn

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.