Snippets: Taking the High Road to a Low Level

DOUG SIMON and CHRISTIAN WIMMER, Oracle Labs

BERNHARD URBAN, Institute for System Software, Johannes Kepler University Linz, Austria

GILLES DUBOSCQ, LUKAS STADLER, and THOMAS WÜRTHINGER, Oracle Labs

When building a compiler for a high-level language, certain intrinsic features of the language must be expressed in terms of the resulting low-level operations. Complex features are often expressed by explicitly weaving together bits of low-level IR, a process that is tedious, error prone, difficult to read, difficult to reason about, and machine dependent. In the Graal compiler for Java, we take a different approach: we use *snippets* of Java code to express semantics in a high-level, architecture-independent way. Two important restrictions make snippets feasible in practice: they are compiler specific, and they are explicitly prepared and specialized. Snippets make Graal simpler and more portable while still capable of generating machine code that can compete with other compilers of the Java HotSpot VM.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Compilers, Run-time environments

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Java, Graal, snippet, compiler, dynamic compilation, just-in-time compilation

ACM Reference Format:

Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, Thomas Würthinger. 2015. Snippets: Taking the high road to a low level. ACM Trans. Architect. Code Optim. 12, 2, Article 20 (June 2015), 25 pages.

DOI: <http://dx.doi.org/10.1145/2764907>

1. INTRODUCTION

Managed language virtual machines (VMs) rely on dynamic compilation to achieve competitive peak performance. The compilers leverage knowledge of internal VM data structures to generate optimized machine code for high-level operations, for example, object allocations or type checks. These dependencies tightly couple the compiler to the rest of the VM: improving memory allocation or a type-check algorithm typically requires compiler changes. Furthermore, it is difficult to express the semantics of high-level operations concisely. Most compilers either express semantics directly in assembly language, which complicates porting to new architectures, or manually construct compiler intermediate representation (IR), a tedious and error-prone process.

We present *snippets* as a mechanism to express the low-level semantics of a high-level operation using a high-level programming language. For example, the semantics of a type check in Java are expressed as a snippet written in Java itself. The compiler uses snippets to *lower* the IR, replacing high-level IR with low-level IR. Because they are written in a high-level language, most snippets are architecture independent,

Authors' address: D. Simon, C. Wimmer (corresponding author), B. Urban, G. Duboscq, L. Stadler, and T. Würthinger, Oracle Labs, 500 Oracle Parkway, Redwood Shores, CA 94065; email: christian.wimmer@oracle.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

2015 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2015/06-ART20 \$15.00

DOI: <http://dx.doi.org/10.1145/2764907>

which facilitates porting the compiler to new architectures. Snippets are specific to the optimizing compiler; that is, the snippet leverages internals of the compiler IR that are not accessible to normal source code. In addition, snippets are *prepared* and *specialized* to minimize their compilation time overhead.

Metacircular Java VMs (Java VMs written completely in Java) define the semantics of high-level Java operations using methods written in lower-level Java. The dynamic compiler in the Jikes RVM [Alpern et al. 1999, 2000; Rogers and Grove 2009], the first metacircular Java VM, inlines these methods without formalization and without an abstraction layer between compiler and VM. The original dynamic compiler in the metacircular Maxine VM [Wimmer et al. 2013], called *CPS*, first introduced the term *snippet* [Mathiske 2008; Simon 2009] to formalize the compiler–VM interface and to control inlining of Maxine snippets¹ via a bootstrapping process. CPS failed to meet the project’s goals for machine code quality and compilation speed and was eventually replaced.² Maxine’s second compiler, called *CIX*, achieved compiler-runtime separation by abandoning Maxine snippets in favor of XIR [Titzer et al. 2010], a new architecture-independent low-level language. We eventually concluded that the failure of Maxine snippets was a consequence of generality: being compiler independent, Maxine snippet implementations could not leverage compiler internals for efficiency. This, together with nonsnippet issues, constrained CPS to generating nonoptimal machine code too slowly.

Despite the inadequacy of Maxine snippets, the appeal of the snippet concept remained. We carefully analyzed the deficiencies and addressed them in a new implementation of snippets for the Graal compiler. Graal is a new dynamic compiler, written in Java, for the Java HotSpot VM and other VMs. As such, it has to compete with the highly tuned *client* and *server* compilers in the HotSpot VM, which are written in C++, in terms of both compilation speed and code quality. Graal snippets succeed because we restrict their generality: snippets are compiler specific, and they are prepared and specialized before use. This article presents the details of why we believe these restrictions are important.

Many benefits of Maxine’s snippet concept remain. Most snippets are architecture independent. They are not part of the compiler but provided by the VM, which makes the compiler independent of the VM. This independence allows us to integrate Graal in multiple VMs (including the Maxine VM), in addition to the original integration into the HotSpot VM. Snippets are written in Java code in a style that is intuitive and easy to understand. This makes it possible for noncompiler experts to provide new snippets when improving algorithms or data structures in the VM.

We evaluate the number and size of snippets provided to Graal by the HotSpot VM, as well as the compilation time spent in lowering snippets. To demonstrate that snippets can be used in a high-performance environment, we compare Graal with the client and server compilers of the Java HotSpot VM. Graal nearly reaches the performance of the server compiler. However, this comparison does not show the relative performance of using snippets versus not using snippets, because all three compilers are based on different IRs and perform different optimizations.

Although this implementation is based on Java, the concept of snippets is language independent. It can be applied in compilers for any programming language where high-level operations need to be lowered. The reflective and declarative features of Java that we use are available in most managed languages running on a VM, for example, in C# and other languages running on the .NET framework.

¹We use the term *Maxine snippet* to refer to their first incarnation in Maxine; every other use of the term *snippet* refers to the redesigned and reimplemented snippets presented in this article.

²<https://kenai.com/projects/maxine/lists/users/archive/2011-03/message/14>.

In summary, this article contributes the following:

- We show that snippets are an intuitive and practical approach to expressing the low-level semantics of high-level operations.
- We present the life cycle and optimization stages of snippets.
- We present implementation details of snippets for the Graal compiler, a dynamic compiler for Java.
- We evaluate the usage of snippets and show that code generated by Graal can compete with the production-quality server compiler in the HotSpot VM.

2. SNIPPET DESIGN

This section presents the implementation-independent ideas of snippets. Subsequent sections introduce the implementation and show the ideas in the context of a high-performance Java VM.

Snippets are written in the same high-level language that the compiler is compiling. They are parsed into the same intermediate representation (IR) as regular methods, and many compiler optimizations can be reused to optimize them. Lowering of a high-level instruction is then a straightforward inlining of the snippet IR into the method IR, without the need for any IR conversion. If the high-level language does not allow system programming, for example, raw memory access, these facilities need to be added (see Section 6.1).

Snippets are *VM specific*: they access details of VM-internal data structures, for example, object headers and type information maintained within the VM. They are part of the compiler-VM interface; that is, they are provided by the VM to the VM-independent compiler (see Section 3).

Snippets are *compiler specific*: to unlock the full potential of compiler optimizations, the snippet implementer must be able to influence low-level aspects of the generated IR such as branch probabilities (see Section 6.6) or scheduling constraints of IR nodes (see Section 6.9). Balancing this need for low-level control with the overall goal of writing snippets in a high-level language and style is an important design point. We use *node intrinsics* to introduce specific IR nodes by means of simple, readable method calls (see Section 5.2). The set of available node intrinsics is not fixed. The snippet implementer can add new IR nodes and use them via node intrinsics. It could be possible to provide a core set of intrinsics to make at least some important snippets compiler independent. We do not explore this idea in this article.

Lowering is a frequent operation, so repetitive parsing and optimization of the same snippet must be avoided to achieve good compilation speed. We define three distinct steps to achieve that (see Section 5.1): *Snippet preparation* produces the initial IR for a snippet. *Snippet specialization* replaces certain snippet parameters with constants and performs constant folding. In the literature, this is also called partial evaluation; however, we use the term *constant folding* within this article since it is more intuitive and reflects the actual compiler optimization that performs the work. Specialization is crucial to eliminate the compilation time overhead of snippets. It allows, for example, configuration parameters (see Section 6.2) and snippets with a variable number of inputs (see Section 6.4). *Snippet instantiation* performs the actual inlining of the specialized snippet IR into the compiled method.

3. SYSTEM STRUCTURE

Graal is an aggressively optimizing compiler written in Java, compiling Java bytecodes to machine code. Graal's clearly defined API and modular structure make it useful in multiple configurations. For the purpose of this article, we use Graal as a dynamic compiler in the HotSpot VM. The product version of the HotSpot VM comes with two

Fig. 1. System structure of the Graal compiler integrated into the Java HotSpot VM.

compilers: the *client* compiler [Kotzmann et al. 2008] and the *server* compiler [Palczyński et al. 2001]. Graal replaces these compilers. Regardless of the compiler being used, method execution starts in the interpreter: only frequently interpreted methods are scheduled for compilation. The compilers use profiling information, collected by the interpreter, for speculative optimizations.

Graal is also used in other configurations. For example, the Maxine VM uses Graal as the ahead-of-time compiler to generate the boot image, as well as the dynamic compiler at runtime. This reuse of one compiler in multiple VMs and configurations is facilitated by separating the compiler from its surrounding environment with a clear interface. The interface of Graal consists of three parts (see Figure 1):

- The inputs to the compiler are Java bytecodes and metadata such as the constant pool. The Graal API contains interfaces that provide information about types, methods, and fields. These interfaces are implemented by the VM.
- Java bytecodes contain high-level instructions such as memory allocations, type checks, and synchronization. The VM provides snippets that specify the semantics of these operations. Graal uses the snippets to lower high-level operations to low-level operations.
- The output of the compiler is optimized machine code, together with reference maps for the garbage collector and metadata for deoptimization [Hölzle et al. 1992].

Snippets can be seen as the “glue” that holds the VM and the compiler together. Although they are dependent on both the compiler and VM, they allow the rest of the compiler (including all standard compiler optimizations) to be VM independent. The remaining sections of this article focus on snippets and how they are used for lowering.

3.1. Graal Intermediate Representation

The intermediate representation of Graal is structured as a directed graph in static single assignment (SSA) form [Cytron et al. 1991]. Each IR node produces at most one value. To represent data flow, a node has *input* edges pointing to the nodes that produce its operands. To represent control flow, a node has *successor* edges pointing to its successors. In summary, the IR graph is a superposition of two directed graphs: the data-flow graph and the control-flow graph. This structure is illustrated in Figure 2. Note that the two kinds of edges point in opposite directions.

Nodes are not necessarily fixed to a specific point in the control flow. The control-flow graph provides a backbone around which most other nodes are *floating*. For example,

Fig. 2. Example Graal IR graph.

```

class A { ... }
class B extends A { ... }
List<A> list = new ArrayList<>();
A getElement(int idx) {
    return list.get(idx);
}

```

(a) Java source code.

```

0: aload_0
1: getfield list
4: iload_1
5: invokeinterface List.get
10: checkcast A
13: areturn

```

(b) Java bytecodes of method getElement.

Fig. 3. Example that uses a Java type check.

the IsNullNode and ReadNode in Figure 2 are floating. These floating nodes are only constrained by their data-flow edges, that is, input values as well as additional dependencies such as memory dependencies and guarding dependencies. The dependencies maintain the program semantics but allow more freedom of movement for operations. When a node needs to express a dependency to a specific branch, it can have an input edge pointing to a specific BeginNode. In our example, the ReadNode must not be scheduled before the IfNode that checks whether its input object is non-null. Therefore, the ReadNode has a data-flow edge to the BeginNode. The design of the IR has been described elsewhere [Duboscq et al. 2013a].

Speculative optimizations leverage deoptimization [Hölzle et al. 1992], which transfers execution from optimized machine code to the interpreter. Deoptimization requires knowing (1) where we want to continue interpreting the code and (2) how to reconstruct the interpreter state at the continuation point from the machine state of the optimized code. In order to know where we want to continue, we keep a reference to the method and the bytecode index. For the VM state, we keep a mapping of the local variables and operand stack slots to their values in the IR. We can then map them to their machine location when emitting deoptimization metadata. In the Graal IR, a deoptimization point is an explicit DeoptimizeNode. In Figure 2, assume that the method has never been called with a null argument yet. Therefore, the branch that returns -1 is not compiled, but speculatively replaced by a DeoptimizeNode. Details about the deoptimization handling in the Graal IR have been published before [Duboscq et al. 2013b].

4. RUNNING EXAMPLE

We use Java type checks (the instanceof and checkcast bytecodes) as the running example for this article. Figure 3(a) shows a common usage of a generic Java collection class. Although not obvious from the Java source code, the method getElement contains

```

// [omitted: method prologue, execute List.get]
// rax: result of List.get
mov ecx, [rax + 8] // load hub, located at offset 8 in the object header
cpl ecx, 0xF800DB92 // compare with hub for A (compile time constant)
jnz L1
// [omitted: method epilogue]
ret // return rax, the result of List.get
L1: call deoptimize // call into VM runtime to perform deoptimization

```

Fig. 4. Optimized machine code for type check in method getElement of Figure 3.

```

@Snippet
static Object instanceOfWithProfile(
    Object object, @ConstantParameter boolean nullSeen,
    @VarargsParameter Word[] profiledHubs, @VarargsParameter boolean[] hubIsPositive) {
    if (probability(NotFrequent, object == null)) {
        if (!nullSeen) {
            deoptimize(OptimizedTypeCheckViolated);
            throw shouldNotReachHere();
        }
        isNullCounter.increment();
        return false;
    }
    Anchor afterNullCheck = anchor();
    Word objectHub = LoadHub(object, afterNullCheck);
    explodeLoop();
    for (int i = 0; i < profiledHubs.length; i++) {
        if (profiledHubs[i].equal(objectHub)) {
            profileHitCounter.increment();
            return hubIsPositive[i];
        }
    }
    deoptimize(OptimizedTypeCheckViolated);
    throw shouldNotReachHere();
}

```

Fig. 5. Source code of snippet for lowering of a Java type check when profiling information is available.

a type check. Since the JVM erases the type of generics, the result of the generic collection access must be checked to ensure that it is of type A. Figure 3(b) shows the bytecodes generated by the Java source compiler. In practice, such type checks never fail. Assume that in our example the list contains only instances of class A, that is, not even instances of the subclass B. Based on profiling information collected by the interpreter, the optimizing compiler speculates that the returned value has the exact type A. Speculation results in the highly optimized machine code shown in Figure 4. The figure shows the actual code emitted by Graal as the optimizing compiler of the HotSpot VM. The type information of an object, called the “hub,” is loaded from offset 8 of the result object. It is compared to the hub of type A, which is embedded as a 32-bit literal constant in the machine code. If the type check fails, deoptimization transfers control back to the interpreter. Deoptimization eliminates the need to emit code that throws a `ClassCastException`.

Figure 5 shows the source code of the snippet for lowering the type check in the HotSpot VM when the profiling information about what types are expected to be seen is reliable. Different snippets are used for lowering when no profiling information is

available. The example source code is simplified to make it more readable but contains all functional aspects of the original code. The adventurous reader is encouraged to look at the original code.³

A snippet is a static Java method with the Java annotation `@Snippet`. A normal, not annotated, method parameter, in our example the object parameter, is an input value that the snippet operates on. A parameter node is replaced with the actual IR node for the corresponding input when the snippet is instantiated. A method parameter annotated with `@ConstantParameter` is a snippet configuration parameter. It is replaced with a constant during snippet specialization, so constant folding eliminates all usages of it (see Section 6.2). A parameter annotated with `@VarargsParameter` is expanded to multiple parameter nodes, which are replaced with multiple inputs when the snippet is instantiated. The snippet is specialized for every distinct array length (see Section 6.4).

The parameters in our example have the following meaning: `object` is the Java object reference that the type check is performed on; `nullSeen` is a Boolean value from the profiling information indicating whether the object reference was ever null; `profiledHubs` is an array coming from the profiling information with one entry for every type that was seen during profiling; and `hubIsPositive` is an array specifying for every profiled type whether the `instanceof` check returns true or false. Currently, type profiling records a maximum of eight types; that is, the arrays have a length between 1 and 8 (both arrays always have the same length). Together with the two different values for `nullSeen`, snippet specialization can result in up to 16 specialized graphs for the snippet. The loop accessing the arrays is exploded; that is, the loop body is duplicated for every array index. This information allows highly optimized speculative type checks. The snippet does not handle types that have never been seen during profiling but deoptimizes instead so that the profiling information can be updated in the interpreter.

Snippets need to use low-level features that are not accessible from standard Java source code. The type `Word` used in the source code represents a native unboxed machine word value (see Section 6.1). A snippet needs to trigger deoptimization when it encounters an object with a type that was not seen during profiling. We model this low-level access with *node intrinsics* (see Section 5.2). A node intrinsic is a special method call in the snippet. The method call is replaced with a single IR node during snippet preparation or specialization. In Figure 5, all node intrinsics are shown in italics. For example, the calls to `deoptimize` are replaced with the IR node for deoptimization, `DeoptimizeNode`. The method parameter (the deoptimization reason) is automatically mapped to a parameter of the `DeoptimizeNode` constructor. The `DeoptimizeNode` ends the control flow in the IR graph. However, the Java compiler does not know that; therefore, the call to `deoptimize` is followed by a `throw` statement to end the control flow also at the Java source code level.

The node intrinsics are tightly coupled with the compiler to get the best possible performance. They know about floating versus fixed nodes of the Graal IR. For example, the `loadHub` intrinsic creates a floating `LoadHubNode` that has a guarding data dependency, which is expressed using the `anchor` intrinsic (see Section 6.9). It is later lowered to memory operations that access the type information in the object header. Node intrinsics also interact with the machine code layout algorithm of Graal. For example, the `probability` intrinsic specifies the branch probability of the `if` statement (see Section 6.6).

Tracing output and counters are essential for debugging snippets and verifying that expected hot paths are really taken most frequently at runtime. It is desirable that such

³<http://hg.openjdk.java.net/graal/graal/file/28117ede7606/graal/com.oracle.graal.hotspot/src/com/oracle/graal/hotspot/replacements/InstanceOfSnippets.java>.

Fig. 6. Graal IR graph of `instanceOfWithProfile` when specialized for one profiled hub.

source code stays in the code base so that it can be activated when needed. Constant folding during snippet specialization enables this coding pattern: as long as the tracing code is guarded by an if statement whose condition can be bound to false during preparation or specialization, constant folding and dead code elimination remove the tracing code. In our example, the two counter increment statements are eliminated as long as counters are not enabled (see Section 6.3). If tracing counters are enabled, exhaustive method inlining inlines the increment methods so that the snippet remains free of method calls.

Figure 6 shows the Graal IR for the snippet when it is specialized for one profiled hub (the length of the `profiledHubs` and `hubIsPositive` arrays is 1) and no null value seen (the value of `nullSeen` is false). The loop is exploded, and the parameter nodes are the actual array elements (and no longer the arrays). In addition, the tracing counters are disabled. The resulting graph is compact, thanks to specialization, so instantiation needs to duplicate only a few nodes.

Figure 7 shows the source code for snippet instantiation. The code is simplified to make it more readable but contains all functional aspects of the original code. The `InstanceOfNode` is the high-level node to be lowered. It references the object to be checked (another IR node), as well as the type to be checked and the profiling information collected by the interpreter. The profiling information is converted from the VM metadata format to the two arrays `profiledHubs` and `hubIsPositive`, which the snippet expects as arguments. These arrays are only temporary data structures during compilation. The snippet is specialized for the particular array length (see Section 6.4) so that the individual array elements can be bound to the then-exploded formal arguments of the snippet graph. In other words, there is no array allocation or array access necessary when the compiled code is executed.

Figure 8 further illustrates the snippet preparation, specialization, and instantiation steps. It uses the context set in Figure 3, that is, the type check of a Java checkcast for type `A` in the method `getElement`. The left-hand side of Figure 8 shows the arguments that are filled by the lower method shown in Figure 7: The object to be checked is the IR node representing the result of the call to `List.get` in the example code. The profiling information conveys the information that this result was never null, so the argument


```

PreparedSnippet instanceOfWithProfile = prepareSnippet("instanceOfWithProfile");

void lower(InstanceOfNode node) {
    ValueNode object = node.getObject();
    JavaTypeProfile profile = node.getProfile();

    if (profile.totalProbability() > threshold) {
        int numTypes = profile.getNumTypes();
        Word[] profiledHubs = new Word[numTypes];
        boolean hubIsPositive = new boolean[numTypes];
        for (int i = 0; i < numTypes; i++) {
            profiledHubs[i] = profile.getType(i).getHub();
            hubIsPositive[i] = profile.isPositive(i);
        }

        Args args = new Args(instanceOfWithProfile);
        args.add(object);
        args.addConst(profile.getNullSeen());
        args.addVarargs(profiledHubs);
        args.addVarargs(hubIsPositive);

        SpecializedSnippet s = ensureSpecialized(args);
        s.instantiate(args, node);
    } else {
        // Use a different snippet.
    }
}

```

Fig. 7. Source code for instantiation of the `instanceOfWithProfile` snippet.

`nullSeen` has the primitive constant value `false`. The two arrays `profiledHubs` and `hubIsPositive` both have the length 1. Their first and only elements are the hub of type A (the constant `0xF800DB92`, embedded into the machine code in Figure 4) and the primitive constant `true`, respectively.

The top right-hand side of Figure 8 shows parts of the prepared snippet graph. The full graph would be too big; therefore, we focus on the parameter nodes and nodes that use them. Also, only data-flow edges are shown; control-flow edges are omitted. The `ArrayLengthNode` and `LoadIndexedNode` are accesses to the two arrays `profiledHubs` and `hubIsPositive` (recall the snippet source code shown in Figure 5). The nodes for the loop and the loop variable `i` are omitted for simplicity. All three dotted `i` boxes refer to the same IR node that represents the loop counter.

Snippet specialization uses the constant argument information to simplify the prepared snippet graph. The parameter `nullSeen` is replaced with the constant node `false`. The length of the `profiledHubs` array is used to replace the array length node with the constant node 1, as well as to explode the loop (see Section 6.4). This loop explosion removes the parameters `profiledHubs` and `hubIsPositive` and replaces them with their respective array element at index 0: `profiledHubs-0` and `hubIsPositive-0`. This transformation removes all array access nodes. The middle right-hand side of Figure 8 shows parts of the specialized snippet graph after the replacement of constants, but before constant folding and dead code elimination. Both the `IfNode` with the condition `false` and the comparison of the constants 0 and 1 are removed by these subsequent optimizations; that is, they are not contained in the final specialized snippet graph. The complete final specialized snippet graph is shown in Figure 6.

Snippet instantiation inlines this specialized snippet graph into the graph for the method `getElement`. The bottom right-hand side of Figure 8 shows the graph after inlining. The object parameter of the snippet is replaced with the input of the type check, that is, the IR node representing the result of the call to `List.get`. The

Fig. 8. Preparation, specialization, and instantiation of the `instanceOfWithProfile` snippet with the arguments from Figure 3.

parameters `profiledHubs-0` and `hubIsPositive-0` are replaced with constant values from the respective argument arrays. The `ReturnNode` from the snippet graph is no longer present; instead, the returned value replaces the `InstanceOfNode` at all of its usages. We show the graph again before constant folding.

The graph after instantiation resembles the ideal pattern for a highly optimized type check: the return value of the type check is the constant `true` because the snippet performs deoptimization instead of returning `false`. This constant `true` is going to simplify the code of `getElement`: constant folding and dead code elimination remove the IR nodes that are responsible for throwing a `ClassCastException`. This is one reason speculative optimizations with deoptimization lead to excellent peak performance.

The example machine code shown in Figure 4 is emitted from this instantiated snippet graph. The machine code contains the hub for type A as the literal constant `0xF800DB92`. The null check in the snippet graph (see Figure 6) does not require any machine code because the compiler back end can fold it into the memory access for the hub (the HotSpot VM uses traps for null checks).

5. ANATOMY OF A SNIPPET

A snippet is a static Java source code method annotated with `@Snippet`. One snippet is used to replace a single high-level IR node with one or more low-level IR nodes, often expressing VM-dependent logic. The latter nodes are obtained by processing and

Fig. 9. Life cycle of a snippet.

converting the snippet source method into an IR graph. The process of substituting a high-level IR node with the IR graph for a snippet is similar to replacing a method call node with the graph of the called method during inlining. We call this process *lowering*.

The parameters of a snippet typically represent inputs of the high-level IR node. They may also represent other properties of the IR node, for example, profiling information. In general, they can represent any value available during lowering, such as other IR nodes and constants. The latter are particularly useful for parameters that guard tracing or debugging source code. The value returned by the snippet replaces the high-level node at all of its usages.

The body of a snippet method expresses the semantics of the node being lowered. All control-flow constructs such as conditionals, loops, and method calls can be used. The latter encourages code reuse between snippets. An important constraint on snippets is that they cannot use operations they themselves implement. For example, a snippet for object allocation cannot do any explicit object allocation itself.

5.1. Snippet Lifecycle

Figure 9 shows the lifecycle of a snippet used for lowering. There are three distinct steps in this process:

- (1) **Preparation:** The Java source compiler is not aware of snippets and produces bytecodes in the same way as for any other method. No special treatment for snippets is necessary on the source code level. The bytecodes of a snippet source method are parsed into an IR graph. A number of transformations are then applied to the IR, some of which are snippet specific. The latter includes node intrinsicification (see Section 5.2), folding (see Section 6.2), exhaustive inlining, and word type replacement (see Section 6.1). The exhaustive inlining processes all calls except for specially handled methods, as discussed later. The preparation step is performed at most once per snippet. Its output is a prepared snippet graph.
- (2) **Specialization:** A prepared graph is duplicated and specialized by replacing certain parameters with constants and applying standard compiler transformations that benefit from constant values in the graph. Specialization usually reduces the graph size by eliminating source code guarded by the constant parameters. Node intrinsicification is rerun in this step to handle intrinsicifications that were deferred until

```

class LoadHubNode extends FloatingGuardedNode {
    @Input ValueNode object;

    LoadHubNode(ValueNode object, ValueNode guard) {
        super(guard);
        this.object = object;
    }
}

@NodeIntrinsic(LoadHubNode.class)
static native Word LoadHub(Object object, Object guard);

```

Fig. 10. Definition of a node intrinsic.

certain values become constant. The specialization step is executed only once for each unique combination of values bound to the constant parameters. Its output is a specialized snippet graph.

- (3) Instantiation: A high-level IR node is replaced with a duplicate of the specialized snippet graph. Parameters of the specialized snippet graph are bound to constants or values associated with the high-level IR node. The instantiation step is executed once for each snippet-lowerable IR node in the graph of a method being compiled. It is a simple replacement of the high-level node with a copy of the specialized snippet IR graph.

Given the relative execution frequencies in the lifecycle, it should be clear why doing as much work as possible in the first two steps is important in terms of making snippets feasible. They are both significantly slower than the third step and their results can be cached. The cache for the prepared graphs is indexed by the snippet method. The cache for the specialized graphs is indexed by the combination of the values bound to the constant snippet parameters. Also, splitting up snippet processing this way means snippet-specific transformations can be omitted from normal compilation. Such transformations would add more time to normal compilation as well as expose potentially unsafe capabilities to normal source code, for example, uses of the `Word` type (see Section 6.1).

5.2. Node Intrinsic

Snippets need to balance two opposing goals: the snippet source code should be readable high-level Java code, but it also needs to use low-level operations and interact with compiler internals in order to not sacrifice machine code quality. We solve this problem with *node intrinsics*, a simple and lightweight way to insert specific IR nodes by means of method calls. Our running example of Figure 5 contains many uses of such node intrinsics.

For example, `loadHub` accesses the object's header word, which contains a pointer to VM-internal type information. The hub needs to provide access to the `java.lang.Class` object but is often a separate data structure and therefore warrants a different name. It is not possible to implement this method in pure Java. Instead, Graal has an IR node called `LoadHubNode` for that purpose. The mapping between the `loadHub` method and the `LoadHubNode` is performed declaratively: the definition of the `loadHub` method is annotated with `@NodeIntrinsic`, with the IR node class as an attribute of the annotation. `LoadHubNode` is a standard IR node in Graal because many operations need to access type information. However, accessing the hub in the object header is VM specific. To ensure compiler-VM separation, the `LoadHubNode` is lowered to memory access nodes later on. Figure 10 shows the definition of the `LoadHubNode` and the node intrinsic.

```

class DeoptimizeNode extends ControlSinkNode {
    final Reason reason;

    DeoptimizeNode(Reason reason) {
        this.object = object;
    }
}

@NodeIntrinsic(DeoptimizeNode.class)
static native void deoptimize(@ConstantNodeParameter Reason reason);

```

Fig. 11. Node intrinsic with constant parameter.

The snippet source code operates on normal Java values of any primitive or object type. The compiler represents values abstractly as IR nodes, with `ValueNode` being one of the base classes. The parameters of a node intrinsic method are automatically mapped to parameters of a node's constructor. It is therefore required that the node intrinsic has the same number of parameters as a constructor of the node class. The *node intrinsicification* phase of Graal, which is performed during snippet preparation and specialization, performs the following steps for every call to a method annotated with `@NodeIntrinsic`:

- (1) Find the class of the IR node specified by the annotation.
- (2) Find the constructor in the class that matches the signature of the node intrinsic. The constructor must use the class `ValueNode` for every nonconstant parameter (see later) of the node intrinsic.
- (3) Using Java reflection, call the constructor with the actual arguments of the node intrinsic call.
- (4) Replace the invoke instruction of the intrinsic method with the newly instantiated node; that is, rewrite all usages of the `InvokeNode` to use the new node.

The method `loadHub` is defined as native so that we do not have to provide a body for it. Since this method is never called, there is no native implementation of it either.

So far, we have covered IR nodes that only have other nodes as parameters, that is, constructor parameters of type `ValueNode`. However, many nodes also have regular primitive or object parameters. In our running example, the `DeoptimizeNode` (which is created by the intrinsic method `deoptimize`) requires a `Reason` value to record the reason for the deoptimization. Figure 11 shows the node class and the intrinsic method definition. To support such nodes, step 2 of node intrinsicification has special support for constant parameters: when a parameter of the intrinsic method is annotated with `@ConstantNodeParameter`, the constructor can have the appropriate primitive or object parameter type. The actual parameter of the node intrinsic call must be a compile time constant, that is, a `ConstantNode`. The node intrinsicification phase extracts the constant value from the node, casts the value to the type of the constructor's parameter, and calls the constructor with the value.

If the actual parameter is not a constant during snippet preparation, node intrinsicification is deferred to snippet specialization. If it is not a constant by the time of snippet specialization, an error is reported. One way to ensure that the actual parameter is a constant is to use a literal constant in the intrinsic call, as it is done in both calls to `deoptimize` in Figure 5. The value can also come from a static final field, a `@ConstantParameter` of a snippet (see Section 6.2), a `@Fold` annotated method (see Section 6.2), or any other expression that can be constant folded.

Node intrinsics make snippets compiler specific: knowledge about the existence and internals of compiler IR nodes is necessary to write the appropriate node intrinsic

definitions. However, this knowledge makes the concept of snippets flexible and extensible, which is important for modularity. The set of available node intrinsics is not fixed, and the snippet implementer can add new IR nodes and use them via node intrinsics. For example, the snippet for lowering of a high-level node can use node intrinsics to create custom medium-level nodes. These nodes are defined by the snippet implementer and interact with optimization phases such as constant folding, global value numbering, or escape analysis using the standard API exposed and expected by these phases. Finally, these nodes are lowered to low-level nodes, again using snippets. To support this pattern, Graal currently performs lowering at three different stages in the standard compilation pipeline.

It could be possible to provide a core set of intrinsics to make at least some important snippets compiler independent. This would hide the node intrinsics behind an API that acts as a factory for all existing low-level nodes. However, it is not feasible or desirable to define all medium-level nodes; that is, giving up the concept of node intrinsics and switching entirely to a factory-based approach would significantly limit the expressiveness of snippets. The `LoadHubNode` is a medium-level node that is generally useful; therefore, it is in the VM-independent part of Graal. Other medium-level nodes that snippets use, for example, to access VM data structures during the lowering of write barriers, are not generally useful and therefore in the HotSpot-specific code. Node intrinsics allow us to handle both cases uniformly.

6. IMPLEMENTATION DETAILS

The previous sections presented the essence of snippets: high-level Java code with node intrinsics to use low-level operations. This section presents more details that enable the efficient use of snippets in practice.

6.1. Low-Level Java Programming

Memory access in Java is limited to named static or instance fields and elements of arrays. Further, the safety features of Java require a number of checks to be performed on access, for example, null pointer checks, type checks, and array bounds checks. Snippets require some amount of system programming for which raw memory access is essential. Therefore, snippets need a raw pointer type. Metacircular Java VMs such as the Jikes RVM [Alpern et al. 1999, 2000; Rogers and Grove 2009; Frampton et al. 2009] and the Maxine VM [Wimmer et al. 2013] pioneered this idea of low-level programming in Java. Similar to these VMs, we define a class `Word` to represent raw machine words. It is modeled as a class, and not as a primitive type such as `long`, to avoid accidental conversions of a number to a `Word`. A compiler phase changes all usages of `Word` to the primitive type `int` or `long`, depending on the bit width of the target architecture. Note that some managed languages, for example, C#, allow system programming as part of the core language, so a snippet implementation for and in C# would not need such a workaround.

6.2. Configuration Parameters

Modern VMs have many configuration parameters, for example, to select garbage collection or locking algorithms. The VM can have multiple snippets for one high-level operation and provide the compiler with the appropriate one based on the current configuration. However, in many cases, a configuration parameter only changes a small detail in a snippet, so it is not desirable to maintain multiple snippets. Snippet specialization, which performs constant folding, solves this problem: a snippet can use configuration parameters without any overhead regarding compilation time or machine code quality, as long as constant folding can reduce the branch condition to a

constant. We support multiple mechanisms to ensure that configuration parameters are constant:

- Java fields that are `final` are replaced with their constant value. This pattern is not limited to boolean fields; for example, we frequently use the idiom that a reference to a counter object is `null` when counting is disabled. Constant folding also eliminates chains of objects accessed via `final` fields. The root of the chain can either be a static `final` field or one of the other sources of constants described in this list.
- A snippet parameter annotated with `@ConstantParameter` indicates that the snippet is specialized for every distinct value. For example, a snippet can have a Boolean trace parameter annotated by `@ConstantParameter`, which is used to guard trace statements. The parameter value is provided when the snippet is instantiated. At this point, actual information about the high-level node that is lowered is available. For example, the trace parameter could be set to `true` when lowering an allocation node only if the name of the allocated type matches a certain pattern.
- If a snippet calls a method that is annotated with `@Fold`, the called method is executed at compile time via reflection. All parameters of the method must be constant themselves; otherwise, an error is reported. The result of the execution is converted to a constant, which replaces the method invocation node in the snippet graph. The `@Fold` annotation is useful to define abstractions that need more flexibility than a `final` field, which has strict rules under the Java specification. The annotated method must be pure; that is, it must not have side effects and must return the same result when called with the same parameters. No automatic check for pureness is performed, so it is the duty of the implementer to fulfill this contract.

6.3. Debugging and Profiling Snippets

The mechanism used for configuration parameters (see Section 6.2) is also useful for trace output and counters in snippets. During debugging, it is invaluable to have trace output. Removing trace statements from the source code for performance reasons and reinserting them during debugging is tedious, so trace statements remain in snippet source code. They are guarded by `if` statements whose conditions are constant folded to `false` when tracing is disabled. The same holds for counters that help finding performance bottlenecks and verifying that execution counts match the developer's assumptions. We leave these counters in place and rely on snippet specialization to remove the counting code unless counters are explicitly enabled.

Debugging snippets usually also involves looking at the IR of the snippet, as well as looking at the IR of a method before and after lowering. We use a compiler IR visualization tool originally developed for the HotSpot server compiler to visualize the Graal IR [Würthinger et al. 2008]. It shows the IR after every optimization phase, including the optimization phases applied during snippet preparation, specialization, and instantiation.

6.4. Varargs Parameters

As seen in our running example, snippets can have a variable number of inputs. A parameter annotated with `@VarargsParameter` represents zero or more inputs. The declared type of the parameter must be an array type. Such a snippet must contain a counted loop that processes all the elements in the array exactly once. The loop itself must be immediately preceded by an `explodeLoop` node intrinsic, which instructs the snippet specialization step to explode the loop, that is, fully unroll all of the constant number of loop iterations. The snippet is specialized on the length of the array argument passed to such a parameter. During the unrolling, array accesses of the array parameter are replaced with explicit parameters for every array element. The prepared snippet

```

class ReadRegisterNode extends FixedWithNextNode {
    final Register register;

    ReadRegisterNode(Register register) {
        this.register = register;
    }
}

@NodeIntrinsic(ReadRegisterNode.class)
static native Word readRegister(@ConstantNodeParameter Register register);

@Fold
static Register threadRegister() {
    return runtime.getThreadRegister();
}

@Snippet
static Word allocate([...]) {
    Word thread = readRegister(threadRegister());
    [...]
}

```

Fig. 12. Accessing the thread register in the allocation snippet.

has one array parameter, but the specialized snippet has multiple scalar parameters. These individual parameters are bound during snippet instantiation to individual input values.

6.5. Architecture Dependencies

Snippets can be architecture independent by using IR nodes and other abstractions over architectural details. For example, our instance allocation snippet uses thread local allocation buffers and so needs access to the register holding the current thread data structure. Figure 12 shows how we use a `readRegister` node intrinsic, the `@Fold` annotation (see Section 6.2), and an abstraction over the platform-specific thread register to write a snippet that is usable for all architectures supported by the HotSpot VM.

Snippets can also be used to handle architecture-specific details of lowering. For example, the `CVTSS2SI` instruction of the Intel x86 instruction set does not precisely match the semantics of the `f2i` bytecode for float-to-int conversion in Java. When the input is NaN, infinity, or larger than `Integer.MAX_VALUE`, the result of the conversion instruction needs to be intercepted. The `CVTSS2SI` instruction returns `Integer.MIN_VALUE` for all of these values. The extra logic to handle this check is expressed by the snippet in Figure 13. The lowering of this snippet is slightly different compared to the previous examples: the high-level architecture-independent conversion node is not replaced with the snippet graph, but the snippet graph is inserted after the high-level node. Additionally, the snippet gets the result of the high-level node as the parameter result (in addition to the usual input of the high-level node, the parameter input).

6.6. Branch Probability Injection

Bytecodes from which snippet graphs are constructed are never executed by the bytecode interpreter. Therefore, the compiler does not have profiling information for these bytecodes. Since they are used to implement many performance-critical operations, this lack of information can be a problem. For example, profiling of branches is essential to optimize the machine code layout, in order to separate frequently executed from infrequently executed code.

```

@Snippet
static int f2i(float input, int result) {
    if (probability(SLOW_PATH, result == Integer.MIN_VALUE)) {
        if (Float.isNaN(input)) {
            return 0;
        } else if (input > 0.0f) {
            return Integer.MAX_VALUE;
        }
    }
    return result;
}

```

Fig. 13. Snippet to fix special return values of the float-to-int conversion on Intel.

To solve this problem, we use node intrinsics to explicitly inject branch probabilities. An example is the first if statement of Figure 5. The probability intrinsic wraps a Boolean condition and takes an additional argument that is the probability of this condition being true. During snippet preparation, it is translated to a BranchProbabilityNode. The probability specified must be a constant as defined in Section 6.2. The compiler looks for the IfNode that takes the BranchProbabilityNode as its input, sets its profile accordingly, and finally removes the BranchProbabilityNode.

6.7. Deoptimization

An important aspect of deoptimization is selecting the bytecode index and method at which execution is restarted in the interpreter. In general, in Graal, execution restarts just after the last side-effecting instruction in the machine code (we refer to Duboscq et al. [2013b] for details about it). Therefore, side-effecting nodes need to be annotated with the metadata required by deoptimization.

It would be possible to deoptimize inside snippet code by making all node intrinsics executable in the interpreter. However, we do not allow such deoptimization in order to avoid presenting the runtime system with states it does not expect to see, for example, an activation frame boundary not being at an invoke bytecode. Instead, when lowering a high-level node that has a side effect, we transfer the metadata from the high-level node that is being lowered to the last side-effecting node of each path in the snippet. All other side-effecting nodes in the snippet are marked with special invalid metadata. This metadata is used to check that no deoptimization can be triggered when only part of the snippet's side effects were executed.

6.8. Foreign Calls

Snippets are typically used to implement the fast paths of operations. To handle executions that miss the fast path, snippets can choose either to deoptimize or to call out to some helper routine. A call to such a routine is called a *foreign call* and differs from a normal Java method call in that it is to C, C++, or assembler code; uses different conventions for passing parameters; or is to a routine with nonstandard register-saving semantics.

A foreign call is expressed in a snippet as a call to a node intrinsic for ForeignCallNode. The VM-specific code is responsible for implementing the linkage between such a call site and the routine. We do not use regular Java native methods for foreign calls because of the restrictions and overhead of the Java Native Interface (JNI). JNI requires the first parameter to be a special environment, which is superfluous when calling into the VM. In addition, VM methods often need a less expensive thread transition; for example, they often do not need to follow the full protocol to make the stack walkable for the garbage collector.

6.9. Expressing Scheduling Constraints

The Graal compiler uses floating nodes. They are not part of the control-flow graph, so all dependencies must be explicitly defined with data-flow edges. When using such nodes in snippets through node intrinsics, the snippet must declare those dependencies. For example, in Figure 5, the `loadHub` can only be executed after its input has been checked to be non-null. Since `loadHub` is a node intrinsic for a floating node (see Figure 6), we need to express its dependency to the null check.

The dependency is expressed via the anchor node intrinsic, which represents its enclosing branch. In the case of the `instanceof` snippet, we use it in the branch where the object cannot be null and we feed its return value to the `loadHub` intrinsic. As seen in Figure 6, the resulting IR graph does not contain an actual `Anchor` object value, but only a dependency expressed with an edge between the `LoadHubNode` and the beginning of the block where the object is known to be non-null. This dependency ensures the correct scheduling of the `LoadHubNode`.

6.10. Lowering of Floating Nodes

Both fixed and floating high-level nodes can be lowered using snippets. However, floating nodes need to be anchored in the context of a scheduled graph when they are lowered: if any fixed nodes are introduced by the lowering, they must be inserted into the control flow at the correct place. In the running example, the `InstanceOfNode` (which our snippet is lowering) is floating, but its snippet includes control-flow nodes (such as the `if` statements). We perform a scheduling of the graph and insert the fixed nodes into the control-flow graph where the scheduler has placed the floating node, that is, at a point that dominates all usages.

7. EVALUATION

In this section, we present details on the concrete snippets we have written as part of the lowering mechanism in Graal. These details provide insight to the memory overhead and compilation time overhead imposed by snippets. We also report performance results for standard benchmarks, to show that snippets can be successfully used in an aggressively optimizing compiler.

Graal is used in different configurations in different virtual machines, for example, the HotSpot VM and the Maxine VM. We decided to use the Graal–HotSpot integration for the evaluation over the Graal–Maxine integration as the HotSpot VM is a production-quality VM, making our results more relevant in practice.

We use the community-driven DaCapo [Blackburn et al. 2006] benchmarks (version 9.12 “Bach”) and the Scala DaCapo [Sewer et al. 2011] benchmarks (version 0.1.0 2012-02-16), as well as the standards-body-defined *SPECjbb2005*, *SPECjbb2013*, and *SPECjvm2008* benchmarks. The results shown are aggregated or averaged for each benchmark group.

The benchmarks were executed on an Intel Xeon E5-2690 v1 with eight cores, two threads per core, at 2.90GHz, 96GByte memory, running Ubuntu Linux 12.04.3 (kernel version 3.5.0). The Java heap size was fixed to 64GByte for all benchmarks and all benchmark files were in an in-memory file system. All results are based on revision 28117ede7606⁴ of Graal from April 21, 2015. It is based on the HotSpot VM of the JDK 8 update 40.

The benchmark *eclipse* from the *DaCapo* benchmarks and the benchmark *compiler.sunflow* from the *SPECjvm2008* benchmarks do not run on JDK 8. They contain

⁴<http://hg.openjdk.java.net/graal/graal/rev/28117ede7606>

	P	S	S/P	S/P _{max}
DaCapo	22.9	49.7	2.2	21
Scala DaCapo	27.4	55.5	2.0	21
SPECjbb2005	21.0	49.0	2.3	13
SPECjbb2013	31.0	76.0	2.5	22
SPECjvm2008	25.4	56.4	2.2	25

Fig. 14. Number of prepared (P) and specialized (S) snippet graphs.

bytecode parsers that do not support the latest Java bytecode specification. Therefore, these benchmarks are excluded in all measurements.

7.1. Usage Metrics

The Graal code base includes 89 snippets. These snippets are used to lower the following operations: arraycopy, type checks, synchronization, object allocation, object cloning, complex write barriers, boxing/unboxing operations, and conversion operations. Figure 14 presents basic usage statistics for these snippets. The first two columns show the average number of prepared and specialized snippet graphs created for each benchmark suite. The last two columns show the average and maximum number of specialized graphs per prepared graph. As can be seen, only about a quarter to a third of the available snippets are actually used in the benchmarks. This is due to the fact that a number of snippets differ from each other only in their parameter types. For example, there are 16 snippets for lowering the boxing and unboxing methods in the JDK (box/unbox for each of the eight primitive Java types), but most applications only use a limited number of these boxed types. Two to two and a half specializations are created per snippet. However, this factor is not evenly distributed, as shown by the last column. In the current implementation, the object allocation snippet causes the greatest expansion as it specializes on the size of the object being allocated. This specialization enables unrolling of the loop in the snippet that zeros the body of a new object.

As mentioned in Section 6, the flexibility of writing snippets in Java makes it straightforward to add instrumentation and logging source code to snippets. For complex snippets, this additional code can cause the prepared snippet graphs to be quite large. For instance, the prepared graphs for our `monitorenter` and `monitorexit` snippets (implementing Java's synchronized keyword) contain 602 and 252 nodes, respectively. Most of these nodes are correlated with tracing code in the snippets. Since all this tracing code is guarded by a `@ConstantParameter` annotated parameter, the corresponding nodes are eliminated during snippet specialization (when tracing is disabled). The resulting specialized graphs have 90 and 36 nodes, respectively. In contrast, snippets with parameters that cause loop unrolling during specialization have a larger graph after specialization. The observation is that the ratio between sizes of prepared and specialized graphs for any given snippet depends solely on the type of code in the snippet.

7.2. Compilation Speed

The impact of snippets on compile time is based on how often they are used during compilation and how much time instantiating a snippet takes. Figure 15 shows the number of compiled methods, the number of snippet instantiations, and the average number of snippet instantiations per method. The latter number is fairly stable across the benchmarks. As such, the percentage of compilation time spent in processing snippets should also be stable. Indeed, this is reflected in Figure 16(a), which shows the total

	Comp	Inst	Inst/Comp
DaCapo	10,191	34,033	3.34
Scala DaCapo	12,575	47,116	3.75
SPECjbb2005	778	3,287	4.22
SPECjbb2013	3,260	18,114	5.56
SPECjvm2008	19,962	72,312	3.62

Fig. 15. Number of snippet instantiations (Inst) relative to number of compiled methods (Comp).

	C	S	S/C (%)
DaCapo	22,742	809	3.90%
Scala DaCapo	26,830	829	3.41%
SPECjbb2005	17,973	733	4.08%
SPECjbb2013	56,034	1356	2.42%
SPECjvm2008	29,762	866	3.29%

(a) With preparation cache, with specialization cache.

	C	S	S/C (%)
DaCapo	26,100	7,447	27.76%
Scala DaCapo	33,368	9,258	28.24%
SPECjbb2005	24,598	8,195	33.32%
SPECjbb2013	70,277	20,469	29.13%
SPECjvm2008	37,553	9,150	23.54%

(b) With preparation cache, without specialization cache.

	C	S	S/C (%)
DaCapo	26,424	7,648	27.98%
Scala DaCapo	33,599	9,236	28.09%
SPECjbb2005	23,724	7,991	33.68%
SPECjbb2013	67,260	20,021	29.77%
SPECjvm2008	39,402	9,581	23.20%

(c) Without preparation cache, without specialization cache.

Fig. 16. Fraction of compilation (C) time spent on processing snippets (S). All times are in milliseconds.

compilation time and the time taken for all snippet processing (preparation, specialization, and instantiation). The latter accounts for about 3% to 4% of compilation time. The critical key to this low overhead is caching of both the prepared and specialized graphs. The importance of caching is made clear by Figure 16(b) and Figure 16(c). When caching of specialized graphs is disabled (see Figure 16(b)), the overhead of snippet processing jumps to between 27% and 34% of compilation time. Further disabling the caching of prepared graphs (see Figure 16(c)) has about the same overhead.

7.3. Performance

To demonstrate that snippets can be used in a high-performance environment, we compare Graal with the client and server compiler of the Java HotSpot VM. The client compiler translates high-level operations to machine code directly, by inserting hand-crafted assembly templates. The server compiler lowers high-level operations within its graph by replacing them with hand-assembled IR graphs. Graal uses snippets to lower high-level operations.

Fig. 17. Performance of Graal compared to other compilers of the HotSpot VM (higher is better).

This comparison does not show the relative performance of using snippets versus not using snippets, because all three compilers are based on different IRs and perform different optimizations. There is neither a version of Graal without snippets that could be used as a reasonable target for comparisons nor a version of the client or server compiler that uses snippets. We include the performance results solely to show that Graal, a dynamic compiler written in Java using snippets, can compete with production-quality dynamic compilers written in C++.

Figure 17 shows the performance results. For the DaCapo benchmarks, we report the geometric mean of all subbenchmarks. For SPECjvm2008, we report the composite result calculated according to the rules of the benchmark harness. For all benchmarks, Graal shows better performance than the client compiler and approaches the performance of the server compiler. The SPECjbb2005 benchmark has the biggest performance difference between Graal and the server compiler. This can be attributed to the extremely high effort spent on optimizing the server compiler specifically for this benchmark, since SPECjbb2005 was the main commercially used benchmark for more than a decade.

8. RELATED WORK

It is traditional to implement VMs in low-level languages such as C and C++. However, metacircular VMs such as the Klein VM [Ungar et al. 2005], the Jikes RVM [Alpern et al. 1999, 2000; Rogers and Grove 2009], and the Maxine VM [Wimmer et al. 2013] have demonstrated the feasibility of using a high-level language instead. High-level languages bring significant benefits: better modularity, memory safety, and automatic memory management. However, high-level languages were not designed for system-level programming, a shortcoming for which different solutions have been developed.

The Jikes RVM is the pioneering metacircular Java VM. Its optimizing compiler [Burke et al. 1999] inlines runtime functions that are written in Java to perform the lowering of high-level operations to a lower level. This function inlining lacks the control and expressive power of snippets and does not support the VM–compiler separation that snippets provide. For example, an inlined method cannot specify its branch probabilities at a fine-grained level (but methods can be annotated as being a slow path or fast path) and cannot interact with the compiler’s scheduler to get the optimal machine code sequence of a handcrafted assembly code, for example, because it cannot specify that a memory read can be floating and scheduled in a range of blocks. The set of compiler intrinsics that the inlined methods can use is fixed, while our node intrinsic mechanism allows snippet implementers to define new IR nodes that were not present in the compiler before. The downside of this flexibility is that snippets are compiler specific. The Jikes RVM reaches a high peak performance, but we believe

that the limitations of its compiler intrinsics approach make it impossible to reach or surpass the peak performance of, for example, the Java HotSpot server compiler.

Jikes RVM uses the `org.vmmagic` library [Frampton et al. 2009] for low-level programming in Java. It was applied to other VMs, such as the OVM [Palacz et al. 2005], Moxie VM [Blackburn et al. 2008], and DRLVM [Apache Software Foundation 2011]. Compilers supporting `org.vmmagic` replace method calls to this library with compiler intrinsics. This approach differs from Graal's node intrinsics in that `org.vmmagic` clients know nothing about the compiler or the nodes to which the calls are mapped.

As mentioned in the introduction, early versions of the Maxine VM introduced the concept of snippets in the CPS compiler, which we now refer to as *Maxine snippets*. Even though CPS never realized the goal of being a high-performance compiler, its experimentation inspired the implementation presented in this article. The primary issue with Maxine snippets was that they were compiler independent, which meant compiler internals could not be leveraged to write efficient snippets. In an attempt to compensate for this, the whole compilation process included some snippet-specific transformations. The latter not only slowed down compilation further but also reduced possibilities for tuning Maxine snippets. The snippet specialization in Graal combined with node intrinsics addresses these issues.

The Open Runtime Platform (ORP) [Cierniak et al. 2005] serves as a framework for VM development, consisting of several compilers and language front ends. Interfaces to runtime services need only be written once, avoiding code duplication. Initially, this was achieved by manually writing *stubs* in assembly language, but it meant doing so for each target architecture. The *low-level interface language* (LIL) [Glew et al. 2004] was designed to address this portability issue. It enabled stubs to be written in a target-independent way. However, this approach used a separate compiler to produce machine code from LIL, which means LIL code paths were not inlined. Glew et al. [2004] suggest a technique called *code implants* that translates LIL to the compiler's IR. This allows the compiler to inline the stub and apply further optimizations. We are not aware if *code implants* were ever implemented in ORP.

The C1X compiler of the Maxine VM started as a port of the HotSpot client compiler [Kotzmann et al. 2008] to Java. A significant new contribution of C1X is a clear separation between the VM and the compiler. XIR [Titze et al. 2010] enables the runtime to specify a high-level operation in a domain-specific language. XIR is close to assembly language and only exists as data structures inside the VM. When translating certain high-level IR nodes to low-level IR nodes, C1X asks the VM for the XIR expressing the semantics of the high-level node. The VM generates this XIR on demand, caching it for subsequent similar requests, much like Graal caches specialized snippet graphs. C1X then converts XIR to its low-level IR. This improves upon LIL in that the XIR code paths are inlined into the generated machine code. However, given that this stage happens after all high-level optimizations have been applied, there are no opportunities for the XIR to be further optimized. For example, two consecutive type checks on the same object do not share the generated code for loading the class pointer from the object. This is not an issue for snippets since they are inserted into the high-level IR of a method being compiled and thus are subject to optimizations such as common subexpression elimination.

PyPy [Rigo and Pedroni 2006] is a framework for implementing interpreters with a focus on dynamic languages. The framework is able to automatically generate a trace-based compiler from an interpreter. RPython [Ancona et al. 2007] is a restricted dialect of Python that is used to implement an interpreter in PyPy. RPython can be executed by standard Python interpreters. Such execution is slow; therefore, a translation process is necessary to achieve high performance. This process builds an interpreter binary along with its trace-based compiler for a specific target platform such as C or Java

bytecode. During the translation, the program is represented as a flow graph. The translation process consists of several transformation steps that modify the graph in place. *System code* functions are used to express the lowering of a high-level operation, for example, object allocation. System code functions are written in RPython and must be parsed by the front end before inserting them into the graph. According to Rigo and Pedroni [2006], the translation process for the Python interpreter prototype takes about 39 minutes, 10 minutes of which is accounted for by the lowering transformation. We believe that preparation of system code functions, as presented in this article for snippets, could improve the translation process. However, that would solely speed up the lowering step, which is only needed after a source code modification to the interpreter.

9. CONCLUSIONS

We presented snippets, a technique for expressing the semantics of high-level operations in Java in such a way that Graal can use them for lowering high-level nodes to low-level nodes during compilation. The techniques presented enable Graal to leverage all the benefits of snippets without any negative impact on compilation time or the quality of generated machine code. Snippets are written in Java code in a style that is intuitive and easy to understand. They retain and even increase the simplicity and beauty of writing runtime functionality in Java. But more importantly, they allow Graal to reach the peak performance of “traditional” compilers like the Java HotSpot server compiler.

Source code for the Graal compiler is available at the OpenJDK Graal Project site [OpenJDK 2013].

ACKNOWLEDGMENTS

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and contributions. We especially thank Mario Wolczko and Michael Van De Vanter for feedback on this article.

The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

REFERENCES

- B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.
- Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Janice C. Shepherd, and Stephen Smith. 1999. Implementing Jalapeño in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 314–324. DOI : <http://dx.doi.org/10.1145/320384.320418>
- Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the Dynamic Languages Symposium*. ACM, 53–64. DOI : <http://dx.doi.org/10.1145/1297081.1297091>
- Apache Software Foundation. 2011. Dynamic Runtime Layer Virtual Machine. Retrieved from <http://harmony.apache.org/subcomponents/drlvm/>.
- S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 169–190. DOI : <http://dx.doi.org/10.1145/1167473.1167488>

- Stephen M. Blackburn, Sergey I. Salishev, Mikhail Danilov, Oleg A. Mokhovikov, Anton A. Nashatyrev, Peter A. Novodvorsky, Vadim I. Bogdanov, Xiao Feng Li, and Dennis Ushakov. 2008. *The Moxie JVM Experience*. Technical Report TR-CS-08-01. Department of Computer Science, Australian National University.
- Michael G. Burke, Jong-Deok Choi, Stephen J. Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. 1999. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Conference on Java Grande*. ACM, 129–141. DOI: <http://dx.doi.org/10.1145/304065.304113>
- Michal Cierniak, Marsha Eng, Neal Glew, Brian T. Lewis, and James M. Stichnoth. 2005. The open runtime platform: A flexible high-performance managed runtime environment. *Concurrency and Computation: Practice and Experience* 17, 5–6 (2005), 617–637. DOI: <http://dx.doi.org/10.1002/cpe.852>
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (1991), 451–490. DOI: <http://dx.doi.org/10.1145/115372.115320>
- Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013a. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.
- Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013b. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*. ACM, 1–10. DOI: <http://dx.doi.org/10.1145/2542142.2542143>
- Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. 2009. Demystifying magic: High-level low-level programming. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM, 81–90. DOI: <http://dx.doi.org/10.1145/1508293.1508305>
- Neal Glew, Spyridon Triantafyllis, Michał Clerniak, Marsha Eng, Brian Lewis, and James Stichnoth. 2004. LIL: An architecture-neutral language for virtual-machine stubs. In *Proceedings of the Virtual Machine Research and Technology Symposium*. USENIX Association, 111–125.
- Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 32–43. DOI: <http://dx.doi.org/10.1145/143095.143114>
- Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization* 5, 1 (2008), Article 7. DOI: <http://dx.doi.org/10.1145/1369396.1370017>
- Bernd Mathiske. 2008. Systems Programming in the Maxine VM: How to Enable It and How to Get Around It. Retrieved from <http://www.agentgroup.unimo.it/ppj08/invtalk1.html>.
- OpenJDK 2013. Graal Project. Retrieved from <http://openjdk.java.net/projects/graal>.
- Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. 2005. Engineering a common intermediate representation for the OVM framework. *Science of Computer Programming* 57, 3 (2005), 357–378. DOI: <http://dx.doi.org/10.1016/j.scico.2004.08.007>
- Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*. USENIX, 1–12.
- Armin Rigo and Samuele Pedroni. 2006. PyPy’s approach to virtual machine construction. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*. ACM, 944–953. DOI: <http://dx.doi.org/10.1145/1176617.1176753>
- Ian Rogers and Dave Grove. 2009. The strength of metacircular virtual machines: Jikes RVM. In *Beautiful Architecture*, Diomidis Spinellis and Georgios Gousios (Eds.). O’Reilly, Chapter 10.
- Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 657–676. DOI: <http://dx.doi.org/10.1145/2048066.2048118>
- Doug Simon. 2009. The Maxine Project: Snippets in the Maxine VM. Retrieved from <https://wikis.oracle.com/display/MaxineVM/Snippets>.
- Ben L. Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra. 2010. Improving compiler-runtime separation with XIR. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM, 39–50. DOI: <http://dx.doi.org/10.1145/1735997.1736005>
- David Ungar, Adam Spitz, and Alex Ausch. 2005. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*. ACM, 11–20. DOI: <http://dx.doi.org/10.1145/1094855.1094865>

Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization* 9, 4 (2013), Article 30, 24 pages. DOI:<http://dx.doi.org/10.1145/2400682.2400689>

Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2008. Visualization of program dependence graphs. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, 193–196. DOI:http://dx.doi.org/10.1007/978-3-540-78791-4_13

Received November 2014; revised March 2015; accepted April 2015