JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

informatik
Schnittstelle Zukunft

Tail Call Optimization in the Java HotSpot™ VM

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

MASTERSTUDIUM INFORMATIK

Eingereicht von:
Arnold Schwaighofer, 9920231

Angefertigt am:
Institut für Systemsoftware

Betreuung:
o.Univ.-Prof. Dipl.-Ing. Dr. Dr. h.c. Hanspeter Mössenböck

Mitbetreuung:
Dipl.-Ing. Dr. Christian Wimmer

Linz, März 2009

Abstract

Many programming language implementations compile to Java bytecode, which is executed by a virtual machine (e.g the Java HotSpot$^{\text{TM}}$ VM). Among these languages are functional languages, which require an optimization that guarantees that certain kinds of method calls do not cause the execution stack to grow unlimitedly. This optimization is called *tail call optimization* and is currently not supported by the HotSpot$^{\text{TM}}$ VM. Implementations of functional languages have to resort to alternative techniques to guarantee that the stack space does not increase unboundedly. These techniques complicate the implementation and also incur a performance penalty.

This thesis presents techniques for supporting tail call optimization in the Java HotSpot$^{\text{TM}}$ Virtual Machine. Our optimization is implemented in the interpreter, the client compiler and the server compiler. Tail call optimization normally removes stack frames to guarantee that the stack space stays bounded. However, some stack frames are required for the Java access security mechanism to work and hence cannot be removed. The virtual machine features a mechanism called *deoptimization* that allows one to rewrite stack frames. We describe an approach that uses the deoptimization infrastructure to compress stack frames when tail call optimization was disabled because of the security mechanism. This approach allows a series of tail calls to execute in bounded stack space in the presence of a stack-based security mechanism.

Kurzfassung

Viele Programmiersprachen Implementierungen kompilieren zu Java Bytecodes, welche von der Java HotSpot$^{\text{TM}}$ Virtual Machine ausgeführt werden. Unter diesen sind auch funktionale Programmiersprachen. Einige dieser Sprachen erfordern eine Optimierung die sicher stellt, dass gewisse Methodenaufrufe den Methodenkeller nicht uneingeschränkt erweitern. Diese Optimierung wird Tail Call Optimierung genannt und wird derzeit von der VM nicht unterstützt. Implementierungen müssen deshalb alternative Techniken verwenden, welche garantieren, dass der Methodenkeller eingeschränkt bleibt. Diese Techniken komplizieren die Implementierung und verursachen einen Geschwindigkeitsverlust.

Diese Arbeit zeigt die Veränderungen die notwendig sind, um Tail Call Optimierung in der Java HotSpot$^{\text{TM}}$ Virtual Machine zu unterstützen. Die Optimierung wurde im Interpreter, Client und Server Compiler implementiert. Tail Call Optimierung entfernt normalerweise Methodenaktivierungssätze. Manche Aktivierungssätze werden für ein korrektes Arbeiten des Java Sicherheitsmechanismus benötigt und können nicht entfernt werden. Die virtuelle Maschine stellt einen Mechanismus - Deoptimierung - zur Verfügung, der es erlaubt Aktivierungssätze zu verändern. In dieser Arbeit wird beschrieben, wie Deoptimierung genutzt werden kann um Methodenaktivierungssätze zu komprimieren, falls Tail Call Optimierung aufgrund des Sicherheitsmechanismus nicht angewendet wurde. Diese Vorgehensweise garantiert, dass eine Serie von Methodenaufrufen in Tail Call Position trotz des Java Sicherheitsmechanismus in beschränktem Methodenkellerspeicher ausgeführt werden kann.

Contents

1 Introduction

Today there are many programming language implementations that rely on the Java Virtual Machine (JVM) as their target platform. The JVM provides them with many features that are required for a modern implementation. One feature, which is currently missing from the JVM platform, is tail call optimization. Tail call optimization is applied to certain kind of function calls allowing a series of such calls to execute in bounded stack space. This optimization is required for implementing many functional languages. Its absence complicates implementing function calls and the workarounds used cause a performance penalty, when compared to normal function calls.

This thesis describes the implementation of tail call optimization in the Java HotSpot$^{\text{TM}}$ Virtual Machine. This chapter introduces the JVM as a target platform for language implementations, describes the context of this work and lists the challenges that arise from implementing tail call optimization on a JVM.

1.1 Target for Programming Language Implementations

> “The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the class file format. A class file contains Java virtual machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.
>
> For the sake of security, the Java virtual machine imposes strong format and structural constraints on the code in a class file. However, any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine. Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java virtual machine as a delivery vehicle for their languages.“

The Java Virtual Machine Specification, [38]

A modern programming language implementation requires many aspects to be taken into account. When implementing a new language the designer not only has to take care that the language itself is sound and correct but he also needs to ensure that the language implementation runs on a variety of operating systems and hardware platforms. He needs to deal with memory management and concurrent execution of programs. Getting this features correct takes a considerable amount of engineering effort.

This is where a virtual machine can help. The idea of a virtual machine is to introduce an abstraction level between generated code and target platform. This abstraction level hides the target platform by supplying a standardized platform independent interface to perform computations. Very often it also provides abstractions for memory management and concurrent execution. The user of the virtual machine programs against the defined interface. The VM provides an environment that ensures that programs execute correctly on the target platform.

A Java Virtual Machine is such an environment. It has a defined interface in the form of an instruction set called Java bytecode. Instead of translating the language to object code of a target platform, a compiler targeting the JVM translates programs to Java bytecode. There are implementations of the JVM on a variety of hardware and operating system platforms. When the program is to be executed the JVM loads the bytecode and performs actions that correspond to the semantic of the bytecode.

As the name indicates the JVM was initially meant to be the target of the Java programming language. The Java language is a general purpose object oriented concurrent language [38]. Hence Java bytecode provides instructions for general purpose computing, e.g. mathematical and logical operations, for concurrent execution, e.g. locking and unlocking of objects, and for allocating and manipulating objects. As these features are also required for other languages, implementors have soon realized that the Java VM can also be used as a target for their compilers, thereby getting automatic memory management, platform independence and concurrent execution support. In addition they can benefit from the wealth of existing libraries. Examples of language implementations compiling to the JVM are

- Groovy [32]: a dynamically typed object oriented language.
- JRuby [42]: an implementation of the programming language Ruby.
- Jython [30]: an implementation of the programming language Python.
- Bigloo [52], Kawa [7], SISC [40]: Scheme implementations.
- Clojure [26], Armed Bear Common Lisp [29]: Common Lisp implementations.
- Scala [43]: a programming language that unifies functional and object oriented programming.
- OcamlJava [13]: an implementation of the programming language OCaml, which is a variant of ML.

1.2 Context

A JVM conforms to an abstract specification, the JVM specification [38]. The Specification defines the bytecode, the object model and the execution model. The JVM executes by loading class files, which contain Java bytecode. Before execution the code is verified to ensure that the VM is not compromised. Java bytecode mimics a stack machine. Operands are pushed onto an operand stack. An operator, e.g. an add instruction or a method call, consumes them and leaves its result on the operand stack. Bytecodes are assembled in methods. Each active method has a stack frame associated with it. It contains its execution environment consisting of local variables, the method's parameters and the operand stack. A recursive call of a method causes creation of a new frame on top of the calling functions frame, thereby growing the execution stack. Objects are allocated on a heap and are automatically reclaimed by a garbage collector when they are no longer referenced. To support concurrent execution a thread model with shared memory is used. Each thread has its own execution stack and performs computation concurrently with other threads. Threads share the heap. Therefore the programmer has to guard concurrent access to memory using special instructions that guarantee mutual exclusion.

The JVM features a security model where code loaded from certain locations can be restricted in what it can do. Potentially dangerous operations, e.g. reading or writing to a file, require certain permissions. On execution of such an operation the JVM checks if the operation is allowed by looking at the execution stack. Each frame maps to a set of permissions via the method it belongs to. If the JVM detects a frame of a method that does not have the permission to perform the operation, the operation fails and a security exception is thrown.

The Java HotSpot™ VM is an implementation of the specification. It is a program that consists of an interpreter and a machine code compiler. At first bytecode is interpreted. When the execution count of a method crosses a certain threshold, the method is compiled as a whole by one of two compilers: the client compiler and the server compiler. The compilers produce optimized code for the target platform of the JVM. This code runs at a much higher speed than an execution of the same method in the interpreter. The execution stack maintains two types of frames, compiled frames containing the execution environment for compiled methods and interpreter frames containing the execution environment for interpreted methods.

1.3 Tail Call Optimization

In many functional languages, iterative computations are expressed by recursive method calls. To describe the iterative process, the last action a method performs is calling another method before returning the result of the called method. This is called the tail call. Without tail call optimization each recursive method call creates a new stack frame thereby growing the execution stack. Eventually the stack runs out of space and the program has to stop. Tail call optimization guarantees that a series of tail calls executes in bounded space not causing the program to stop. It is implemented by replacing the caller method's frame by the called method's frame. To support iteration by recursion functional languages require tail call optimization.

1.4 Problem Statement

To be able to support functional languages better, the goal of this thesis is to add a tail call instruction to the instruction set and implement tail call optimization in the Java HotSpot™ Virtual Machine. The following challenges are solved in the IA-32 version of the VM.

- We added support for tail call versions of the different method invocation bytecode instructions. The programmer indicates a tail call to the VM by prefixing the invocation bytecode with a special bytecode. We modified the VM, so that it recognizes these instructions as tail call versions of a method call.
- The bytecode verifier recognizes tail calls and checks the conditions under which they are legal. Programs that contain tail calls at places where they violate correct program execution are rejected.
- We modified the interpreter to perform tail call optimization on tail call method invocations.
- We modified the client and server compiler to support tail call optimization. To deal with the different types of tail calls we introduce special method prologs.
- The Java HotSpot™ VM features mixed execution stacks. An interpreted method's frame might be followed by a compiled method's frame and vice versa. The tail call optimization implementation maintains correct behavior in this environment.
- Tail call optimization sometimes requires that a stack frame grows to accommodate additional method parameters. In the Java HotSpot™ VM compiled frames have a fixed size and cannot grow. The implementation solves this by detecting when a frame needs to grow and using an interpreter frame in this case.

- The security model requires information stored in the stack frame of called methods. If this information differs from calling to called method in a tail call, replacing the calling frame by the called frame might destroy information and change the security behavior. In this case the implementation disables the tail call and leaves the stack frame on the stack. This might result in a stack overflow exception. To maintain the tail call optimization guarantee, the implementation compresses the execution stack by removing stack frames that contain duplicate information before the stack overflow occurs.

1.5 Structure of the Thesis

Chapter 2 describes the motivation and theory behind tail call optimization and lists alternative methods, which are used by language implementations on the JVM to circumvent the absence of tail call optimization. At the end a definition of tail call optimization is given, which is used for the rest of the thesis.

Chapter 3 describes the Java HotSpot™ VM as the target of the implementation. The optimization is implemented in the interpreter, client compiler and server compiler for the IA-32 platform. The runtime of the VM is adapted. This chapter describes these components.

Chapter 4 describes the implementation of tail call optimization in the VM. The changes to the bytecode instruction set and the bytecode verifier are described. An overview of the different method invocations and the method abstraction is given. Tail call optimization interacts with the Java access control security mechanism. We describe how the implementation maintains the security semantics. Next the changes to the interpreter are explained, followed by the adaptations in the compiler and the runtime. At the end of the chapter two implemented improvements are described, which enhance the performance of the implementation.

Chapter 5 evaluates the implementation using a program that mainly consists of tail calls. The performance of tail call optimized code is compared to normal method calls and a trampolined version of the program. This illustrates the worst slowdown or the best speedup that is to be expected in real programs when tail call optimization is enabled.

Chapter 6 lists related projects and compares them to this work. Finally chapter 7 summarizes the work and suggests some improvements to the current implementation.

2 Tail Call Optimization

Execution of methods causes the creation of new stack frames. The execution stack grows. Tail call optimization prevents the stack from growing by replacing the caller's frame by the called method's frame. This can only be done safely under certain conditions. This chapter introduces the theory behind tail call optimization. It starts with an example, which is used throughout this thesis. Using an abstract description of method execution a normal and a tail call optimized calling sequence is illustrated and the conditions under which the optimization is valid are described.

Tail call optimization can be used to improve performance but functional languages require it to express iterative computation. We show some typical control constructs in functional languages as motivation for the need of tail call optimization. Then we show approaches used by language implementations on the Java VM that require it. In the context of the Java VM the caller's stack frame cannot always be replaced due to the access security mechanism. At the end of this chapter we give a less strict definition of tail call optimization used for rest of this thesis.

2.1 Example

For the purpose of illustrating what happens during tail call optimization the following Java example of calculating the length of a list is used throughout this thesis. The list is described as an abstract datatype `List`. A `List` can either be an empty list `Empty` or an element `ListElem`, which is prepended to a list. The `ListElem` object has two fields. The field `element`, which holds the element of the list and the field `rest`, which holds the remaining part of the list.

To construct a list, `ListElem` constructors are nested appropriately, as shown in the `main` method of `List`. The `length` method calls the `accLen` method passing zero as argument. The `accLen` method takes one parameter `n`, the length computed so far. If `accLen` is called on a `ListElem` object, it adds one to the length computed so far and passes this value to the `rest`'s method `accLen`, which recursively computes the length of the rest of the list. The `Empty`'s method `accLen` returns the parameter passed to it. It builds the base case of the recursion, which causes the return of the result to the length function.

```java
public abstract class List {
  public int length() {
    return accLen(0);
  }
  abstract protected int accLen(int n);
  static void main(String args[]) {
    List l = new ListElem(1, new ListElem(2, new Empty()));
    int len = l.length(); // == 2
  }
}

public class ListElem extends List {
  private Object element;
  private List rest;
  public ListElem(Object element, List rest) {
    this.element = element;
    this.rest = rest;
  }
  protected int accLen(int n) {
    int newLength = n + 1;
    return rest.accLen(newLength);
  }
}

public class Empty extends List {
  public Empty() {}
  protected int accLen(int n) {
    return n;
  }
}
```

Listing 2.1: `List` code example

2.2 Normal Method Call Sequence

An active method uses a data structure called *stack frame* to store information needed for its execution. In general it contains at least the following four items.

- Local data, an area where the function stores its local variables or temporary values such as spilled registers.
- Dynamic link, a reference to the frame of the method that called the current method.

- Return address, the instruction in the calling function where execution resumes when the current method finishes.
- The parameters of the current method.

While there are other ways to link stack frames of calling methods together, most language implementations use a stack-like data structure, as this is an efficient structure due to fast allocation, memory locality and the fact that today's hardware platforms are optimized for calling sequences that use stack-like operations.

Each recursive call creates a new frame on top of the calling methods frame. Returning from a method to its calling method removes this frame. This results in a stack-like data structure. Hence the name stack frame. The currently executing method's frame is the top frame on this stack. When the function finishes its stack frame is conceptually *popped off* the stack.

To indicate the extend of a frame there are two pointers: the *stack pointer*, which marks the top of the stack and the *frame pointer*, which usually points to a fixed position near the bottom of the stack frame. The frame pointer can be used to access parameters and local variables, which are positioned at a fixed offset to it. It is only needed if the size of a stack frame changes during execution of a method. Otherwise it can be computed by subtracting a constant number, the stack frame size, off the stack pointer. Figure 2.1 shows a typical stack layout at a state where a method `f` called `g`.

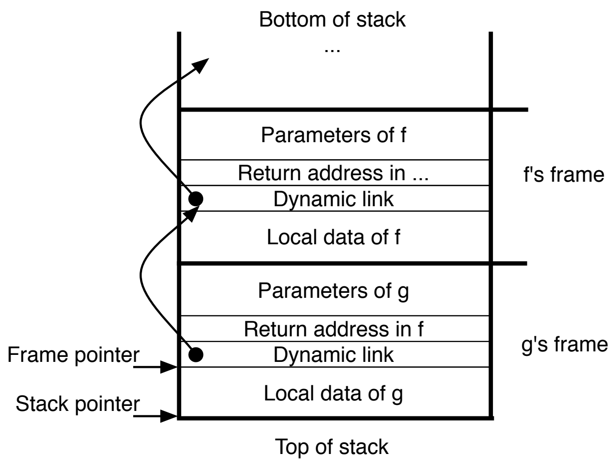


Figure 2.1: Stack layout

There are three distinct phases involved in a function call sequence. The *prolog* is executed at the entry of a method and is responsible for setting up the method's stack frame and storing state of the calling method that is destroyed by the called method (e.g. registers). The *call site* places the parameters, stores the return address on the top of the stack and jumps to the called method's entry. When the method finishes it executes the *epilog*, which invokes instructions that restore the calling method's state and tear down the stack frame.

Parameters of methods are usually passed in machine registers and on the stack depending on the calling convention used and machine registers available. For the rest of this chapter we assume for simplicity that all arguments are passed on the stack. The area where parameters are stored when calling a method is referred to as *outgoing argument area*. This area is either implicitly created by pushing the methods arguments onto the top of the stack or it is explicitly created as part of the stack frame. A compiler knows all called functions and can create an area at the end of a stack frame, which has enough space for the parameters of the called functions. Figure 2.2 (a) illustrates this. When a method is called its parameters are moved to the outgoing argument area.

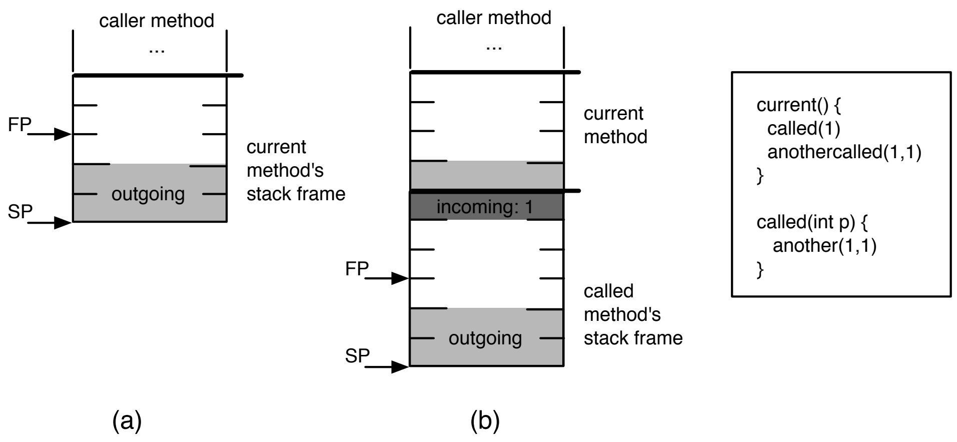


Figure 2.2: Incoming and outgoing argument area

The section of the outgoing argument area of the caller that contains the arguments of the currently called function is referred to as the *incoming argument area* of the called function. See Figure 2.2 (b) for an example. The Java HotSpot^{TM} VM creates the outgoing argument area as part of the stack frame setup. Therefore we assume this method for the rest of this thesis.

Figure 2.3 shows the state of the stack during the three phases when calling `accLen` from `length` of Listing 2.1. The leftmost figure shows the state before `length` performs the call. `SP`/`FP` denote the stack/frame pointer. There is the return address and the stored frame pointer of `length`'s calling function on the stack. The area between stack and frame pointer normally contains space for local variables and outgoing parameters. Because `length` has no local variables only one stack slot is reserved for the argument to `accLen`.

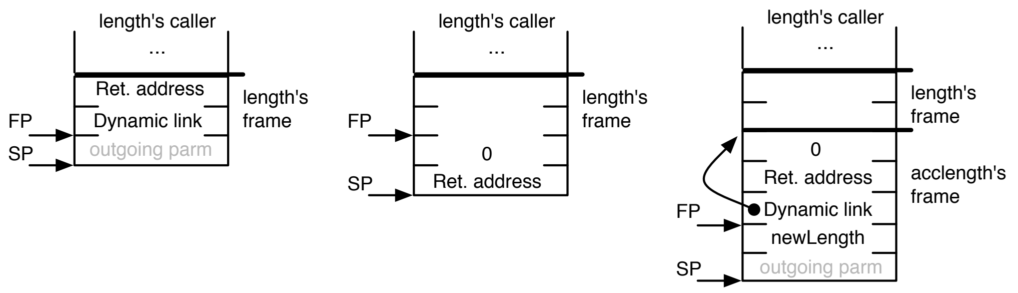


Figure 2.3: Stack states during a call sequence

The call site stores the parameter of the call to `accLen` on the stack. Next it places the address of the next instruction in `length` after the call to `accLen` on the top of stack. Then it continues execution of the program at the method entry of `accLen`. The stack's state is shown in the middle figure.

The prolog of `accLen` stores the frame pointer of `length` and increases the stack pointer to make room for the local variable `newLength` and outgoing parameters. The figure on the right depicts the stack at this stage. After the recursive call of `accLen` returns, the epilog is executed. It tears down the `accLen`'s stack frame by setting the stack pointer equal to the frame pointer. Then it pops the frame pointer of `length` off the stack. At last it retrieves the return address in `length` from the top of stack and continues execution at the pointed to instruction. The state again resembles the figure on the left. We can observe that the stack grows with each recursive method invocation.

2.3 Tail Call Conditions

The example in Listing 2.1 has one problem. If the list constructed in the `main` method is long, e.g. thousands of elements, thousands of recursive calls are performed. This causes the execution stack to grow until it eventually runs out of space resulting in a memory error or as it is the case in Java in an exception. This where tail call optimization can help. A tail call optimized method invocation safes stack space by replacing the calling frame by the called frame instead of creating a new stack frame. But this optimization can only be done if the information in the calling frame is no longer needed. The information in the calling frame is needed, if the callee (the called method) accesses memory stored in the local data section of the calling frame or if the caller performs any instructions after the called method returned besides returning itself.

Definition A call from method `f` to `g` is a *tail call* if this call is the last instruction (in the series of instructions of `f`) before `f` returns.

In the context of Java above condition is satisfied if the following conditions hold.

- `f` immediately returns the result of `g` after it was invoked or it immediately returns without a value.

```
int f() {
  return g();
}
void f() {
   g();
   return;
}
```

- The call from `f` to `g` is not surrounded by an exception handler or by a `synchronized` block as shown below.

```
int f() {
  try {
    return g();
  } catch(Exception e) {}
}
int f() {
  synchronized(obj){
    return g();
  }
}
```

- The method `f` is not marked as `synchronized`

```
synchronized int f() {
  return g();
}
```

A `synchronized` statement around the method call causes the insertion of a `monitorexit` instruction between the invoke of `g` and the return instruction thereby disabling the tail call. If the method call is surrounded by an exception handler then removing `f`'s frame removes the information about how `f` handles an exception that `g` might throw. Conceptually `f` has to look at whether an exception has occurred after `g` returns. So `f` no longer immediately returns after the call to `g`. The same principle holds for `synchronized` methods. Before the method `f` returns, it has to release the locked object. Hence there is an instruction between the call to `g` and the return, disabling the tail call.

The Java language passes all parameters by value. The only possible references to memory are pointers to objects that are allocated on the heap. The callee method can not access memory in the caller thus this condition is always satisfied.

2.4 Tail Call Optimized Method Sequence

If we want our example to succeed on arbitrary long lists, the recursive call in `accLen` to `rest.accLen` must be tail call optimized. The method call sequence then executes in constant stack space instead of growing the stack with every recursive method call.

Tail call optimization of a call from method `f` to `g` is implemented by executing the following steps.

- Moving the parameters of `g` onto the place of `f`'s parameters, so that it looks like `f`'s caller called `g` directly. Instead of moving `g`'s parameters to `f`'s outgoing argument area, they are moved to `f`'s incoming argument area, i.e. `f`'s caller's outgoing argument area.
- Removing the stack frame of `f` from the top of the stack.
- Jumping to the method entry of `g`.

Figure 2.4 shows those steps during the call of `rest.accLen` in our example. The figure on the left shows the stack before the tail call to `rest.accLen`. The local variable `newLength` has the value one. The `length` method passed zero as parameter to the current method.

The tail call proceeds by putting the parameter to `rest.accLen` on the proper stack slot just below `length`'s frame, i.e. on `length`'s outgoing argument area. Next it pops the stack by setting the stack pointer to the current value of the frame pointer. Then it pops the frame pointer of `length` off the stack. Figure 2.4 in the middle shows the state of the stack at this point. It resembles a state as if `length` directly called `rest.accLen` with a parameter of one.

In a third step the tail call jumps to the method entry of `rest.accLen`. It's prolog stores the frame pointer and creates the area for the local data. This state is shown in the right figure. The stack frame of the initial `accLen` method is replaced by the stack frame of the called `rest.accLen` method. Further recursive tail calls of `accLen` reuse the same frame. The requirement that the execution stack must not grow unlimited is satisfied.

2.5 Motivation

Tail call optimization can help improve performance over a normal method call. If caller and callee method require the same frame size, the compiler can omit the creation of a new stack frame thereby saving some method call overhead. The optimization reuses

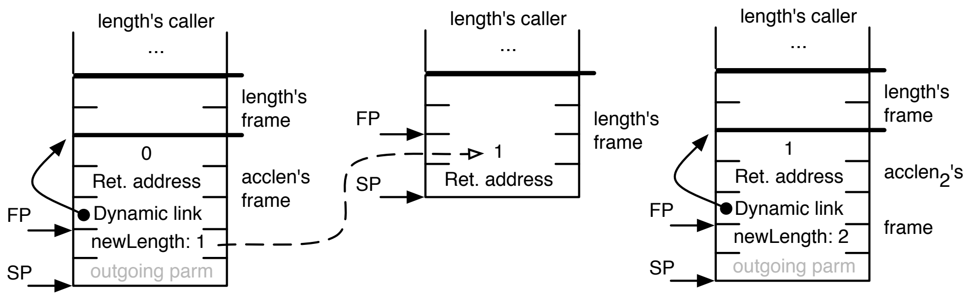


Figure 2.4: Stack states during a tail call sequence

stack space. The memory required to perform a series of tail calls remains constant. This can lead to better memory locality of the program again improving performance. But the main motivation for this work is not improving performance but to gain expressiveness by introducing a tail call optimized call to the JVM. Languages such as Scheme require tail call optimization to be performed in order to be able to express iterative computations [53]. Scheme features no native looping constructs such as a `while` or `for` loop. In order to express a loop, Scheme programmers use a recursive tail call. Listing 2.2 shows how the factorial function is calculated using a `for` loop and how a scheme programmer writes the same function using a tail recursive call.

```
int fact(int n) {
  int prod =1;
  for (int curr=1; curr <= n; curr++)
    prod = prod * curr;
  return prod;
}

int scheme_fact(n) {
  return scheme_fact_helper(n, 2, 1);
}

int scheme_fact_helper(int n, int curr, int prod) {
  if (curr > n)
    return prod;
  else
    return scheme_fact_helper(n, curr+1,  prod * curr);
}
```

Listing 2.2: Factorial function

But we not only gain the ability to express direct loops using recursive method calls by supporting tail calls. As Steele states in [24] a tail call may also be viewed as a “goto” with parameters. Having such a construct allows us to define arbitrary control structures using method calls. In the following we show two examples of such structures.

With the guarantee that a method call acts like a “goto” statement (not accumulating stack space) we can express the transitions of a state machine using method calls. Listing 2.3 illustrates such a state machine with a traffic light, which has three states represented by instances of Subclasses of the `Light` class. The `Light` class has one abstract method called `transitionTo`, which represents the transition to the state. The Subclasses of `Light` implement this method. The code is shown only for the `RedLight` class. The method executes some state specific behavior (signalling of the current color). The last instruction executed is the transition to the next state (the next color). This call has to be tail call optimized or the stack overflows after a certain number of transitions.

```
public class RedLight extends Light {
  Light next;

  public void transitionTo() {
    signalRed();
    waitForSeveralSec();
    unsignalRed();
    next.transitionTo();
  }
}

main(){
  Ligth startState = new RedLight();
  startState.next = new GreenLight();
  startState.next.next = new YellowLight();
  startState.next.next.next = startState;
  startState.transitionTo();
}
```

Listing 2.3: State machine example

Another example frequently cited where the optimization is useful, is for writing interpreters. One way to implement an interpreter is to have an evaluation function, which recursively evaluates expressions and their subexpressions. Within an interpreter it is desirable that the execution of language constructs that do not extend the execution environment in the semantics of the language does not extend the execution environment of the interpreter. Listing 2.4 shows part of such an interpreter. Similar to the example before, if the recursive calls to `evaluate` are not tail call optimized the, interpreter is in danger of running out of memory on complex expressions.

```
public class Evaluator {
  Evaluator ifEvaluator = new IfEvaluator();

  Number evaluate(Expression exp) {
    if (exp.startsWithIF())
      return ifEvaluator.evaluate(env, exp);
    else if (exp.startsWith...
  }
}

public class IfEvaluator extends Evaluator {
  Number evaluate(Expression exp) {
    if (super.evaluate(exp.getIfPredicate()))
      return super.evaluate(exp.getConsequent());
    else
      return super.evaluate(exp.getAlternative());
  }
}
```

Listing 2.4: Interpreter example

2.6 Approaches in Uncooperative Environments

As tail call optimization is required for functional languages, implementors on the JVM have to work around the fact that there is no native tail call support. There are various methods of how a tail call optimization can be simulated in an environment that does not support it. Historically those techniques where developed in the context of C compilers, which served as backends for functional language implementations. See for example [55], which shows how to emulate tail calls using the trampoline technique in a ML to C compiler.

The first technique is to only handle self recursive tail calls like the call in Listing 2.2. The content of the function is transformed, so that the recursive tail call is replaced by a jump to the beginning of the function, after the parameters have been assigned their new value. The result of this transformation is shown in Listing 2.5. This transformation is called tail recursion elimination [41] and is implemented in the following compilers which target Java bytecode: the Scala compiler, a Standard ML to Java compiler [6], OcamlJava and the Kawa Scheme compiler. Note that this method does not support general tail call optimization.

Another technique is to compile the whole program (or the relevant parts that contain tail calls) into one function and simulate tail calls by jumps to parts of it. This can be seen as a generalization of the above method. This method supports general tail call optimization, provided the target methods are all known at compile time. In a dynamic environment such as the JVM, where classes may be loaded into a program after it has been compiled, this method is not feasible. The 64 Kilobyte size limit of a method further limits the use of this method on the JVM. To the best of our knowledge there are no implementations on the JVM that use this method.

```
 int scheme_fact_helper(int n, int curr, int prod) {
 startlabel:
  if (curr > n)
    return prod;
  else {
    curr=curr+1;
    prod = prod * curr;
    goto startlabel;
  }
}
```

Listing 2.5: Tail recursion elimination example

The trampoline technique allows to express general tail calls. A *trampoline* is a piece of code, which repeatedly calls an inner function. If the inner function wishes to do a tail call it simply returns an object (called `Continuation` in the example below) containing information at which function to resume execution and which arguments to apply. By returning to the trampoline instead of creating a new stack frame, the stack does not grow unlimited. Listing 2.6 illustrates how this could be implemented for the factorial function. The trampoline is in the `factorial` function in form of the `while` loop, which repeatedly applies the continuation. As a result of the application a new `Continuation` is received, which again is applied. This process is repeated until an instance of the `ResultContinuation` is received, which contains the result of the computation. The `FactContinuation` stores the arguments to the computation and implements the computation in form of the `apply` method. Instead of the tail call a new `FactContinuation` object is returned.

The Kawa Scheme compiler gives the user the option to turn on this method. But due to the performance overhead this option is off by default.

Another method which is a variant of the method above is to only occasionally compress the stack. Instead of shrinking the stack on every tail call, this is only done every so often to prevent the stack from overflowing. In the context of the JVM the Funnel [50] compiler is the only one to use this technique. Listing 2.7 shows how this could be implemented for the factorial example. The trampoline in the `factorial` function is extended so it catches the continuation in form of an exception. The `apply` function is modified, so that it returns a continuation only if the tail call depth, the number of sequent calls to the `apply` function, exceeds a limit. Otherwise the `apply` method is directly called.

The disadvantage of the last two methods is that they incur a considerable performance overhead—upto 15% slower [50]—and they complicate the implementation effort. This increases the motivation to implement native tail calls on the JVM.

```
int factorial(int n) {
  Continuation c = new FactContinuation(n, 2, 1);
  do {
    c = c.apply();
  } while (! c instanceof ResultContinuation);
  return c.result;
}

class FactContinuation {
  int n; int curr, int prod;

  FactContinuation(int n, int curr, int prod);

  FactContinuation apply() {
    if (curr > n) {
      return new ResultContinuation(prod);
    } else {
      return new FactContinuation(n, curr+1, prod*curr);
    }
  }
}
```

Listing 2.6: Using a trampoline to achieve tail call optimization

2.7 Definition

The primary goal of tail call optimization, at least when it is required in functional languages, is to enable the programmer to express iterative computation in constant space with a recursive method call. In the context of languages like Scheme one can give a syntactic definition of what calls are tail calls (essentially those calls that are in a position where they are the last instruction before the method returns). Tail call optimization can then formally be defined using an abstract machine as it is done in [16]. The essence of the description is that for a machine that uses stack-like stack frames, tail call optimization replaces the calling methods frame with the called methods frame. The goal to stay within bounded space is achieved.

For Java bytecode the recognition of tail calls is also quite simple. A call is a tail call if the method call instruction (`invokestatic`, `invokevirtual`, `invokeinterface`) is immediately followed by one of the return instructions (`ireturn`, etc) and the method containing the tail call is not synchronized. There must be no exception handler installed for the call instruction.

The access security mechanism of the JVM requires information within a stack frame. If this information differs from the tail calling method to the called method the compiler

```
int factorial(int n) {
  Continuation c = new FactContinuation(n, 2, 1);
  do {
    try { c.apply(0);
    } catch (Continuation cont) {
      c = cont;
    }
  } while (! c instanceof ResultContinuation);
  return c.result;
}

class FactContinuation {
  int curr; int prod; int n;
  void apply(int depth) {
    if (curr > n) {
      throw new ResultContinuation(prod);
    } else {
      this.curr = curr+1;
      this.prod = prod*curr;
      if (depth < MAX_TAIL_CALL_DEPTH) {
        apply(depth+1);
      } else
        throw this;
    }
  }
}
```

Listing 2.7: Using exceptions to compress the stack

cannot simply remove the calling method's frame or the execution of the program might violate the security rules. As the goal of this thesis is to guarantee tail call optimization if a call is a tail call, we have to adopt a different definition of what tail call optimization means than the one above.

We define the meaning of tail call optimization in terms of a series of sequent tail calls. A series of sequent tail calls is a series of recursive calls where each call is a tail call. The example below shows such a series with method's `f`, `g`, `h` involved.

$$f \xrightarrow{tailcalls} g \xrightarrow{tailcalls} h \xrightarrow{tailcalls} g \xrightarrow{tailcalls} f$$

Definition Tail call optimization guarantees that a series of sequent tail calls executes in bounded stack space.

The modified VM achieves this by replacing the caller's stack frame by the called method's stack frame if possible. When this is not possible either because of the security mechanism or other technical reasons (see Chapter 4), the VM guarantee's that no stack overflow occurs in a series of sequent tail calls no matter how deep the recursion is. The maximum bound within the series executes is the memory reserved for a thread's stack. By using this definition we stay within the spirit of the purpose of tail call optimization in the context of functional languages.

3 Java HotSpot™ VM

This chapter introduces the reader to the Java HotSpot™ VM, the context of this work. We first describe the basic functionality behind the Java Virtual Machine and follow up with a description of the abstract execution model, which serves as specification for implementations of a Java Virtual Machine. The last part of this chapter explains the implementation aspects of the Java HotSpot™ VM that are relevant for this work.

3.1 Java Virtual Machine

As the name indicates a Java Virtual Machine provides a virtual machine environment for the programming language Java [22]. Sun designed Java to be a safe general purpose object oriented and concurrent language, which is portable across many platforms. The language's syntax is close to C++ with the more complicated features (e.g. multiple inheritance) and security critical features (pointer arithmetic) left out. To provide portability, Java source files are not compiled to machine specific object code but are instead translated to so called *class files*, which contain a platform independent representation of the code to execute and meta information (e.g. a symbol table). The representation is called *Java bytecode*. When a Java program runs this bytecode is executed by a Java Virtual Machine on the respective platform. The connection between the original Java program and the VM is the bytecode. The bytecode and the behavior of the VM is specified separately from the Java language. As bytecode models an abstract machine, it can be and is used as target for other languages.

Java is a type-safe language. Programs are checked against type rules during compilation to ensure that they are safe. It is for example not allowed to add an integer value to a variable declare as holding an object reference. To guarantee that those type rules also hold when the bytecode is executed on a virtual machine, the bytecode is also typed. Before a VM executes a method it verifies the bytecode. Hence class files that source from an untrusted origin, e.g. from the internet, can be trusted not to execute malicious code. If the verification fails an exception is thrown.

Java bytecode supports general purpose computing with instructions for integer and floating point arithmetic and control flow instructions. Objects are instances of classes, which define their methods. Classes can also have static methods. The bytecode features instructions to allocate objects and to invoke methods on them. The programmer does not need to take care of deallocating the memory of objects. Garbage collection performed by the VM deallocates an object when there are no longer any references to it. Many classes of memory errors are thus prevented. Concurrency is supported by the VM in the form of threads and a shared memory model and the bytecode provides synchronization primitives.

3.2 Abstract Execution Model

The Java Virtual Machine specification [38] defines the behavior of a Java Virtual Machine. This is done in the form of the specification of the bytecodes, the class file format and an abstract model of execution. The execution model specifies that before execution of a method the surrounding class must be loaded, linked and initialized. The class file format consists of methods and their bytecode, a symbol table called the *constant pool* and some meta information like the class' super class and further attributes.

Approximately 200 bytecode instructions are specified. The first byte of a bytecode instruction, the *opcode*, encodes the operation to be performed. It is followed by bytes encoding the operands. Bytecode refers to other entities like class or method symbols via symbolic references. These references are stored in the constant pool at a specific index. For brevity the actual bytecode refers to symbols via their index in the constant pool.

The loading process constructs an in-memory format of a class normally by loading its binary representation from a file. After the class is successfully loaded it is linked. An executable runtime representation of the class is created in the JVM. The linking process verifies that the loaded class is well formed and the contained bytecode adheres to the semantic requirements as stated in the specification. The bytecode verifier performs this check. Initialization finally executes class variable and static initializers. Figure 3.1 illustrates this process.

After a class is initialized, its methods can be executed or instances of it can be created. Each active method has an execution environment, which contains the parameters to the method, local variables, an operand stack and a current instruction index. This is called the active method's frame. The execution of Java bytecode models a stack machine. Bytecode instructions produce and consume values on the operand stack. For example

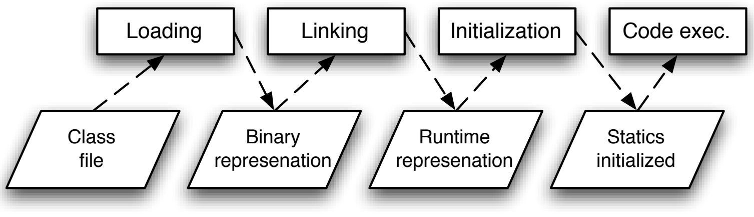
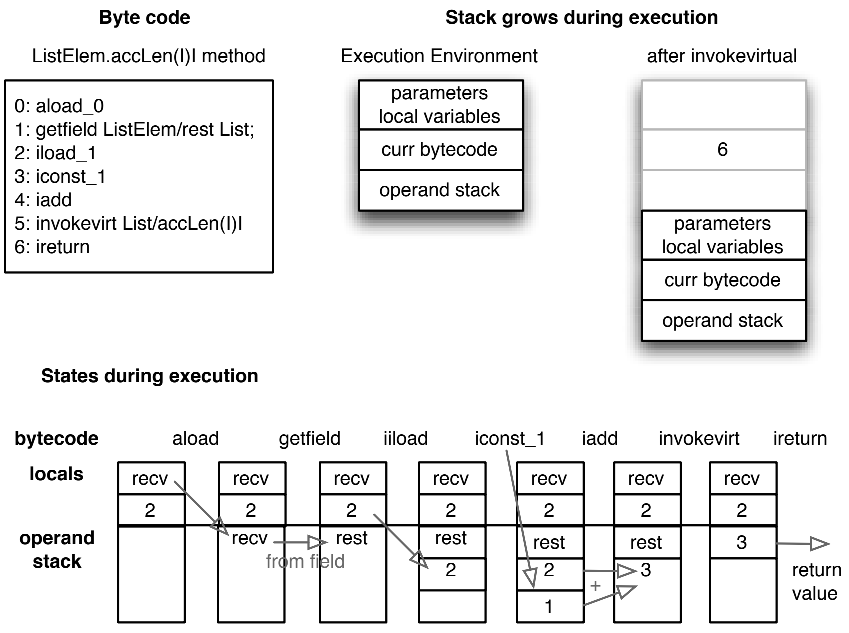


Figure 3.1: Class loading and linking process

an `iadd` instruction pops two values of the operand stack and puts the result value back onto it. Invocation of a method stores the current instruction index and creates a new execution environment, which is linked to the calling methods environment. On return of a method this link is followed to resume execution in the calling method.



```
0: aload_0
1: getfield ListElem/rest List;
2: iload_1
3: iconst_1
4: iadd
5: invokevirt List/accLen(I)I
6: ireturn
```

Figure 3.2: Environment during execution of `accLen` method

Figure 3.2 illustrates how the execution stack grows (top half). The rightmost figure shows the state of the execution stack after `ListElem.accLen` calls `List.accLen` method on the rest of the list. The old execution environment stores the fact that it has to resume at instruction index 6 once control resumes to it. A new stack frame is created for the `List.accLen` method.

The VM views local variables and parameters uniformly and addresses them with the same bytecode instructions. Parameters are considered as the first local variables, any real local variables follow them. The `ListElem.accLen` method has two parameters: the receiver object the method is called on and the integer parameter. The bytecode loads the first parameter with the instruction `aload_0` to the operand stack. The instruction prefixes 'a','i' denote that it operates on an object or on an integer.

The bottom half of Figure 3.2 illustrates how the execution environment changes during the execution of the bytecodes of the `ListElem.accLen` method when called with the argument 2. The bytecode loads the receiver to the operand stack. The `getfield` instruction uses this object to retrieve the `rest` field. Then the argument (2) is pushed onto the stack. The constant 1 is put on top. An `iadd` instruction adds those two together and stores the result back onto the stack. Then a recursive method call to `List.accLen` is performed. It uses the two operands on the stack and returns a result value on the stack (3 in this case, the receiver is an object of type `Empty`). The `ireturn` instruction causes the method to return the result. Execution is continued in the caller.

The specification does not say how to implement the features above but is strict enough, so that programs are portable across VMs from different vendors. In the following we describe how the Java HotSpot™ VM implements them.

3.3 Java HotSpot™ VM

The Java HotSpot™ VM implements above execution model by providing a mixed mode execution environment [2]. Before execution starts, the *runtime* loads the code of a method. At startup methods are executed in an *interpreter*. The VM runtime maintains information how often a method has executed. When this execution count crosses a certain threshold, an optimizing just-in-time compiler transforms the method's bytecode to machine code. Subsequent executions of the method use the optimized code. The VM does not compile all methods to machine code at startup of the program, because compilation of infrequently executed methods negatively impacts the overall execution time.

Sun delivers the VM with two different compilers: The *client compiler* and the *server compiler*. The goal of the client compiler is to achieve fast startup and a small memory footprint while still providing good performance. Its primary target are desktop applications. The server compiler targets long running applications, where a longer startup and compilation time can be tolerated. It achieves better code quality at the cost of longer compilation times.

The interpretation of long running loops has a negative impact on performance. Execution of a loop does not increase the invocation count of a method, yet there might be as much time spent during a loop as during the execution of a method. The VM addresses this problem by using an optimization called *on stack replacement* [21]. The interpreter maintains a loop counter for every method. If this counter exceeds a limit, execution is continued in a compiled version of the method, at the point where the interpreter left of.

To further enhance code performance, the compilers make assumptions about conditions at certain points in a method. This enables the compiler to perform more aggressive optimizations. To guard against possible executions where the conditions not hold, the compiler inserts checks. If this check fails, the compiled frame is transformed into a interpreted frame and execution is continued in the interpreter. This process is called *deoptimization* [28].

The runtime is also responsible for garbage collection. The Java HotSpot^{TM} VM uses an exact garbage collector. It knows exactly which addresses in memory point to objects. The interpreter knows the location of object pointers at any point during execution. When garbage collection is required the interpreter is stopped at the currently executing bytecode. The compiled code uses *safepoints*, special points in a method where the location on the stack frame of object pointers is known. When code reaches a safepoint and garbage collection is requested execution is halted. Figure 3.3 shows how the interpreter, compiler and runtime play together.

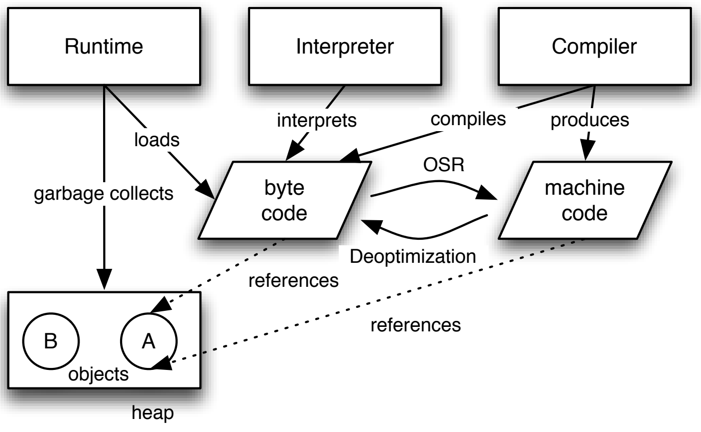


Figure 3.3: Overview of the Java HotSpot^{TM} VM

The VM is implemented in the programming language C++ and uses many of its features like inheritance, templates and stack allocated objects.

3.3.1 Runtime

The runtime provides several important features, which are required for the VM to operate. It is separated from the compiler and interpreter via a runtime interface - a collection of classes that encapsulated the functionality of the runtime and data structures to pass data to and from the runtime. This facilitates that the different parts can be developed independently of each other. The runtime is made up of the following parts.

- The classloader is responsible for loading classes and verifying them.
- Debugging information enables garbage collection and debugging of compiled methods.
- The garbage collector reclaims memory from objects that are no longer referenced.
- Deoptimization enables the transfer from compiled back to interpreted code.
- On stack replacement enables a transition from interpreted to compiled methods during method execution.

Representation of objects Within the Java HotSpot^{TM} VM, objects allocated on the heap are instances of `oopDesc` subclasses. A pointer to such an instance is called *oop*. For example, a Java language object is an instance of an `instanceOopDesc` and a pointer to such an object is an `instanceOop`. The abbreviation oop stems from the term *ordinary object pointer*, which is used for direct object pointers, in contrast to *handles*, which are used in the runtime to identify object pointers. A handle is an object that is known to the runtime and contains a pointer to an object. If garbage collection happens the pointer in the handle can be updated.

Class loading involves the creation of VM specific data structures to represent the loaded class. For every loaded class the Java HotSpot^{TM} VM creates a data structure—the `instanceKlassOopDesc`—holding the class methods and fields, a *method table* used for dynamic method dispatching, the constant pool and further meta information about the class. Class loading involves the following steps.

- Loading of the class from a file or via a user defined classloader from a different source. The VM checks that the syntax of the file adheres to the specification.
- Linking of the class checks that the semantics of the class file is correct (e.g. a class can not be its own superclass). The VM verifies the constant pool symbols and it verifies the bytecode by abstract interpretation. Objects representing the methods—the `methodOopDesc`—are created and filled with bytecode.
- Initialization calls static variable and static class initializers.

Figure 3.4 shows the created structure when the `ListElem` class is loaded. The super class entry points to a representation of the class `List`. The `fields` array contains the class field members `rest` and `element`. The constant pool contains the symbols used by the class. The field storing the *protection domain* indicates where the class was loaded from. The runtime uses this information when it performs security checks whether code called by this class's methods is allowed to perform certain operations (like reading from or writing to a file).

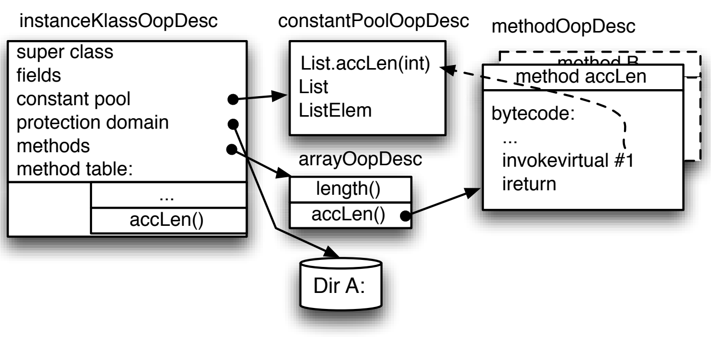


Figure 3.4: VM internal representation of `ListElem` class

Debugging information is used by the runtime for garbage collection, deoptimization and when debugging is requested. Debugging like deoptimization transfers control from compiled code to the interpreter. The compilers produce highly optimized code so that it is not possible to know at every instruction how a corresponding interpreter stack looks like, e.g. the location of the local variables, where pointers to objects are stored and which objects are locked. To still be able to perform this transition the compiler creates *safepoints* at which this information is available.

The compiler emits code at a safepoint, which checks with the runtime whether the current code should halt and execution should continue in the runtime. This can be done efficiently by trying to write to a predefined memory page. If the runtime wants to stop execution of compiled code, it protects that memory page. When compiled code reaches a safepoint it tries to write to that page, the operating system signals a memory error to the program, the thread running the compiled code is suspended and program execution resumes in a signal handler [1]. The signal handler transfers control back to the runtime.

Debugging information consists of *oop maps* and *scope descriptors.* Oop maps specify which addresses on the stack in a stack frame contain pointers to objects and are primarily used by the garbage collector. They map an instruction address to a list of offsets. To compute the location of a pointer the runtime adds this offset to the stack pointer of the method's stack frame. An oop map includes the incoming parameters of a method.

The scope descriptor consists of two lists of *scope entries* and a list of *monitor entries.* The scope entries describe where the local variables and the values on the operand stack reside in the compiled frame. The monitor entries describe at which offsets pointers to monitor objects are on the stack.

The compiler generates debugging information for each safepoint in the code and stores it together with the method's code.

Garbage collection is supported by the Java HotSpot^{TM} VM in the form of various garbage collectors. The default collector for client application is a sequential collector. The application is stopped for garbage collection and one processor is used to perform the collection. There exists a concurrent collector [19], which runs interleaved with the program for low maximum pause times and a parallel collector, which performs collection in parallel on multiple threads but stops the program for maximum throughput. All the collectors are generational collectors [56]. This means that the heap is split into several generations. New objects are created in the young generation and collected by a stop-and-copy [11] algorithm. After several collections they are moved to the old generation. The collectors usually employ a mark-and-compact [58] algorithm to collect the old generation. The young generation is collected more frequently because the expectation is that many young objects die early [37]. The VM employs a third generation - the permanent generation. This generation is collected independently of the other two. It contains objects the VM uses like `instanceKlassOopDescs` and `methodOopDescs`. Compiled code must only contain references to objects in the permanent generation. This is required for performance reasons, because otherwise every collection of the young generation has to update the addresses of object pointers in compiled methods.

Deoptimization is the process of converting a compiled frame back to an interpreted frame. If class loading invalidates an assumption in compiled code, the runtime causes a traversal of all threads and their stacks and checks whether a method on the stack used this assumption. If a frame on the stack used the invalidated assumption, the frame is marked for deoptimization. The runtime changes its return address to point to an deoptimization entry in the runtime. When the compiled code returns to this method, the changed instruction pointer causes the frame to be deoptimized. This is called *lazy deoptimization*. The compiler can also insert checks into compiled code, which, if they fail, cause the current frame to be deoptimized.

Deoptimization happens at safepoints. Figure 3.5 shows the stack during deoptimization. The runtime stub performing the deoptimization is on top of the stack. The runtime uses debugging information to construct an array of *virtual frames*. A virtual frame represents an interpreter frame with its associated state (current bytecode index, monitors, locals and operand stack) during the deoptimization process and is stored on the heap. Because of method inlining a compiled frame can result in multiple virtual frames. With the help of the virtual frames the runtime replaces the compiled frame on the stack with possibly multiple interpreted frames. The runtime then changes its return address, so that execution continues in the interpreter.

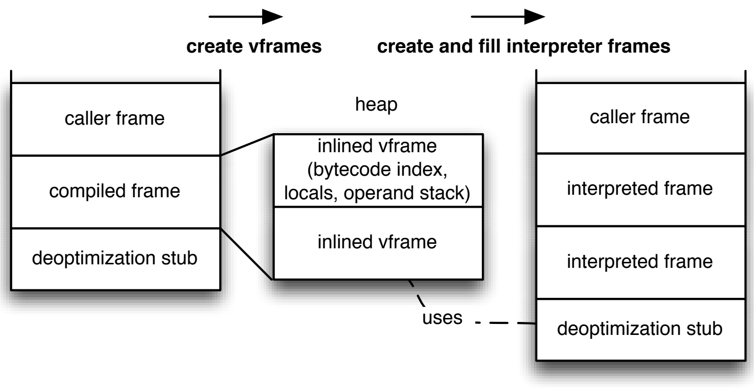


Figure 3.5: The deoptimization process

On stack replacement allows a loop running in the interpreter to transfer control to a compiled version of the loop's method. Java bytecode expresses looping constructs through backward branches. The interpreter maintains a loop counter for every backward branch. If this loop counter crosses a certain threshold, the interpreter calls into the runtime. The runtime allocates memory on the heap, which it fills with the current state of the interpreter frame: local variables and possible monitor objects. On return from the runtime the interpreter removes its stack frame and jumps to a special entry point in the compiled method - the *OSR entry*. The OSR entry creates a compiled frame, moves the data from the heap structure to the frame and resumes execution at the loop.

3.3.2 Interpreter

The interpreter operates by iterating over the bytecode instructions. It executes actions that model the effect of the current instruction. Thereby it changes the operand stack, modeled on top of the current interpreter frame. The execution of a method invocation instruction causes the creation of a new interpreter frame. The parameters that are pushed on the operand stack, thereby become part of the locals of the new interpreter frame.

At startup the VM generates code pieces that the interpreter performs for each bytecode [23]. The VM creates so called *assembler templates*, which contain the platform dependent machine code. This machine code implements the behavior of the bytecode. The start address of each template is stored in a dispatch table, which is indexed by the byte that represents the corresponding instruction.

During execution of a method the interpreter maintains a pointer to the current bytecode. The interpreter uses the value of the current bytecode to index into the table, retrieve the address of the corresponding assembler template and jump to the start. The assembler template executes usually modifying the operand stack and finally returning to the interpreter. The interpreter then increases the current instruction pointer by the size of the current instruction and execution continues there. Figure 3.6 shows how template table, the interpreter, bytecode and the assembler templates play together.

To improve performance the interpreter keeps frequently used values in registers. An example is the current bytecode index or the pointer to the current method. If an assembler template wants to use these registers, it stores their content to a reserved slot in the current interpreter frame. The same happens before method invocation. All registers holding the state of the calling method are saved to the calling method's stack frame before a new interpreter frame is created.

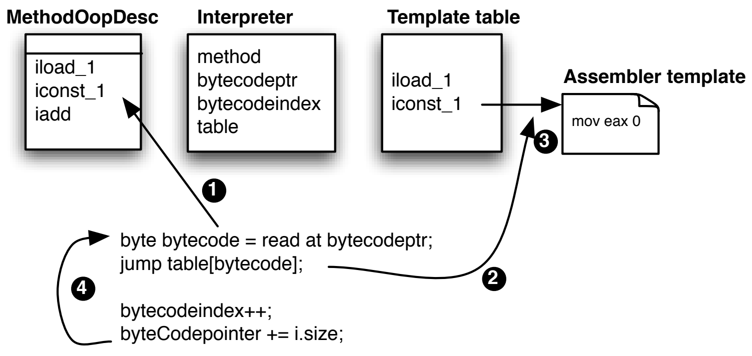


Figure 3.6: Execution of a bytecode in the interpreter

3.3.3 Client Compiler

The client compiler [33] was developed for interactive desktop applications. Because bytecode is compiled just in time, the compilation time is perceived to be part of execution time of a program. So this compiler focuses on fast compilation speed while still providing good execution speed of the compiled code. Compilation acts on whole methods and is split into multiple phases.

- Generation of a high-level program representation and application of global optimizations. The part that implements this is to referred as front end.

- Generation of a low-level representation, which is close to the platform's machine code and suitable for register allocation. This and the following phase is implemented by the back end.
- Generation of machine code by iterating over the low-level representation and emitting code.

At first the bytecode of a method is transformed into a platform independent *high-level intermediate representation* called HIR. This is done by abstract interpretation over the method's bytecode. The HIR consists of the *control flow graph* (CFG), which is modeled with a set of basic blocks. Each basic block has a list of predecessors and a list of successors. A basic block consists of a linked list of instructions, which are to be executed consecutively and which are not interrupted by a jump or a jump target. The HIR is in static single assignment (SSA) form [17]. This means that the value of every variable is only assigned once. Variables can be replaced by the instruction computing their value. This representation simplifies the implementation of several global optimizations, which are applied during and after the construction of the HIR. Examples of optimizations performed are constant folding and value numbering.

The CFG is built in two passes. The first pass computes the basic blocks by iterating over the bytecode looking for jump targets and jumps. A jump target starts a basic block, a jump or return instruction ends a basic block. Next, the second pass fills the basic blocks with a list of instructions. This list is built by abstract interpretation using a stack-like data structure to simulate the effect of the operand stack and a state array to eliminate local variable loads and stores. Variables are replaced by instructions computing their value.

All nodes of the CFG are subclasses of the `Instruction` class. There are nodes to represent the high-level instructions and basic blocks. Below is a simplified listing of the classes we can encounter in the HIR.

- The class `BlockBegin` and subclasses of `BlockEnd` (`If` representing a conditional branch, `Goto` an unconditional jump and `Return` the end of the current method) mark a basic block.
- `Phi` instructions merge result values from different predecessor basic blocks.
- Logical (`LogicOp`) and arithmetic (`ArithmeticOp`) operations. They refer to their operands via a pointer to the instruction computing the value.
- Constants are instances of the class `Constant`
- `Local` instances are placeholders for incoming method parameters.
- Subclasses of `AccessField` represent reads (`LoadField`) and writes (`StoreField`) from and to fields.

- Method invocation is represented by the `Invocation` class.

Each instruction stores its bytecode index, type and a pointer to the next instruction. Some instructions that must be executed in order (e.g. loads and stores to fields) are marked as *pinned*. Instructions not marked as pinned, can later be emitted in a different order, determined only by their data dependency (e.g. code for the input operands of an instruction must be emitted before the code for the instruction itself).

Figure 3.7 shows how the compiler links instructions together using a data structure simulating the VM's stack. The compiler creates HIR instructions for the bytecode instructions and puts them onto the stack data structure. The `iload_1` instruction causes the compiler to retrieve the entry in the state array of second local variable. In this example it contains the value of the first parameter.

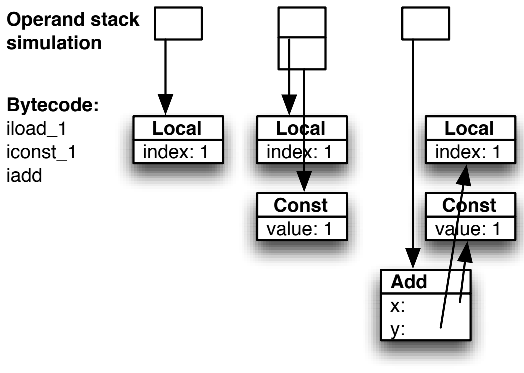
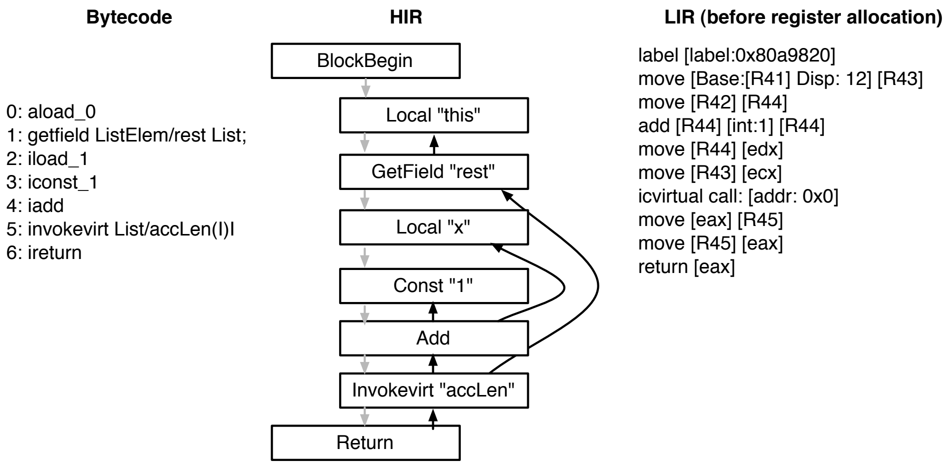


Figure 3.7: The compiler building the instruction list within a basic block

Next the backend of the compiler builds the *low-level intermediate representation*, called LIR. The LIR is no longer in SSA form. Instead of pointers to instructions computing the input value, it uses a unlimited register set to denote shared in- and outputs. If an instruction uses the result of another instructions, both instruction refer to the same virtual register. One instruction stores the result in the virtual register, the other uses this virtual register as one of its operands. In addition to holding virtual registers the operand of LIR instructions can be: physical registers (e.g. if the target requires a certain register for the operation), addresses (e.g. to refer to fields or array elements), stack slots and constants. During the construction of the LIR, the compiler replaces each HIR instruction by one or more LIR instructions. An example LIR instruction that does not exist in the HIR is the `lir_move` instruction. It is used for example when field or parameter load and stores are replaced.

This representation is more suitable for register allocation because registers are explicitly visible. Also machine code generation is easier because platform specific limitations are taken into account.

Figure 3.8 shows the HIR and LIR generated for the `accLen` method. The grey arrows represent the links that each instruction has to its next one, dark arrows represent inputs to an operation. The LIR is stored in the `BlockBegin` instruction of the HIR. Virtual registers `R41`, `R42` are used for the "this" pointer and the parameter x (they are defined in the std_entry block, which has been omitted to simplify the presentation). Every basic block starts with a label, which marks the beginning of the basic block. The next move instruction stores the value of the field rest in virtual register `R43`. It uses the "this" pointer and a field offset (`Disp:12`). The add instruction adds 1 to the incoming parameter. Before the virtual call to `accLen` the parameters are placed in fixed registers (`ecx`, `edx`) defined by the calling convention. Finally the `return` instruction passes the result of the method to the caller.



Bytecode

```
0: aload_0
1: getfield ListElem/rest List;
2: iload_1
3: iconst_1
4: iadd
5: invokevirt List/accLen(I)I
6: ireturn
```

LIR (before register allocation)

```
label [label:0x80a9820]
move [Base:[R41] Disp: 12] [R43]
move [R42] [R44]
add [R44] [int:1] [R44]
move [R44] [edx]
move [R43] [ecx]
icvirtual call: [addr: 0x0]
move [eax] [R45]
move [R45] [eax]
return [eax]
```

Figure 3.8: HIR and LIR for the ListElem.accLen method

The compiler uses a linear scan allocator [59] to assign physical register of the processor to the virtual registers. The linear scan algorithm operates on a whole method and uses one linear pass over the method. If the allocator runs out of physical registers the content of a register is saved to the current stack frame. This process is called *spilling* and the spaces on the stack are called *spill slots.*

After the register allocator has finished, the compiler emits the machine code. It translates each LIR instruction to machine specific instructions. At this point the compiler knows the state of the operand stack and where variables and monitors reside. It uses this information to generate debugging information at safepoints. Finally the compiler inserts code for uncommon cases like exception handling or garbage collection at the end of the method.

3.3.4 Server Compiler

The server compiler [44] targets long running server applications. For this kind of applications compilation time does not matter much, since it only has impact on the performance during a warm up phase. Also server systems usually have more processors and compilation can be done in background. The compiler produces highly optimized code at the cost of longer compilation times.

In contrast to the client compiler, which uses a control flow graph (basic blocks where each basic block maintains a sequence of instructions), the server compiler uses a data structure [15] as intermediate representation similar to a program dependence graph [20]. Both data dependence and control dependence are represented by edges pointing from the use of a value to its definition. The resulting graph is more complex but allows for an easier implementation of global optimizations.

The intermediate representation, which the compiler uses until register allocation, is also in SSA form. The IR uses two kinds of nodes. In the beginning the compiler builds a platform independent representation of the program called the *ideal graph*. During instruction selection the compiler generates machine specific nodes from the ideal graph resulting in a *MachNode graph*.

The server compiler uses the following phases, which are also found in traditional compilers.

- Parsing constructs the ideal graph from bytecodes. During parsing the compiler performs local optimizations.
- Machine independent optimization operate on the ideal graph. Some of them are applied iteratively until they reach a fixed-point, e.g. no changes happen from one application to another. During this phase global value numbering, constant propagation, dead code elimination and some loop optimizations are applied.
- Instruction selection generates the MachNode graph using a bottom-up rewrite system [45]. The ideal graph is split into subtrees. An optimal assignment of machine specific nodes to the subtree is built by recursively applying a BURS algorithm. The possible instructions and their associated costs are recorded in an architecture description file, which is used by the algorithm.
- A global code motion algorithm [14] constructs a control flow graph. Basic blocks are created and filled with instructions.
- Scheduling orders the instructions within a basic block. The compiler selects between available instructions based on a score. The score is computed using several

heuristics (e.g. instructions that store to the stack or memory are given a higher score in order to free registers used for their input operands early).

- Register allocation converts the graph into a non-SSA form and assigns physical registers. A graph coloring [10] allocator is used. It is slower than the linear scan allocator of the client compiler (Asymptotic time complexity of O(n^2) instead of 0(n) where n is the number of virtual registers) but produces better code (less spills). At the end of register allocation the compiler generates the debugging information for safepoints.
- Machine code is emitted by iterating over the MachNode graph.

The server compiler uses a mechanism similar to the client compiler to build the graph. It maintains the current VM state, which holds the state of the operand stack and the values of the method's locals during parsing. The program dependence graph consists of nodes representing the operations and edges representing control flow and data flow dependence. The nodes are implemented as instances of a subclass of `Node`. Edges are simply pointers to other nodes. Each node has an array of `Node` pointers, which represent its inputs and produces one value. If a node has several results, it conceptually returns them as tuples, e.g. data and control flow. The compiler inserts projection nodes to project a result out of this tuple. Examples for nodes that only produce a data output are arithmetic operations (e.g. `AddI`) or logical operations (e.g. `CmpI`). An example of a node that produces only control flow output is the `If` operation. A `Region` node takes the place of basic blocks. It merges control flow outputs from other predecessor blocks. A corresponding `Phi` node merges their data outputs.

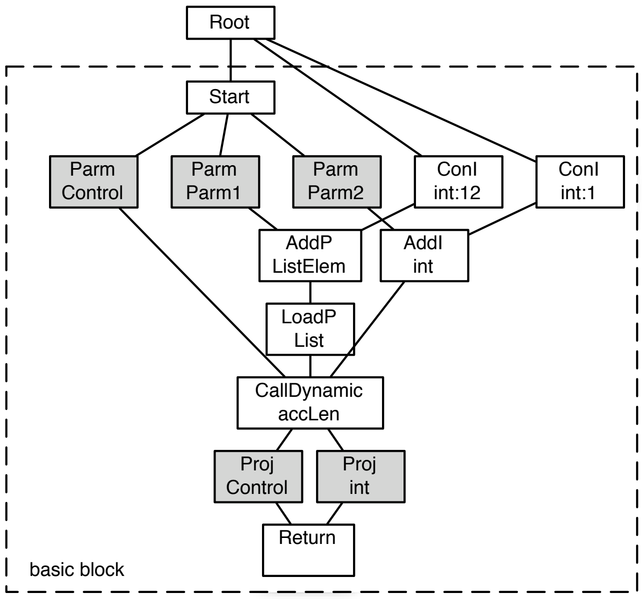


Figure 3.9: Ideal graph of `accLen` method

Figure 3.9 shows a simplified (e.g. exceptional behavior is left out and some output values are omitted) ideal graph created by the compiler for the `accLen` method. The `Root` node is generated for every method. It is used as input for instructions that exit the method and simplifies algorithms operating on the graph. The `Start` node represents the state of a method before any instructions are executed. It has a result tuple containing the control flow and the parameters. Those values are extracted by projection nodes `Parm` Control, `Parm` Param1 (containing the object) and `Parm` Param2 (containing the integer parameter). The bytecode instruction `getfield` is replaced by two instructions that compute the field's address and load its value (`AddP` and `LoadP`). The `iadd` is represented by the `AddI` instruction, which uses the method's integer parameter and a constant (`ConI`). The method call (`CallDynamic`) uses the two data values. It also has a control flow input, which source is the control flow output of the `Start` node. Finally the `Return` instruction marks the return from the method. It merges control and data flow outputs (`Proj` Control and `Proj` int) of the method invocation.

4 Implementation

This chapter describes the implementation of tail call optimization on the Java HotSpot^{TM} VM. The optimization is implemented for the IA-32 platform. The goal of this work are guaranteed tail calls, i.e. the programmer marks a call as tail call and the VM guarantees that the call is optimized or it throws an exception. The bytecode verifier checks the conditions that must hold for a correct tail call. A method call is treated differently within the VM dependent on whether the call target is known to be unique at compile time or not. Tail calls use the same mechanism as regular calls to improve call performance.

Removing the caller's stack frame might violate security semantics. The implementation detects such cases. Depending on with which option the VM was started, it either throws an exception or executes a regular call. If the execution stack overflows the deoptimization infrastructure is used to compress the stack. We can therefore guarantee that a series of tail calls executes in bounded stack space even in the presence of the Java access security mechanism.

4.1 Target Hardware Platform

Sun provides implementations of the VM that run on two architectures: x86 and Sparc in two addressing modes 32bit and 64 bit. Both the compilers and the assembler templates generate machine code of the target platform. This work provides an implementation of tail calls for x86 in 32bit mode also referred to as *IA-32* [31]. The execution environment of IA-32 is made up of the following parts.

- Address space: The VM generally uses a flat memory model. Memory appears as a continuous space of maximum 4 Gigabyte.
- General-purpose registers and operations: 8 general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` and `esp`), an instruction pointer (`eip`), control and flags registers are used together with integer, control flow and memory access instructions to handle program flow and integer arithmetic.

- Stack operations: Several operations (e.g. `push`, `pop`, `call` and `ret`) are available to manipulate the program stack. They manipulate or use the stack pointer register `esp`.
- x87 Floating Point Unit: Several floating point registers together with floating point instructions provide support for floating point arithmetic. The floating point data registers are organized as a stack. This complicates code generation.
- XMM registers: With the introduction of the SSE extension 8 additional XMM register are available, which can hold floating point values. Several instructions operating on these registers are available which implement single (SSE1) and double (SSE2) precision floating point arithmetic.

IA-32 is a register-memory architecture [25] and as such can access memory as part of any instruction. The instructions operate on zero or more operands. The instruction format of logical, arithmetic and data move instructions allows the specification of two operands and one must be a register operand. The other operand can be either an immediate value, a register or a memory operand. The following listing shows typical instructions.

```
mov %eax %ebx   // Move the content of register ebx to eax.
mov %eax [%ebx] // Move the content of memory at the address
                // designated by the value in ebx to eax.
mov [%ebx] %eax // Move the content of eax to memory at address ebx.
add %eax %ebx   // Add eax ebx and store the result in eax.
add %eax 5      // Add five to the value in eax.
```

The VM detects whether the SSE instructions are available and uses the appropriate instructions for encoding floating point arithmetic. For this work we assume the presence of SSE2, i.e. it was not verified that code generated for the floating point unit is correct in the presence of the tail call implementation. Both compilers and the interpreter use `esp` as stack pointer for their execution stacks.

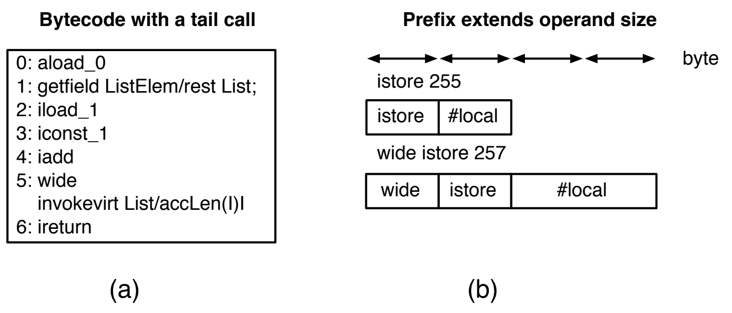
4.2 Bytecode Instruction Set Changes

Java bytecode features four bytecode instructions for the different kinds of method invocations.

- `invokestatic` calls a static method in a class.
- `invokevirtual` calls a member function of an object. The method dispatch code uses the type of the object, i.e. the objects class or one of its super classes must implement this method.

- `invokeinterface` calls a method implemented by an interface. The call searches the particular runtime object for the appropriate method.
- `invokespecial` calls an initialization method, a super class method or a private method. The call target is known before runtime.

Each bytecode instruction has a fixed size and is encoded in a series of bytes. The first byte denotes the instruction type and is possibly followed by operands. Currently the `wide` prefix is used to prolong the operand size of the following instruction's operand. Figure 4.1 (b) illustrates this. Normally the `istore` instruction has an operand of one byte specifying the index of the local variable to store to. If the `istore` instruction is prefixed with `wide` a two byte operand can be used to specify the index. Using the `wide` prefix before a method invocation has no meaning and code containing it is rejected by the bytecode verifier. We use it to denote a tail call. The VM is modified so that it recognizes such marked calls and handles them appropriately. Figure 4.1 (a) shows the modified bytecode of the `ListElem.accLen` method.


Bytecode with a tail call

```
0: aload_0
1: getfield ListElem/rest List;
2: iload_1
3: iconst_1
4: iadd
5: wide
   invokevirt List/accLen(I)I
6: ireturn
```

Figure 4.1: Bytecode of a method containing a tail call

4.3 Bytecode Verifier

During class loading the bytecode verifier is called to check that the bytecode does not compromise the virtual machine. The verifier checks that the following conditions hold [9]:

- The maximum operand stack size specified in the class file is maintained during execution of the bytecode.
- Targets of control flow instructions must be within the bounds of the methods bytecode and must point at the start of an instruction. Control flow can not fall off the end of the bytecode.
- Instructions have appropriately typed arguments on the operand stack.

- Loads from local variables have the appropriate type.
- The start and end values for the exception handler point at the start of instructions. The index that specifies the start of the exception handler code must start at a correct instruction.

Before Java version 6, the Java HotSpot^{TM} VM used a data flow analysis [35] to verify above conditions. With the introduction of Java 6 this check proceeds by type checking [48] instead of type inference. This was done in order to reduce startup time. The type inference algorithm requires a complex data flow analysis, which takes longer than the type checking implementation. Class files that want to use the new verification mechanism specify a class file format version of `50` or greater. To aid the type checking such a class file must also contain extra type information for certain points in the method. This data is called the *stack map table*.

The stack map table consists of zero or more *stack map frames*. A stack map frame describes the type state of local variables and the operand stack at a specific bytecode index. It is generated at every instruction that changes the control flow, so that the next instruction in the control flow is not the next instruction in the bytecode stream (e.g. `goto` or `ireturn`) and at target of branches. The verifier uses this data during type checking. It simulates the effect of instructions between two stack map frames. If it arrives at an instruction that is associated with a new stack map frame it compares its internal representation with it. If they match, i.e. the types on operand stack and local variables are the same, it proceeds. Otherwise the verification fails.

```
// Initialized with max_stack, max_locals of the current method.
StackMapFrame current_frame;
// Stack map table contains stack map frames stored in class file.
StackMapTable table(method);
// Representation of the methods bytecode.
ByteCodeStream bcs;
bool changes_control_flow = false;
while (bcs.hasNext()) {
  opcode = bcs.next();
  bci = bcs.bci();
  if (changes_control_flow || table.has_frame_at(bci)) {
    // Check that stack map frame at bytecode index from class file matches current_frame.
    table.frame_at(bci).match(current_frame);
  }
  switch (opcode) {
    case _iadd:
      // Fails if operand stack does not contain an integer.
      current_frame.pop_stack(integer_type);
      current_frame.pop_stack(integer_type);
      current_frame.push_stack(integer_type);
      changes_control_flow = false;
      break;
    case _goto:
      table.check_jump_target(current_frame, goto_target);
      changes_control_flow = true;
      break;
```

Listing 4.1: Bytecode verification of a method

Figure 4.1 shows a simplified version of the code used by the verifier. If the verifier encounters an `iadd` instruction, it pops two integer types off the simulated operand stack. If this fails because the operand stack does not have two such values the verifier throws an exception - the verification has failed. Then the result type of `iadd` is pushed. The `goto` instruction causes verification that the current stack map frame matches the frames stored in the stack map table at the current instruction and at the jump target. To deal with tail calls the verifier checks the following conditions at a tail call site.

- The invoke instruction must be immediately followed by a return instruction.
- The current method is not synchronized.
- No exception handlers are installed over the tail call invocation.

4.4 Method Invocation Overview

Method invocation plays an important role in an object oriented language. Dividing functionality between many methods of objects is considered good practice. Therefore it is important to have fast method calls to get a well performing implementation. In general we can differentiate between two kinds of method invocations.

- Static calls have a target that is known before runtime. They can usually be translated into a simple `call target` assembler statement where `target` is the address of the called method.
- Dynamic calls have a target that is computed at runtime. They usually dispatch on the type of the receiver object. Examples of dynamic calls are virtual method and interface calls. They are implemented by first computing the address of the called method based on the objects method table and then calling that address. The following example shows the code for a virtual method call via a method table.

  ```
  mtable = receiver.type.mtable // Query object for method table.
  target = mtable[meth_offset]
  call target
  ```

Static calls are faster, not only because the target does not have to be computed at runtime, but also because modern processors can schedule the code after a call to a known target earlier, further improving performance of the executed code. In an object oriented language like Java most calls are virtual calls or interface calls. For a virtual call the method offset is known at compile time. It can be computed because all classes that implement a virtual method are arranged in a class hierarchy. If a class `A` extends a class `B` it inherits all its methods. The method table of `B` then simply consists of the method table of `A` with new methods of `B` appended to it.

But a class can implement an arbitrary number of interfaces independent of the class hierarchy. At runtime the method table has to be searched for an interface method because the offset can not be predetermined. A class `C` might implement interface `I1` with a method `i1`. Another class `D` might implement two interfaces `I1`,`I2` with respective methods `i1`,`i2`. A third class `E` might only implement interface `I2`. The offset for interface `I2.i2` in the method table is either zero or one depending on the type of the object. Dispatch code for interface calls has to proceed by searching the table.

The interpreter resolves call targets every time it arrives at a call site. To improve performance of calls in compiled code the VM uses three techniques.

- Inlining removes the overhead of a call by replacing it with the body of the called method. The compiler can only inline a method if the target is known at compile time.
- Class hierarchy analysis CHA [18] inspects the loaded classes. If there is only one class that implements a virtual method or only one class that implements an interface method, the compiler can treat the call like a static call. It either inlines the call or emits code for a static call. Because the JVM supports dynamic class loading, the runtime records a dependency between such optimized calls and the calling method. Deoptimization is initiated for the calling method, if class loading causes the result of a previous CHA to be invalidated.
- Inline caching [27] works bases on the assumption that although a dynamic call cannot be guaranteed to always have one target, specific call sites might encounter only one dynamic receiver type during execution of the program. Instead of a dynamically dispatched call, a static call to the first encountered receiver's method is emitted. The target of this call is a special method prolog that checks whether the receiver is of the expected type. If this check fails, the call target is replaced by a dynamically dispatched call.

To support inline caches and dynamic linking of call sites, the compiler does not emit the dispatch code or direct static calls to the target method. Instead it emits a call to the runtime. When the call site is invoked for the first time control transfers out of the compiled code into the runtime. The runtime can then patch the call site with the appropriate target.

Figure 4.2 (a) illustrates how inline caching works. Initially every dynamic call goes to the runtime. The first time the call executes, control transfers to the runtime. It patches the call to point to a special prolog of the current receiver's method and continues execution there. The prolog checks that on subsequent calls to this method the receiver type is the same (the call passes the assumed receiver type as argument). If this check fails, the runtime relinks the call to a dynamically dispatched call. Figure 4.2 (b) shows how dynamic dispatch works using the type pointer, which is embedded in the header of every object. A header word points to the object's class. Embedded in that class is the method table, which is used during a dynamic call to get its target.

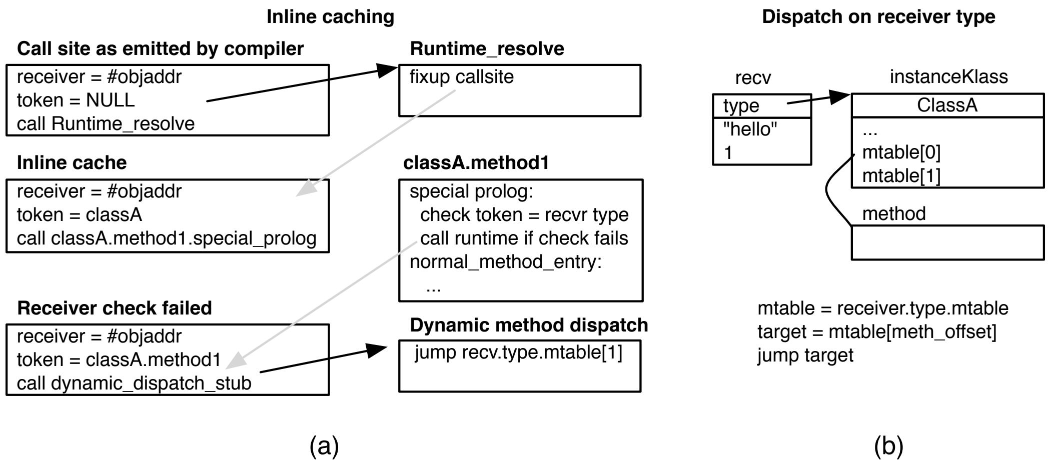


Figure 4.2: Dynamic dispatch and inline caching

A call site in compiled code can be in one following four states depending on the type of the call.

- *Unresolved*: After a method is compiled, all call sites target the runtime.
- *Static*: If the compiler has determined statically that there can only be one possible receiver type (either through CHA or because the invocation is a static invocation), the runtime patches the call target to point to the receiver's method entry.
- *Monomorphic*: All dynamic (virtual and interface) calls that do not have a statically determinable unique receiver type initially use an inline cache. Because the call can handle only one type of object, it is called monomorphic.
- *Polymorphic*: If the receiver type check fails at a monomorphic call, the target of that call is changed to point to a stub that does the dynamic dispatching.

A method call in the VM in general consists of the following parts.

- Lowering of arguments. The interpreter puts all arguments on the stack. This happens as part of the simulation of operand stack bytecodes. The compiler uses some registers for arguments and places the rest on the stack.
- Setting special tokens (compiler only). Before the call instruction the compiler emits an instruction that stores a value to a register or to a stack slot that is reserved for this purpose. This instruction is later patched by the runtime to contain the expected class (`instanceKlassOop`) in case of a inline cache call or to hold the `methodOop` for polymorphic calls (used by the interpreter to determine which method to execute). The tail call implementation sets an additional security token to deal with access security (see Section 4.6).
- Call to target. Depending on the type of call and the state of the call site the target is either the runtime, an inline cache, dynamic dispatch stub or an adapter to interpreted code.
- Check at target. The target might include a check that involves the token set before the call. If the check fails, control resumes in the runtime. The runtime handles the failure by replacing the call target.

To support monomorphic calls every compiled method has two entry points: an *unverified entry point* (UEP), which performs the class check of the inline cache and resumes control in the runtime if the check fails, and a *verified entry point* (VEP) the start of the normal prolog of the method, i.e. it contains the method's stack frame setup followed by the method's code.

Compiled methods expect a different layout of their arguments than interpreted methods. To bridge between interpreted and compiled methods adapters exist, which shuffle the arguments accordingly. The adapters are code fragments that are executed before execution continues in the interpreter or in compiled code.

When implementing tail calls, the question arises, when to move the arguments to the caller's caller frame and when to remove the calling frame. This can be done before the actual call instruction happens as explained in Section 2.4. The interpreter actually does this. It knows the target and can determine whether the tail call is allowed (looking at the security information). A second variant is to place the arguments like for a regular call but then proceed execution of the call in a special prolog, which moves the arguments onto the caller's caller and pops the caller frame off the stack. After this, the prolog continues at the normal method entry. The advantage of this method is that the prolog, i.e. the receiver, can decide whether to really pop the frame or proceed like a normal method call. For compiled code this variant of executing a tail call was initially chosen for the following two reasons.

First, a call site may go into the runtime. Whenever control enters the runtime a garbage collection can be triggered. This collection causes a stack traversal looking for object pointers. Because the tail call has moved the arguments onto the calling method's incoming argument area instead of the outgoing argument area the runtime does not see the object pointers. Figure 4.3 illustrates this problem.

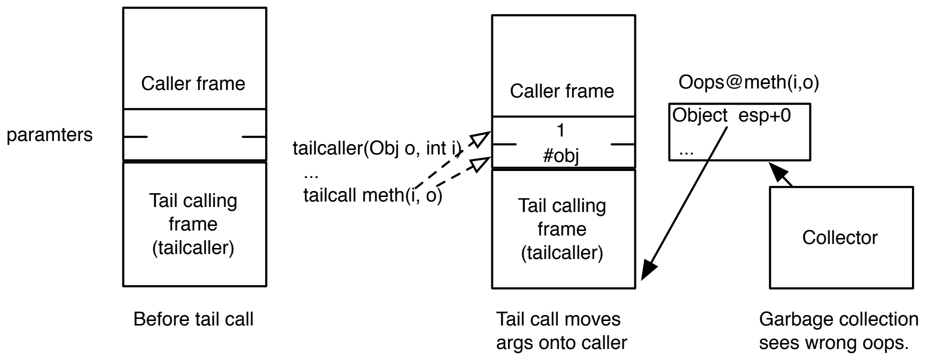


Figure 4.3: Wrong oop map at tail call when entering runtime

If the arguments are moved at the target, this problem does not exist because the runtime sees the state of the frame before the arguments are moved. This problem can be dealt with by introducing special code in the runtime, which corrects the oop map in such a case.

Secondly, a polymorphic tail call can decide in the dynamic dispatch code not to remove the frame from the call stack because of security issues (see Section 4.6). If the code at the call site has already moved the arguments onto the caller's caller before the call, execution can not continue at a normal method entry. The arguments have to be moved back onto the caller's outgoing parameter area. This complicates the implementation because extra entry points are needed, which shuffle the arguments back onto the caller frame. Hence it was initially decided to move the arguments in the called method's entry. Compiled tail calls move their arguments and pop the frame at the call target (the called method's prolog). The existing method entry points are extended by tail call entry points, which mirror the functionality of the normal entry points and also move the incoming arguments and remove the calling frame.

4.5 Method Abstraction

Call sites in compiled code can be static, monomorphic or polymorphic. The call could be executed in the interpreter and target a compiled method or vice versa. To support the different kind of calls the VM uses the following data structures.

- `instanceKlassOopDesc`
- `methodOopDesc`
- `nmethod`
- `AdapterHandlerEntry`

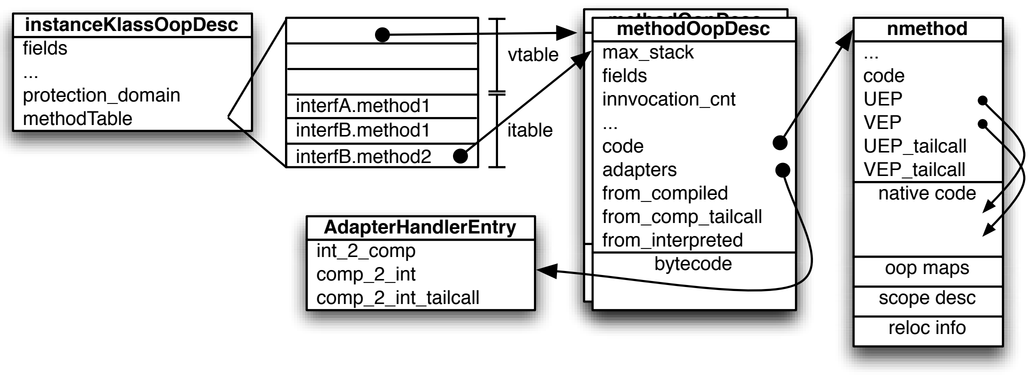


Figure 4.4: Data structure used for method calls

Figure 4.4 shows how those data structures are connected. The `instanceKlassOopDesc` is the internal representation of a Java class. Among other information (e.g. offset of fields) it contains the method table. The method table consists of entries for the class's virtual methods, followed by entries for interface methods. The table is used during dynamic method dispatch. A virtual method call indexes into the table to get the target method. An interface call searches the table for a matching interface. The protection domain is used by dynamically dispatched tail calls to verify that caller and callee are in the same security domain.

The `methodOopDesc` represents a Java method. It is generated during class loading and contains the methods bytecode, some meta information (e.g. the maximum operand stack size or the invocation counter), a pointer to the method's adapters, entry points to continue execution at, and a pointer to the method's compiled code (the `nmethod`).

The `nmethod` describes a compiled method. It is generated after the compiler translated a method and stores the method's code, debugging information, addresses of the different method entry points and relocation information. The relocation information stores the call instructions' types (e.g. static or dynamic call). The runtime uses the method entry points and the relocation information during resolving the call target, e.g. when it sets the target of a monomorphic call.

The `AdapterHandlerEntry` contains the adapter code to be executed when a transition between compiled and interpreted code happens. It is shared among methods that have the same signature. The method entry points in the `methodOopDesc` might point to such an adapter. The `from_compiled` entry points to a compiled-to-interpreted adapter as long as the method is not compiled. After the method is compiled the runtime sets the value of this entry point equal to the compiled code's prolog. Thus, dynamically dispatched method calls that use this entry point always encounter the correct code.

To support tail calls we add extra entry points to a compiled method. The code at this entry points performs the transfer of the methods parameters onto the calling's frame incoming parameter area (i.e. the caller's caller frame) and removes the calling frame. Finally the code continues execution at the normal method entry.

4.6 Access Security Mechanism

The Java HotSpot$^{\text{TM}}$ VM makes security decisions based on the execution stack [57]. The VM has access to valuable system resources, e.g. files or network access. As classes can be dynamically loaded and the code in those classes can sometimes be only partially trusted, it is desirable to restrict access to resources for certain classes and their methods. To support this, the VM features the concept of protection domains [51]. In the context of the VM a protection domain encloses a set of classes whose instances are granted the same permissions. A protection domain object can be constructed with a static set of permissions, or it can be instantiated, so that the permissions for this protection domain object are obtained at runtime using a policy object. The policy object can be user defined. The default policy object obtains the permissions from a file.

The class loader assigns the protection domain to a class during loading. This makes it possible to assign different permissions to classes that were downloaded over the internet than to local trusted classes.

To check whether a method has permission to perform an action in the current execution context, it calls the `checkPermissions` method on the `AccessController` object. To indicate what kind of permission it needs, it passes an instance of a subclass of `Permission`. This allows the definition of arbitrary permission kinds, without the access controller having to know every type of permission. It can act on the abstract `Permission` type.

```
FilePermission fp = new FilePermission( 'file/path', 'read');
AccessController.checkPermissions(fp);
// Read file ...
```

The `checkPermissions` method then traverses the execution stack of the current thread gathering all callers. Each called method maps to a protection domain via the class it is defined in. The so obtained protection domains are used to answer, whether the checkPermissions call succeeds or fails with a `SecurityException`. The permissions are built by intersecting all permissions obtained. If there is one protection domain that does not have the requested permission the check fails. Figure 4.5 illustrates this procedure.

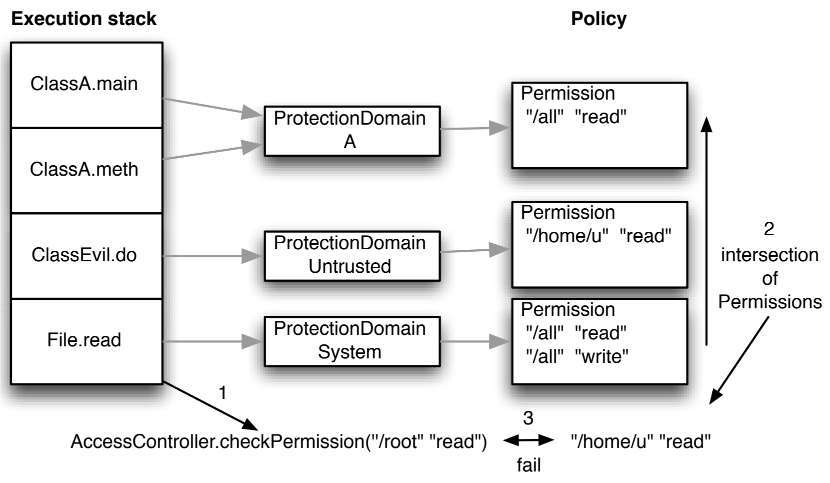

Execution stack
Policy
ClassA.main
ClassA.meth
ClassEvil.do
File.read
ProtectionDomain A
ProtectionDomain Untrusted
ProtectionDomain System
Permission "/all" "read"
Permission "/home/u" "read"
Permission "/all" "read" "/all" "write"
2 intersection of Permissions
1
AccessController.checkPermission("/root" "read")
3
fail
"/home/u" "read"

Figure 4.5: Access security mechanism

If a method's frame is removed, the security behaviour might change. Hence a tail call could change the security behaviour, because it removes the calling frame. If the calling frame is the only frame on the stack that points to its protection domain, a possibly more restrictive permission set is lost. Therefore a possibly dangerous operation, which might have otherwise failed with a security exception, can now succeed. A safe tail call implementation must prevent this from happening.

On the other hand, if the calling method and the tail called method have the same protection domain (in the methods' classes), the caller frame can safely be removed. During a stack walk the protection domain is still observed. We can illustrate those two cases with Figure 4.5. If the method `meth` tail calls `do`, `meth`'s stack frame is removed. This has no observable impact on the security behaviour because the protection domain `A` is still referenced by `main`. But if the method `do` tail calls method `read`, there is no frame left pointing to the untrusted protection domain. In this case the permission check succeeds, although the untrusted code has executed.

To prevent the removal of unobserved protection domains, every call site needs to traverse the current call stack, checking whether the protection domain of the caller already occurs in one of the previous callers. This is feasible but incurs a considerable execution time overhead. Our implementation uses a more conservative approach. The VM prevents illegal security behaviour by checking that the protection domains of tail calling method and tail called method are equal. If they are not, it proceeds either by leaving the frame on the stack (i.e it proceeds like a normal call) or by throwing an exception. The desired behaviour can be set by a parameter passed to the VM. The overhead of this method is much lower, as long as the stack does not overflow.

For static and monomorphic calls the check only has to be done once, when the runtime links the target of the call. Protection domains are set during class loading and can not be changed thereafter. Therefore it is sufficient to check whether caller and callee domain match once for calls with one target that does not change. Polymorphic calls (using the dynamic dispatch path) can have multiple targets. Therefore the equality has to be checked on every call. Polymorphic tail call sites pass a security token containing the protection domain of the caller method. The dynamic dispatch stub for tail calls checks whether the protection domain of the receiver method matches. If the check fails, it proceeds at a normal method entry. Otherwise it continues at the tail call entry. Figure 4.6 shows this.

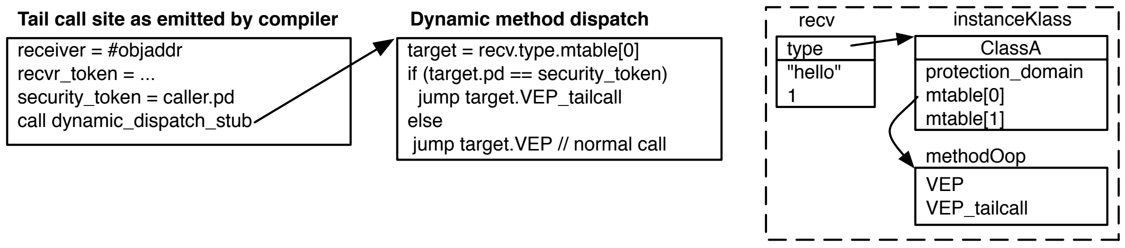


Figure 4.6: Dynamically dispatched tail call uses security token

If there is a series of tail calls, which are disabled because of differing protection domains, and the series is long enough, a stack overflow would normally happen. The implementation prevents this by compressing the stack at that point using the deoptimization infrastructure. A vframe representation of the stack is built containing only those vframes in a series of tail calling vframes that have different protection domain. Because a typical application involves only a hand full of protection domains the stack is significantly compressed and execution can resume. Figure 4.7 illustrates this process.

4.7 Interpreter

The interpreter executes a method by interpreting bytecode per bytecode. It executes an assembler template for each bytecode. To support tail calls templates for `invokevirtual`, `invokestatic`, `invokespecial` and `invokeinterface` that are prefixed with the `wide` bytecode were added. This section describes the additions to the interpreter.

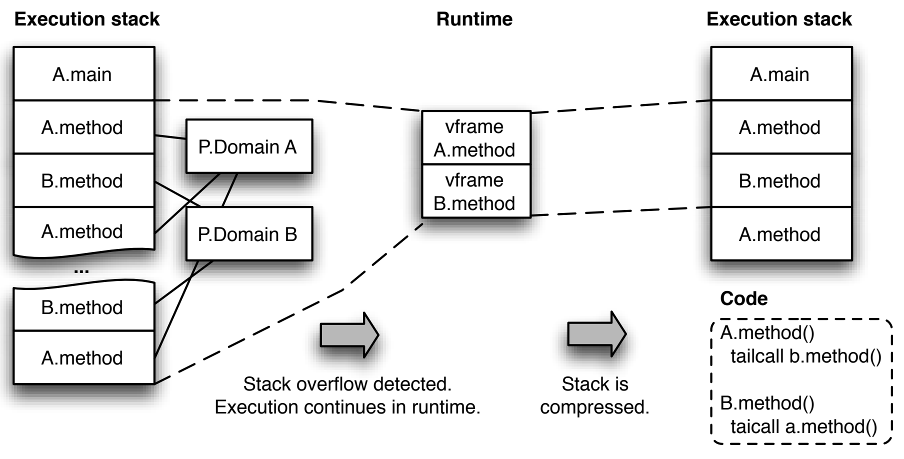


Figure 4.7: Tail call stack frames are lazily compressed

4.7.1 Dispatch of 'wide' Templates

In order to support bytecodes that are prefixed with the `wide` instruction, the interpreter maintains a second template dispatch table, which stores the associated assembler templates. The assembler template for `wide` uses this second table to dispatch to the template for the following bytecode. The following code shows the content of the `wide` assembler code.

```
wide:
  // index contains the opcode, e.g. invokevirtual = 182
  index = bytecodepointer[1]
  template = wide_table[index]
  jump template
```

During startup the template interpreter gets built by executing functions that generate the assembler templates into a code buffer. For each bytecode a C++ method is executed that generates the corresponding machine code. The start address of this machine code is then stored in the dispatch table. To handle tail calls four methods are added that generate the tail calling code for the respective invocation bytecode. The start addresses of the resulting code is stored in the `wide` dispatch table at the corresponding offset.

4.7.2 Interpreter Execution Environment

The execution environment of an interpreted method consists of the interpreter stack frame and the content of the machine registers. The interpreter frame consists of:

- Locals and parameters: The interpreter stores parameters and locals in a continuous area at the beginning of the frame. Accessing local variables and parameters can be handled uniformly without special code. At method entry the slots on the expression stack holding parameters become part of this area.
- Return address: holds the link to the caller. Contains either an address of the interpreter or of compiled code.
- Old frame pointer: stores the frame pointer of the caller frame and is saved at method entry by the interpreter.
- Old stack pointer: contains the real stack pointer of the caller frame. It is saved at method entry. Adapter code between compiled and interpreted frames creates an area between the two frames for shuffled parameters. The interpreter removes this area on exit of a method using this pointer.
- Last stack pointer: stores the value of the stack pointer, i.e. the state of the operand stack before a method invocation. The interpreter restores this value on return of a method invocation. This allows the creation of a variable area between two frames to store shuffled parameters in case of a interpreted to compiled method transition.
- On stack replacement state: stores a boolean value whether on stack replacement is allowed for this frame. Added for the tail call implementation to be able to turn off on stack replacement for certain methods running in the interpreter.
- Monitorblock: contains data structures for locks. Each monitor object uses two stack slots, where an optional object header word and a pointer to the locked object can be stored. This is used to implement a thin locking scheme [3, 4]. If only one thread locks an object, the lock can reside on the thread's stack frame and can use cheaper synchronization instructions than when multiple threads use the lock.
- Operand stack: holds the stack the bytecodes operand on.

During execution of a method the interpreter stores the address of the current bytecode instruction in register `esi`. `ebp` is used as frame pointer. To access values stored in the current stack frame the interpreter uses offsets relative to `ebp`, e.g. the old stack pointer is available at address `ebp` plus 8 (each stack slot has 4 bytes). `esp` is used as the stack pointer. Figure 4.8 shows the layout of an interpreter frame.

4.7.3 Interpreter Method Execution

The interpreter progresses by executing assembler templates for each bytecode. In addition to the normal bytecode templates the interpreter also has several entry points, which it uses during method execution. The *method entry point* contains an assembler

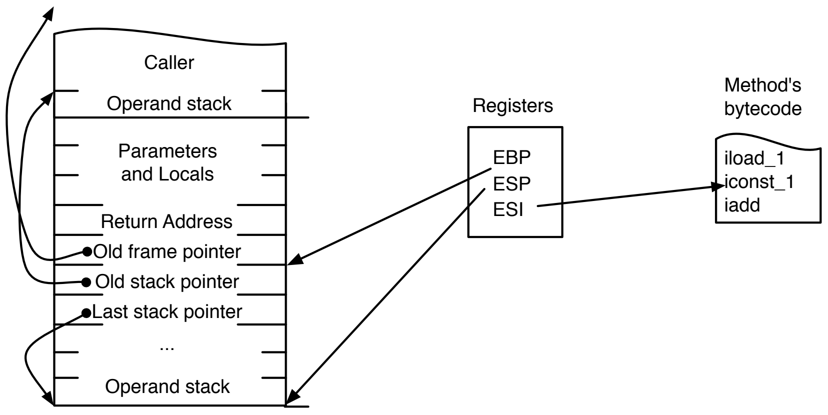


Figure 4.8: Interpreter stack frame layout

template, which sets up the new stack frame and gets executed when the interpreter enters a method. The *return entry point* is the continuation point after a method has returned. It restores the state of the method, i.e. the values of the registers, after a method call has returned.

To understand how the interpreter works during the execution of a method, it is useful to split its actions into 5 stages.

- Normal bytecode execution of bytecodes that do not call another method or throw an exception. The assembler template for the respective bytecode is executed, manipulating the operand stack or accessing local or object fields.
- Invocation of a method. One of the four invocation bytecodes gets executed. The interpreter enters the corresponding template. The template stores the state (bytecode pointer, last stack pointer etc.) to the current stack frame. Next it pushes the interpreter's return entry onto the stack and jumps to the entry of the called method. If the call is a tail call the parameters are moved to the caller's caller frame, the return address is adjusted and the current frame is removed.
- Entry of a method. The method entry template gets executed, allocating a new stack frame. It adds stack slots for local variables and moves the return address below them. The remaining stack slot values are placed on the stack. After the stack frame is created, the interpreter jumps to the method's first bytecode template.
- Exit of a method. The template of one of the `return` bytecodes gets executed. It removes the current stack frame and continues execution at the caller's return address.

- Return from a method. The interpreter's return entry restores the stack pointer using last stack pointer, removes the parameters from the operand stack and continues execution at the next bytecode instruction.

Invocation of a method One of the `invoke...` templates gets executed. The following pseudo code shows the code contained in an invoke template. In reality functions expand to multiple assembler statements.

```
invoke_virtual:
  safe_curr_bytecode_pointer();
  // Load methodOop, vtable index and flags (contains number of parameters).
  load_invoke_cp_cache_entry(methodOop, index, flags);
  load_receiver(ecx, esp, flags); // Load the receiver from stack to ecx.
  push_return_adress(Interpreter::return_address_entry);
  klass = receiver.type;
  targetMethodOop = compute_receiver_target(klass, index);
  profile_call(targetMethodOop);
  safe_last_sp();
  ebx = targetMethodOop
  jump targetMethodOop.from_interpreted;
```

The template first safes the current bytecode pointer (bcp) to the stack. Next it loads the callee's `methodOop`, method table index and flags containing the number of parameters to registers. This is done by looking at the constant pool index following the opcode in the bytecode stream. The code uses this index to retrieve an entry containing `methodOop`, index and flags from the constant pool cache. If this entry is not initialized, the interpreter calls into the runtime to update the constant pool cache. This possibly involves class loading. Following this, the code loads the receiver from the operand stack to the register `ecx` because a compiled target expects it there. Then the return address is pushed onto the top of stack, the call's target method is computed (dynamic dispatched via the class' method table), the invocation count is increased, the last stack pointer is saved and finally the code continues execution at the target (the `from_interpreted` entry in the `methodOop`). The target `methodOop` is stored in `ebx` because the interpreted method entry point expects it there. Figure 4.9 illustrates how the stack changes during this step when executing the `ListElem.accLen` method. Only stack slots that change are drawn.

Static calls `invokespecial/invokestatic` proceed similar but omit the loading of the receiver. The computation of the target uses the methodOop directly to get to the `from_interpreted` entry.

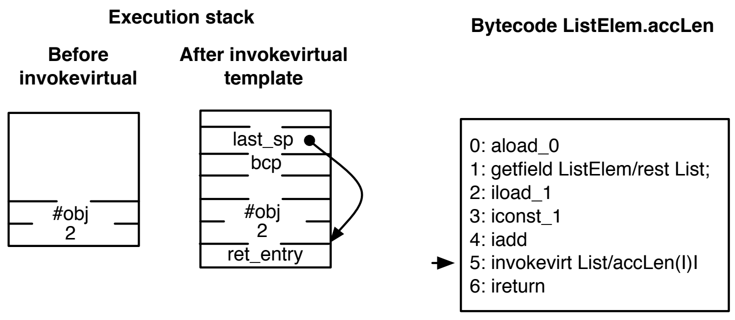


Figure 4.9: Execution stack of interpreter at the invocation of a method

Entry of a method The method entry point template in the interpreter is called either from another interpreted method or from compiled code. The `methodOop` of the method to be executed was stored by the caller in `ebx`. Parameters of the method are on the top of the stack. The following code shows the content of the assembler template.

```
method_entry:
  pop(return_addr);
  extend_params_with_locals();
  push(return_addr);
  generate_fixed_frame(current_thread_disable_osr());
  bang_shadow_pages();
  dispatch_next();
```

The template first moves the return address from the stack to a register. Then the parameters are extended by the required space for local variables. After this, the return address is pushed back to the stack. In the next step the template generates the frame by:

- Storing the old frame pointer to the stack and setting the new one to the current stack pointer.
- Storing the old stack pointer to the stack, which is passed in `esi`.
- Reserving a stack slot for the last stack pointer.
- Saving the method, method data and constant pool cache pointer to the stack.
- Saving the pointer to the start of locals.
- Retrieving the bytecode pointer (address at fixed offset in `methodOop`) and storing it to the stack.
- The value of the on stack replacement stack slot is set depending on whether the caller has passed a flag on the current thread (the `JavaThread` object) to disable on stack replacement for this method. This is added to support tail calls.

Then a check is emitted whether the stack overflows. The memory area at the end of every thread's execution stack is protected. The interpreter emits an access to the memory area above the current stack frame. If this access happens in the protected memory area the operating system signals an exception to the VM. The VM can then gracefully handle the stack overflow because the error happens at a defined point in the program. This process is called *stack banging.* Finally the bytecode pointer is stored to `esi` and the first bytecode is dispatched via the template table. Figure 4.10 (a) shows how the execution stack changes during this phase. It uses the recursive call of `ListElem.accLen` as example and only shows the stack slots that are relevant for the stack frame handling.

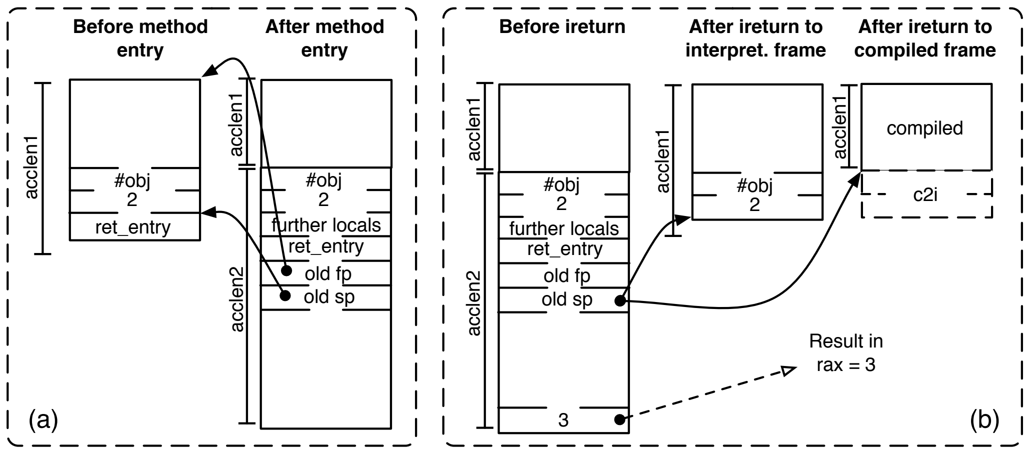


Figure 4.10: Execution stack during method entry and at the exit of a method

Exit of a method When a method is finished it calls one of the `return` bytecodes. The corresponding template gets executed.

```
ireturn:
  safe_result_from_stack_to_register(eax);
  movptr(ebx, Address(ebp, frame::interpreter_frame_old_sp_offset); // Get sender sp.
  leave(); // Remove frame anchor. esp=ebp; pop(ebp);
  pop(ret_addr); // Get return address.
  mov(esp, ebx); // Set sp to old sp.
  jump ret_addr;
```

The code first safes the result value from the top of the operand stack to a register. Next it retrieves the old stack pointer stored in the current frame. Then it removes the current frame and retrieves the return address in the caller. Finally, it sets the current stack pointer to the value of the saved old stack pointer and jumps to the return address. The usage of the old stack pointer allows compiled-to-interpreted adapter code to insert space between a compiled and an interpreted frame for the shuffled parameters. The adapter remembers the original stack pointer of the compiled frame in `esi` and then puts the shuffled parameters on the stack. On entry the interpreter safes the value of `esi` as old stack pointer.

Figure 4.10 (b) shows how the stack changes during this step with two possible return variants. The first shows the return to an interpreted method. The second shows the return to a compiled method. By using the old stack pointer, the interpreter guarantees that the code in the compiled method encounters the correct stack pointer.

Return from a method After a called method returns, execution continues in the return entry template of the interpreter. It removes the parameters and continues execution at the next bytecode.

```
return_entry:
  movptr(esp, Address(ebp, frame::interpreter_frame_last_sp_offset));
  restore_bcp();
  remove_method_parameters();
  dispatch_next();
```

The code first restores the stack pointer (top of operand stack) from the stack slot, where it was safed before invocation of the method. Then the bytecode pointer is reset to the state before the call. Next the parameters on the operand stack are removed. The interpreter uses the constant pool cache to get the number of parameters of the just invoked method. Finally the next bytecode template is dispatched.

Figure 4.11 demonstrates this step. Notice how a possible interpreter-to-compiled area is removed using the saved last stack pointer.

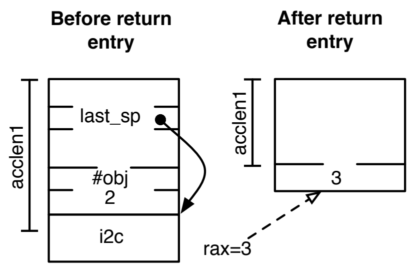


Figure 4.11: Execution stack at the return entry to the interpreter

Tail call additions To support tail calls we added the different `wide_invoke...` templates. Those templates contain the same code like normal invocation templates, as well as code that moves the parameters and removes the current stack frame.

```
wide_invoke_virtual:
  safe_curr_bytecode_pointer();
  // Load methodOop, vtable index and flags (contains number of parameters).
  load_invoke_cp_cache_entry(methodOop, index, flags);
  load_receiver(ecx, esp, flags); // Load the receiver from stack to ecx.
  push_return_adress(Interpreter::return_address_entry);
  klass = receiver.type;
  targetMethodOop = compute_receiver_target(klass, index);
  profile_call(targetMethodOop);
  if (protection_domain_mismatch()) {
    // Regular call or throw exception depending on flag.
    regular_call_continuation:
      safe_last_sp();
      jump targetMethodOop.from_interpreted;
  } else {
    tail_call();
    jump targetMethodOop.from_interpreted;
  }
```

The `protection_domain_mismatch` function checks, whether caller and callee protection domain are equal using the methods' classes. If they are not, the call is executed like a normal method call or an exception (`TailCallException`) is thrown depending on a flag the VM was started with. The `tail_call` function expands to code that performs the tail call.

```
tail_call:
  parent_is_not_interpreter_jcc(regular_call_continuation);
  safe_return_addr();
  safe_old_frame_pointer();
  safe_old_stack_pointer();
  move_parameters_from_top_of_stack_to_start_of_locals();
  store_return_addr_after_moved_parameters();
  esi = safed_old_stack_pointer;
  ebp = safed_old_frame_pointer;
  esp = address_of_stored_return_addr;
```

`parent_is_not_interpreter_jcc` checks whether the caller of the current frame is interpreted. If it is not interpreted, the call proceeds like a normal call. This is done because of two reasons. First, the compiled code needs an interpreted frame for certain types of tail calls. It creates those frames by calling into the interpreter. If the called method (running in the interpreter) does a tail call that removes the current frame, this undos the intentions of the compiled code and no interpreted frame is on top of the execution stack. The second reason has to do with compiled-to-interpreted transitions. Assume that the current method is called from a compiled method. The current method is interpreted. Hence there is a compiled-to-interpreted adapter on the stack. The current method performs a tail call to a compiled method. Without the check the current interpreter frame is removed and there is no code executed that removes the adapter area resulting in wrong stack pointers. Figure 4.12 illustrates the problem.

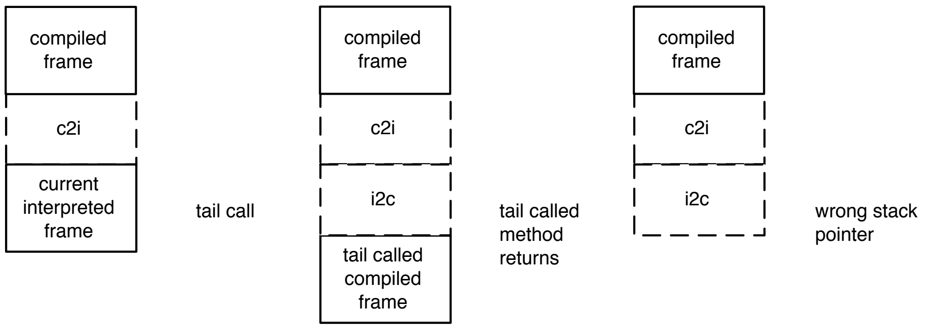


Figure 4.12: Wrong stack pointer after removal of interpreter frame

After the code has determined to perform a tail call, it safes the return address, old stack/frame pointer from the current stack frame to the top of the stack. Then it moves the parameters from the operand stack to beginning of the current interpreter frame where the local area starts. Next it places the safed return address after the moved parameters. It sets `esi`,`ebp` equal to the old stack and frame pointer. Then it sets the current stack pointer `esp` equal to the address of the moved return address slot. The stack now looks as if the caller's caller has invoked the called method. Finally, the code jumps to the target method. Figure 4.13 illustrates how the stack changes during the execution of the tail call code.

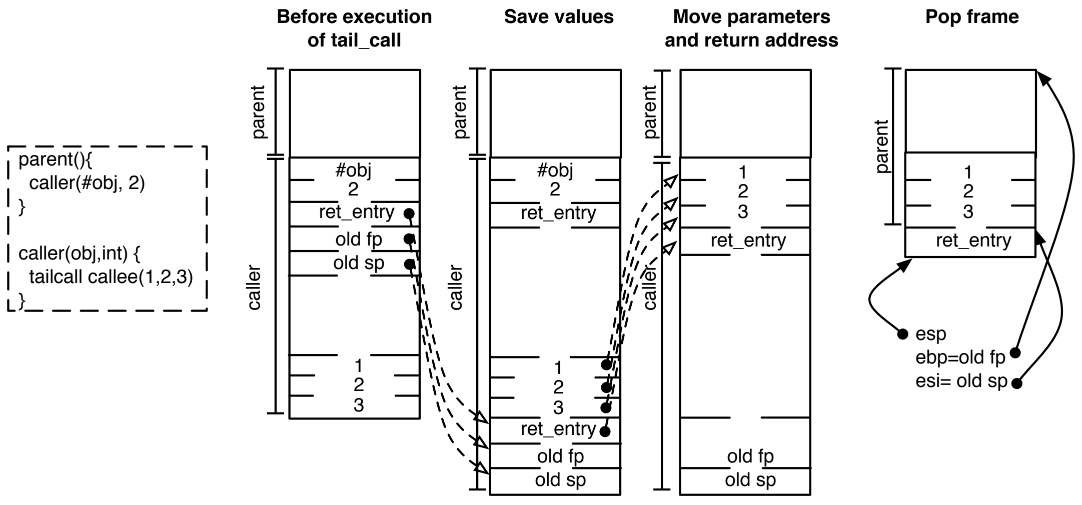


Figure 4.13: The stack changes during an interpreter tail call.

Compiled code sometimes calls a method in interpreted mode to create an interpreted stack frame. The interpreter might do on stack replacement, if the called method contains a long running loop. To prevent the called method from performing on stack replacement, the compiled code sets a flag on the current thread to disable on stack replacement. The method entry of the interpreter stores this fact to the current stack frame. The assembler template for the `branch` instruction is modified to include a check whether on stack replacement is turned off for this frame. This check inspects the value of the OSR stack slot.

4.8 Compiler

The compiler (client or server) generates code for a whole method. A method call is viewed as a unit during compilation, i.e. for every invocation a certain combination of machine instructions is emitted. These instruction only depend on the type of the method call. The call site can be considered in isolation. A call consists of:

- Lowering of arguments: Compiled methods have a fixed frame size, which includes outgoing parameters. The compiler emits code that moves the parameters to their stack slot in the current frame's outgoing parameter area.
- Actual call site: The call site sets possible tokens (security, class) and performs the call to a target.

Figure 4.14 shows how a method invocation is treated as a unit during compilation by the client compiler.

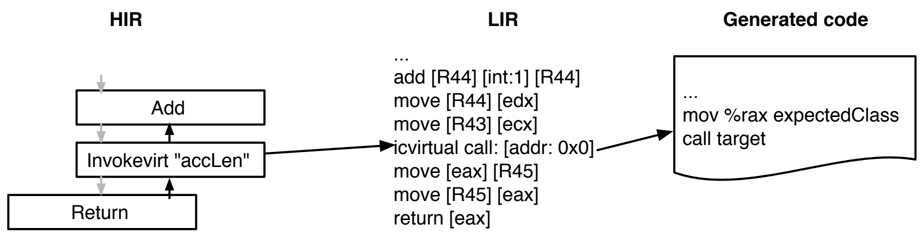


Figure 4.14: A method invocation is treated as a unit during compilation.

The tail call implementation uses the same form for a tail calling method invocation. Only the actual target of the call is different.

4.8.1 Sibling and Non-Sibling Tail Calls

A tail call might be calling from a method with few parameters to a method with many parameters. Because the parameters are moved onto the caller's caller frame, the tail call needs to make sure that there is enough space. If a tail call happens from a method that requires n stack slots for its parameters to a method that needs $\leq n$ stack slots, we can be sure that there is enough space in the caller's caller because it was reserved when calling the caller. Because stack frames have a fixed size, if the callee requires more than n stack slots, we cannot move the parameters to the caller's caller frame. Figure 4.15 shows the problem. Method invocations that require $\leq n$ stack slots are called *sibling tail calls*, invocations that need $\geq n$ stack slots are referred to as *non-sibling tail calls*.

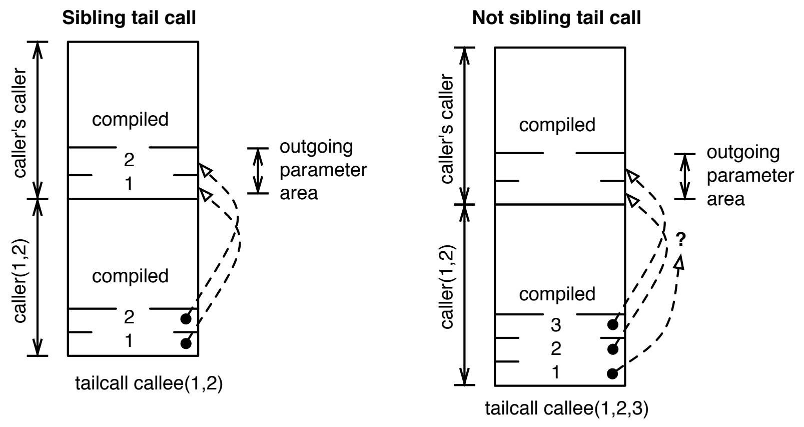


Figure 4.15: Sibling and non-sibling tail calls

This problem can be solved in two ways:

- Making the stack frame size variable.
- Insert an adapter frame that has variable size.

To make a compiled stack frame variable in size, every access to a value in the stack frame has to use the frame pointer. This means that the frame pointer is reserved for this purpose. The assumption that the size of compiled stack frames is constant is spread around the code in the whole VM (e.g. stack walking, oop maps and debugging information use the stack pointer to refer to locations in the stack frame, server compiler does not use/set the value of the frame pointer). Making the stack frame size variable means changing many different components of the VM. It also negatively effects performance of methods in the server compiler, which uses the frame pointer as normal assignable register during register allocation.

Therefore the compiler uses the second method. The compiler differentiates between sibling and non-sibling calls. A sibling tail call jumps to a method entry that moves the arguments onto the caller. A non-sibling tail call method entry checks if the parent (caller's caller) frame is extendable, i.e. an interpreted frame. An interpreted frame is extendable because it restores its stack pointer from the last stack pointer slot. If the parent is extendable, the non-sibling tail call method entry can extend the frame and move the parameters onto it. If the parent is not interpreted, the method entry starts execution of the called method in the interpreter. This creates an interpreted and extendable frame on the stack. If this is a series of tail calls the next tail call has a parent frame that is interpreted. Figure 4.16 illustrates the two possibilities.

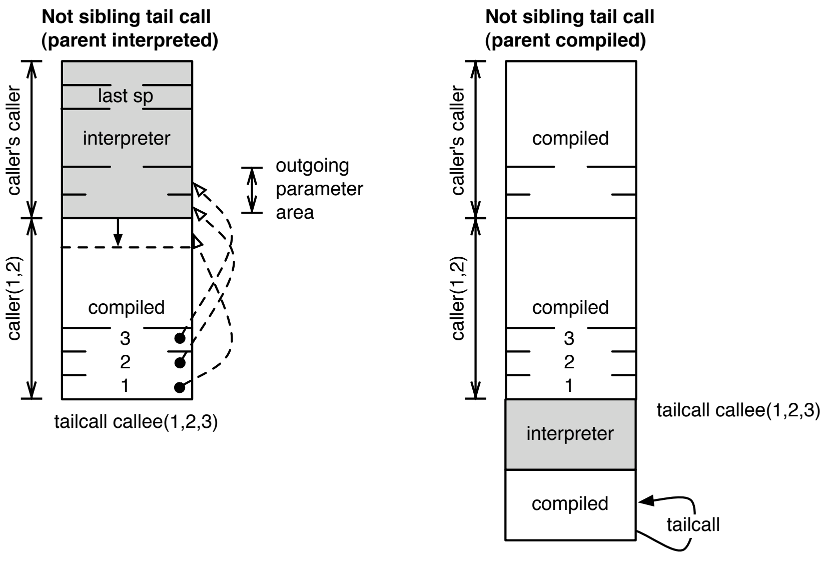


Figure 4.16: Handling of non-sibling tail calls

This method incurs some overhead for non-sibling calls because the program executes in the interpreter for one method execution. To prevent this from happening, the modified JVM features a flag `MinOutgoingArgsSlotSize` with which the user can set the minimum stack slot size for the outgoing area of every method. Every method allocates this many outgoing stack slots. Hence every call that requires less or equal many stack slots for parameters is a sibling tail call and does not incur any overhead.

4.8.2 Call Sites in Compiled Code

The call site generated by the compiler, provided it did not inline the call, can have two forms:

- Static or optimized virtual calls: The compiler knows that there is only one possible target of the call. `invokestatic` and `invokespecial` always have a known target method. If the compiler can determine during compilation that a virtual method call can only have one target, it also emits a static call.
- Virtual call: There might be multiple different targets depending on the receiver type. The compiler emits code that can handle them.

Figure 4.17 shows a high-level overview how the compiler generates the call site depending on the invocation type and the possible receiver types.

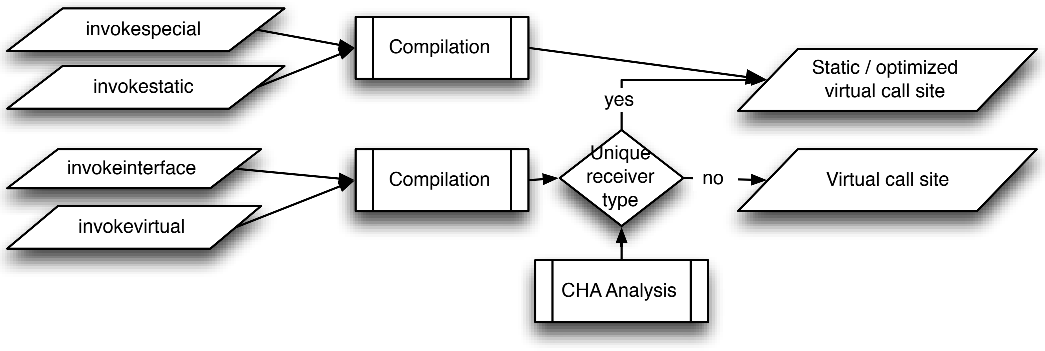


Figure 4.17: The compiler generates call sites depending on the possible receiver types

Static and Optimized Virtual Calls

A static or optimized virtual call is emitted, if the call target, i.e. the `methodOop` of the call, is known to be unique. The following assembler code is emitted.

```
call resolve_static_call_runtime_entry
```

The call goes to a runtime entry that patches the call with the appropriate target, the first time the call site is executed. The patched target is the address stored in the `from_compiled` entry in the `methodOop`. Depending on whether the method is compiled at the time of patching, this entry either contains the verified entry point of the compiled method or the address of a compiled-to-interpreted adapter.

Depending on the current target, the call site can be assigned a state. This state can change during execution of the program because a method is compiled or recompiled. Figure 4.18 shows the different possible states.

When a method is compiled the call sites' target addresses point into the runtime. The first time the call site gets executed the runtime replaces the target address of the call with the address of the called method. The runtime uses the `methodOop` to determine the target of the call. If the compiler has already compiled the method the `from_compiled` contains the verified entry point of the compiled method.

```
call methodOop.VEP
```

If there is no compiled code the runtime `from_compiled` entry contains a compiled-to-interpreted adapter, which shuffles the arguments and continues execution in the inter-

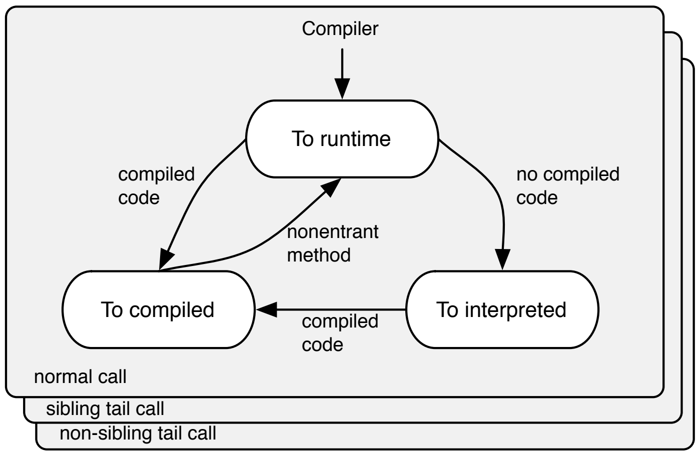


Figure 4.18: The different states of a static call site

preter. Because the compiled-to-interpreted adapter expects the `methodOop` in register `ebx`, the runtime does not patch the address of the adapter but instead uses the address of a stub that is generated for every call that sets the value in `ebx` first before jumping to the adapter.

```
  call to_interpreter_stub
  ...

to_interpreter_stub:
  mov %ebx methodOop
  jmp c2i_adapter
```

It is possible that a method is compiled, after the runtime has set the target of that method to interpreted state. Therefore the interpreter adapter checks whether the called method is compiled, before it continues running the method in interpreted mode. If there is compiled code, it fixes the call site, so that the next time the call site is invoked, it goes to the compiled code.

The compiler may recompile methods. This makes the old version of the compiled method invalid. The compiler marks the old methods as non-entrant by patching the entry points of the methods with code that goes into the runtime. Compiled call sites still point to those entry points. The next time the compiled call site executes, it arrives at the patched entry point. Execution continues in the runtime. The runtime fixes the target of the call site to point to the runtime entry responsible for resolving the call site and continues execution in the new method.

To handle tail calls, conceptually another dimension is introduced. A call can either be a normal call, a sibling tail call or a non-sibling tail call. We can imagine that for the tail calls two more layers containing the same states are introduced as show in Figure 4.18. Depending on the type of call: normal call, sibling tail call or non-sibling tail call we are in one of the three layers. The transitions stay the same but can only happen in one layer. The layers are reflected in the VM by different entry points. The compiler emits the initial runtime entry, which either points to the function `resolve_static_call`, `resolve_static_tail_call` or `resolve_not_sibling_static_tail_call` depending on the type of call. These functions use the respective method entry `from_compiled`, `from_compiled_tail_call` and `from_compiled_no_sibling_tail_call` to patch the call site.

The protection domain check happens during the execution of `resolve_static_tail_call`. If caller's and callee's protection domain mismatch, the call is not patched to go to the entry that removes the frame `from_compiled_tail_call` but instead to the normal method entry `from_compiled`. Because protection domains are only set during class loading and not modified thereafter, it is sufficient to check the correspondence once during resolving the call target.

Virtual Calls

If the compiler can not determine that there is only one possible receiver type, e.g. if both a class and a subclass implement a method or an interface is implemented by several classes, it emits a virtual call site. The call site can handle monomorphic as well as polymorphic calls.

```
move %eax NULL  // Class token for monomorphic call.
move [esp+0] NULL // Security token for polymorphic calls
call resolve_virtual_call
```

Figure 4.19 shows the different states a virtual call site can be depending on the target of the call.

Initially the compiler emits code that calls into the runtime. When the call site is called for the first time the program continues its execution in the respective runtime resolve function. This function sets the state of the call site to compiled monomorphic by setting the class token to the current receiver's class and by patching the call target to be the unverified (class check) entry point of the receiver method.

```
move %eax instanceKlassOop // Class token for monomorphic call.
move [esp+0] NULL // Security token for polymorphic calls
call unverified_entry_point
```

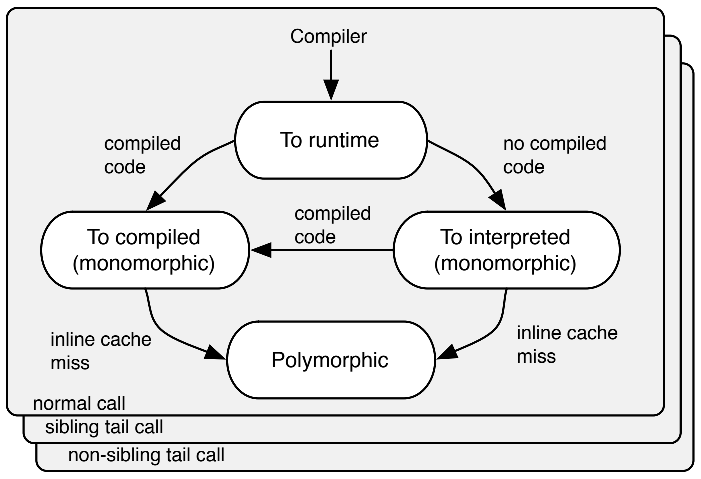


Figure 4.19: The different states of a virtual call site.

If the receiver's method is not compiled, the runtime sets the class token to a `compiledICHolderOop`. This data structure stores the expected class and the `methodOop` of the receiver method. The target of the call is set to a special compiled-to-interpreted adapter entry.

```
move %eax compiledICHolderOop // Class token for monomorphic call.
move [esp+0] NULL // Security token for polymorphic calls
call c2i_unverified_entry_point
```

This entry uses the information in the `compiledICHolderOop` to check whether the actual receiver type matches the expected one and continues execution in the interpreter using the `methodOop`. The code in the adapter also checks, if a compiled version of the receiver method exists and calls into the runtime to fix the call site if it does.

If the inline cache check fails at the unverified entry point of a compiled method or in the adapter control transfers into the runtime. The runtime than patches the target of the call site to go to a stub that does the dynamic dispatching. The `move %eax` instruction is updated so it contains the `instanceKlassOop` of the interface if the call is an interface call or `NULL` if the call is a virtual call.

```
move %eax instanceKlassOop // EITHER:Class of interface for interface calls.
move %eax NULL             // OR:    Call is a invokevirtual.
move [esp+0] protection_domain // Security token for polymorphic calls
call dynamic_dispatch_stub
```

To support tail calls we introduce another dimension. A virtual call can be either a normal, sibling or non-sibling tail call. This is reflected in code by different entry points. For every normal call entry point there exists a sibling and non-sibling tail call version. The protection domain check for monomorphic call sites happens once, when the runtime patches the call site. Because monomorphic calls only have one possible receiver type this is sufficient. The dynamic dispatch stubs for tail calls are extended to include a protection domain check. Every time the dispatch code is entered it checks whether the protection domain of the caller (which was set at the call site, see code above) matches the protection domain of the receiver type. If this check fails, the code jumps to the normal method entry else it jumps to a tail call method entry.

Relocation Information

Compiled code may contain object pointers (oops) to runtime objects. When garbage collection happens the collector needs to know the location of those pointers. Generated code might move and targets of jumps and calls have to be updated. The runtime needs to know the type of a call during patching. To support those operations the compiler maintains *relocation information* objects, which are stored together with the compiled code in the `nmethod` object. These objects conceptually point to an address in compiled code and associate meta information with it. There are several types depending on the information they should convey. The following listing contains types that are relevant for this work.

- Oop type: The address pointed to is an object pointer. A virtual call site contains such a relocation info for the class pointer.
- Virtual call type: It marks a virtual call site.
- Optimized virtual call type: It marks an optimized virtual call site.
- Static call type: It marks a static call site.

To support the differentiation of the call site, a sub type that specifies the kind of call is added. The relocation information for calls contains an extra field that specifies whether a call is a normal call, sibling tail call or a non-sibling tail call. Figure 4.20 shows the relocation information generated by the compiler at a virtual tail call site.

Non-entrant Methods

If the compiled code of a method becomes invalid a method might be recompiled or has to be executed by the interpreter. The code becomes invalid, if it contains an optimized virtual call and loading of a new class invalidates the assumption that this call only has one receiver type. Static call sites in other methods might still point to the code of the old method. To prevent such call sites from entering the wrong code, invalid methods are marked as non-entrant. The runtime patches the first instructions of the different

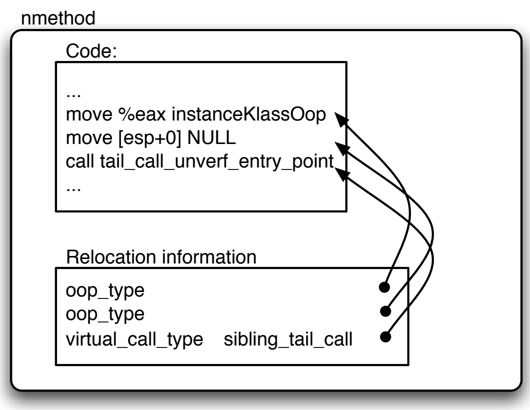

nmethod
Code:
...
move %eax instanceKlassOop
move [esp+0] NULL
call tail_call_unverf_entry_point
...
Relocation information
oop_type
oop_type
virtual_call_type sibling_tail_call

Figure 4.20: Relocation information of a virtual tail call site

entry points to the old method with calls back into the runtime. The target of that call is the runtime method `handle_wrong_method`. This method fixes the target of the static call site to a runtime entry that resolves the call. The next time the static call site is executed the target is fixed to point to the appropriate method entry of the new method. Figure 4.21 illustrates this process.

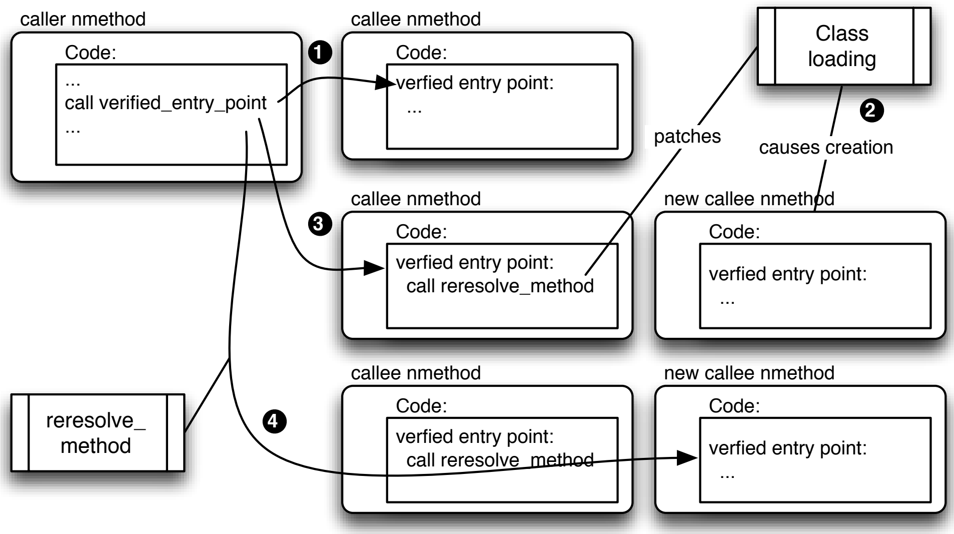


Figure 4.21: Marking methods non-entrant causes static calls to be re-resolved.

To support tail calls the tail call entry points also have to be marked as non-entrant if a method becomes invalid.

4.8.3 Resolving a Call

When the compiler emits code for a call, the target of the call points to one of the resolving methods.

- `resolve_static_call`, `resolve_optimized_virtual_call`: The runtime entry resolves the call target to point either to the callee method's verified entry or to the method's compiled-to-interpreted adapter (via the to-interpreter stub).
- `resolve_virtual_call`: Resolves the call target to the unverified entry points and sets the class token.

To support tail calls, versions for sibling and non-sibling tail calls are added, e.g. there is a `resolve_static_tail_call` and a `resolve_static_not_sibling_tail_call`. All these entry functions call the function `resolve_sub_helper`, which implements their behaviour. The parameters to the function indicate the type of the call. We modified the function to perform the following two steps.

- Protection domain check. This step only happens if the call is a tail call. The runtime compares the protection domain of the caller with the protection domain of the callee. The protection domain is stored in the `instanceKlassOops`, which hold the respective methods. If the protection domains are not equal, the runtime either throws an exception or the following call site patching patches the call to a normal call entry instead of to a tail call entry.
- Compute the method entry. The runtime computes the entry point in the method, which is stored in either the `methodOop` or in the corresponding `nmethod` if the callee is already compiled. The resulting entry point depends on whether the call is static, optimized virtual or virtual and on whether the target method is compiled or not. With the introduction of tail calls, the call sub type also needs to be taken into account. The following table shows which entry is received in this step.

	Normal call	Sibling tail call	Non-sibling tail call
Static call site Callee is			
interpreted	c2i_entry	c2i_tail_call_entry	c2i_not_sib_tail_call_entry
compiled	verified_entry	verified_tail_call_entry	verified_not_sib_tail_call_entry
Virtual call site Callee is			
compiled	unverified_entry	unverif_tail_call_entry	unverif_not_sib_tail_call_entry
interpreted	c2i_unverified_entry	c2i_unverif_tail_call_entry	c2i_unverif_not_sib_tail_call_entry

4.8.4 Dispatch Stubs

The VM supports virtual calls through the use of dynamic dispatch stubs. When a call site is set to polymorphic state, the address of the call is set to such a stub. For tail calls a protection domain check is introduced.

```
dispatch_stub:
  target = ... // Interface or vtable lookup.
  method_entry_offset = from_compiled_tail_call_offset();
  if (callerClass.protection_domain != target.method_holder.protection_domain)
    throw TailCallException; // OR:
    method_entry_offset = from_compiled_offset();
  jump target[method_entry_offset];
```

Depending on the kind of the call, i.e. normal call, sibling tail call or non-sibling tail call, the dispatch stub jumps to the

- `from_compiled`
- `from_compiled_tail_call`
- `from_compiled_not_sibling_tail_call`

entry point in the `methodOop`.

4.8.5 Frame Layout and Calling Convention

Every time a method is called, a stack frame is allocated on the execution stack. Stack frames of compiled methods have a different layout than their interpreted counter parts. A compiled stack frame is more compact. It contains the following stack slots:

- Return address: The address in the caller method where to resume execution if the current method returns.
- Dynamic link: Normally contains the caller's frame pointer. The server compiler uses the frame pointer register as a regularly assignable register.
- Monitor area: The data structure used for locking objects is stored here. The size depends on the number of locked objects. For every locked object there is an entry of two words containing the object's address and header word.
- Spill area: The area where the register allocator temporarily stores registers, if an operation requires more registers than are currently available.
- Outgoing parameter area: The compiler reserves an area at the end of a frame, which is large enough to hold the parameters for any of the called methods.

When a method is called, not all parameters are passed on the stack. The compiler uses register `ecx` and `edx` to pass integers or object pointers. If the method call is virtual, `ecx` contains the address of the receiver object. For floating point values register `xmm0` and `xmm1` are used. If the method has more parameters than fit in aforementioned registers, they are passed on the stack. To support passing of the protection domain token the first parameter slot is reserved. The method that computes the location of parameters is modified so it assigns real parameters above the reserved stack slot. Figure 4.22 shows the layout of a compiled stack frame.

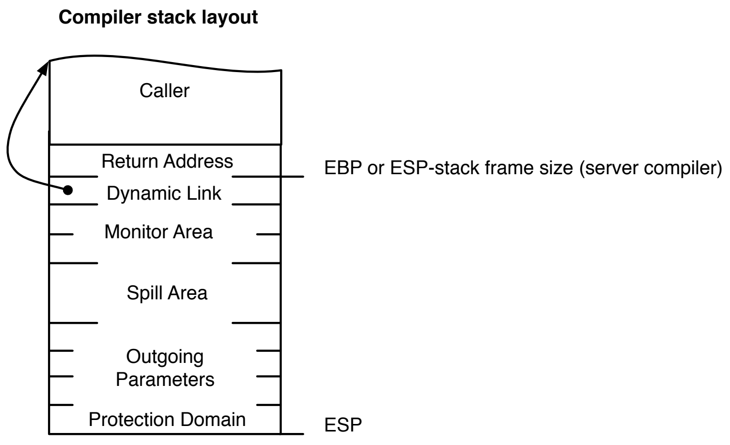


Figure 4.22: Stack frame layout of compiled frames

Tail call entry points need the size of the tail calling frame in order to be able to remove it. Code compiled by the client compiler always uses the register `ebp` to hold the current frame pointer. When a tail call entry point is entered, the code can use the value in `ebp` to remove the calling frame. The server compiler uses `ebp` as a normally assignable register and it assumes that a method call does not destroy its contents, i.e. the callee safes the content on entry and restores it on return. Another register has to be used to indicate the extent of the calling frame. The call site code of tail calls, emitted by the server compiler, stores the value of the stack pointer `esp` minus the frame size in `esi`. The server compiler's tail call entry points use this value to remove the frame.

4.8.6 Method Entry Points

A virtual call site can be in monomorphic state, which requires a check in the method entry of the callee. Tail calls need to move arguments and remove the caller frame at method entry. The VM supports this by having different entry points to a compiled method. The entry point's addresses are stored in the `nmethod` object together with the method's code. The following is a listing of all entry points together with the code that is executed, when the entry point is entered.

- Verified entry point: The verified entry point corresponds to a normal method entry. At first the code writes to memory area below the current stack pointer. The memory at the end of every threads execution stack is write protected. If a stack overflow occurs, the instruction fails and a signal is sent to the VM by the operating system. Execution continues in the VM's runtime, which throws a StackOverflow exception. Next the code safes the old frame pointer, sets the new frame pointer and creates the new stack frame. At this point the frame setup is complete. The address at this point is stored, so that code generation can refer to it. Finally, execution continues with the method's code.

```
VEP:
 mov [%esp - bang_offset] %eax // Check whether the stack overflows.
 push %ebp                     // Safe the frame pointer.
 mov %ebp %esp                 // Set new frame pointer (only in client compiler)
 sub %esp new_frame_size       // Create new frame.
frame_complete_label:
```

- Unverified entry point: This entry supports monomorphic calls. An inline cache check is performed before execution continues at the verified entry point. If the check fails because the expected class does not match the receiver's class, the program continues in the runtime.

```
UEP:
 // Eax contains expected class. Ecx the receiver.
 cmp  %eax  [%ecx + oop::class_pointer_offset]
 jne  handle_wrong_method_ic_miss
VEP:
 ... // continue at verified entry point
```

- Verified tail call entry point: This is the target of sibling tail calls. The code moves the arguments from the outgoing parameter area of the caller (`esp` to `esp`+parameter size) to the outgoing parameter area of the caller's caller. The outgoing parameter area of the caller's caller starts two stack slots above `ebp` or `esi` if the server compiler is used.

```
VEP_tailcall:
 for (param = 1.. number of parameters)
   mov %ebx [%esp + (param)*wordSize]
   mov [%ebp + (param+1)*wordSize] %rbx
 leal %esp [%ebp - framesize] // Compute the new stack frame.
 jmp frame_complete_label
```

  Then it removes the caller's frame and creates the new frame by setting the stack pointer `esp` to the value of the caller's frame pointer `ebp`/`esi` minus the new frame size. Finally, the code jumps to the point, where the frame setup is completed in the verified entry point. This is the beginning of the method's actual code.

- Unverified tail call entry point: The code starts with an inline cache check. Then performs the same actions as the verified tail call entry point.

- Verified non-sibling tail call entry point: The code first checks whether the caller's caller frame is an interpreted frame. The code of the interpreter is emitted in a sequential memory area. The return address is inspected, whether it lies within this area.

  ```
  UEP_tailcall:
   mov %ebx [%ebp +wordSize]          // Return address
   cmp %ebx address_interpreter_start
   jl not_interpreter_continuation    // Continue if addr is above lower bound.
   cmp %ebx address_interpreter_end
   jg not_interpreter_continuation    // Continue if addr is below upper bound.
  ```

  If the check succeeds, the code moves the parameters to the end of the interpreted stack frame. It uses the interpreter's last stack pointer stack slot entry to determine the destination of the arguments. Because the stack slots containing the interpreter frame's return address and frame pointer is overwritten by the argument moving, those two values are stored to the top of the stack. After the arguments are moved the two values are restored from there.

  ```
   safe_ret_addr_and_old_frame_pointer(); // Argument move will write to stack slots.
   mov %eax [%ebp] // Get frame pointer of caller's caller (the interpreted frame).
   mov %ebx [%eax + last_stack_pointer_offset] // Get the last stack pointer.
   // Copy arguments starting with the lowest argument on the stack.
   for (int src_slot = arg_slots, dest_slot=-1; src_slot > 0; src_slot--, dest_slot--)
     // Saved old_ebp, old_retaddr on top of stack hence +2.
     mov %eax, [%esp + wordSize*(2+src_slot)]
     mov [%ebx + (wordSize*dest_slot)] %eax
   Move return address to new place below move arguments.
   Restore the safed old frame pointer in %ebp.
   jmp verified_entry_point
  ```

  Figure 4.23 illustrates what happens during the argument move. The interpreter frame is extended. Because the interpreter restores the top of the operand stack using the last stack pointer on return to the method, this extension is legal. After the arguments are moved and the frame pointer restored the code continues at the verified entry point. If the parent frame is not interpreted the code continues execution of the called method in the interpreter. The caller frame is not removed.

  ```
  not_interpreter_continuation:
   mov %ebx methodOop    // The interpreter expects the current method in %ebx.
   jmp c2i_adapter_entry // Continue execution in the interpreter.
  ```

  In a series of tail calls the subsequent calls have a parent frame that is interpreted.

- Unverified non-sibling tail call entry point: The code starts with an inline cache check. Then it performs the same actions as the verified non-sibling tail call entry point.

The server compiler generates slightly different code for tail calls. The call site in server compiled code stores the caller's frame pointer in register `esi`. Instructions that use the caller's frame pointer use this register instead of `ebp`.

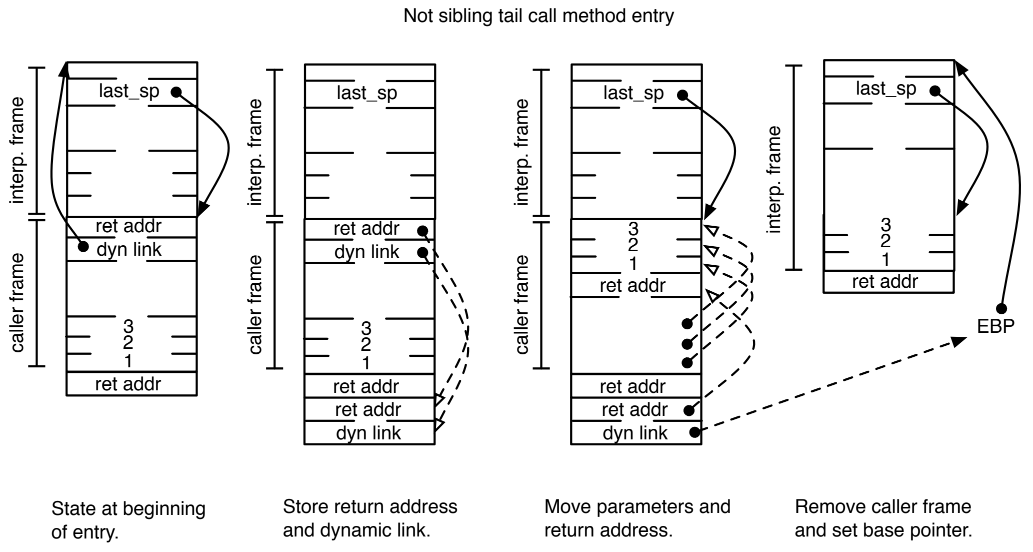

Not sibling tail call method entry
interp. frame
caller frame
last_sp
ret addr
dyn link
3
2
1
ret addr
EBP
State at beginning of entry.
Store return address and dynamic link.
Move parameters and return address.
Remove caller frame and set base pointer.

Figure 4.23: State of the stack during a non-sibling tail call (caller's caller is interpreted)

4.8.7 Compiled to Interpreted Transitions

Interpreted and compiled code uses the same execution stack. Because interpreted frames have a different argument layout than compiled frames, whenever a transition from one to the other happens, the parameters on the stack have to be rearranged. The VM uses special adapter code to do the shuffling. The code is stored in an instance of the class `AdapterHandlerEntry`.

Every `methodOop` has a pointer to an `AdapterHandlerEntry` object, which contains the code for shuffling the methods arguments. The `methodOops` entry point `from_interpreted` or `from_compiled` points to an entry point in the adapter, depending on whether the method is compiled or not. Adapters are shared among many methods. If two methods have the same type signature for their arguments, they share the same adapter object, e.g. the static method `int a(int, int)` and the static method `float b(int, int)`.

Adapters create an area on top of the current stack frame, where they move the rearranged arguments. They are called instead of directly calling the interpreter or a compiled method. There are two kinds of adapters corresponding to the direction of the transition.

- Interpreted-to-compiled (i2c) adapter: The code in the adapter rearranges the arguments according to the compiled calling convention, e.g. the first two integer or object arguments are moved to the registers `ecx` and `edx`. Then it continues at execution at the method verified entry point. On return, the interpreter frame that called the adapter, restores its stack pointer using the last stack pointer slot value. Hence the area created for the arguments is removed. Figure 4.24 (a) illustrates the execution stack during the transition. The adapter area is drawn in grey.
- Compiled-to-interpreted (c2i) adapter: The code in the adapter rearranges the arguments according to the interpreted calling convention, i.e. arguments that reside in registers are put onto the stack. Before the area for the arguments is created, the code stores the original stack pointer in register `esi`. The interpreter saves this value on entry of a method as old stack pointer. When the interpreted frame returns, it uses the old stack pointer to remove the extra parameter area. The compiled frame sees the correct stack pointer. After the adapter has moved the arguments, it jumps to the method's interpreter entry. Figure 4.24 (b) illustrates the execution stack during the transition.

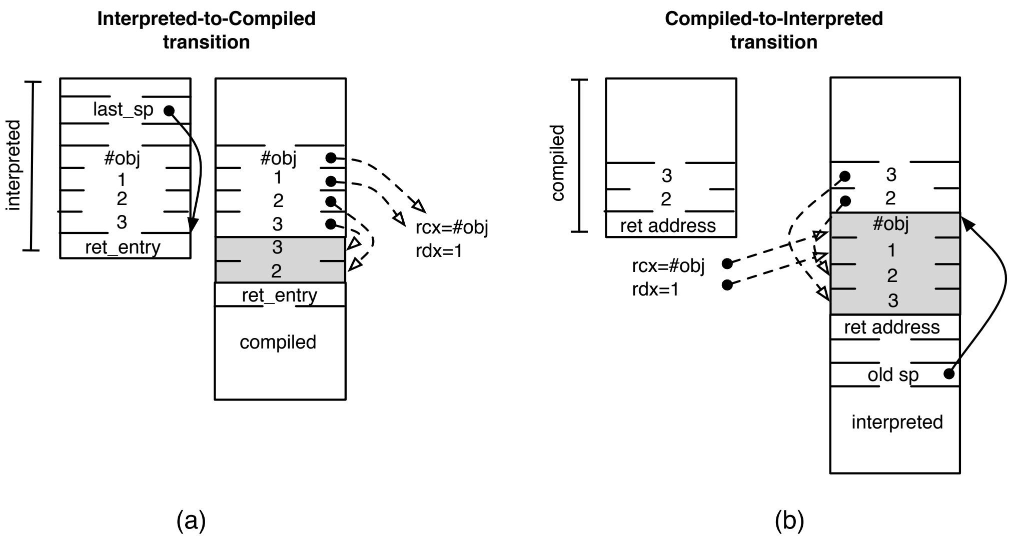


Figure 4.24: Interpreted-to-compiled and compiled-to-interpreted transitions

An `AdapterHandlerEntry` has several entry points. There is one entry point that holds the i2c adapter. A method invocation in the interpreter always computes the correct target of the call. The target's `methodOop` is stored in register `ebx` when the interpreter executes the template for the invocation. The i2c adapter code can always assume that the `methodOop` is correct. If a method invocation is a tail call, the interpreter removes the caller frame before continuing at the target method. Hence there is only one i2c adapter needed. Compiled call sites that are in monomorphic state make an assumption about the target `methodOop`, which has to be checked before entering the interpreter.

For static or polymorphic call sites this check is not necessary. Tail calling method invocations in compiled code remove the calling frame at the callee's method entry, c2i adapters need to duplicate this behaviour before entering the interpreter. Hence every method has multiple c2i adapter entries.

- C2i_entry point: This entry is called either from a static call site or from a polymorphic virtual call site. The `methodOop` that the call site stored in `ebx` can be assumed to be correct. The code in the entry point performs the following steps.
  - Patch caller's call site: Static call sites might still target interpreted code, although there is compiled version. The adapter checks whether there is compiled code. If a compiled version exists, the code calls the runtime to patch the call site to point to the method's verified entry point.
  - Create adapter area: The code creates an area on top of the stack to hold the parameters according to the interpreted convention and moves the arguments there. The return address on the stack is moved below the newly created area. The code passes the original compiled frame's stack pointer in register `esi` to the interpreter.
  - Jump to callee in interpreted mode: The adapter continues execution at the `methodOop`'s interpreter entry (`from_interpreted`).
- C2i_unverified_entry point: This entry is called from a monomorphic call site. The call site has stored a `compiledICHolderOop` in `eax`. It contains the expected receiver class and the callee `methodOop`. Before continuing in the interpreter the code needs to check that the actual receiver type matches the expected. The code performs the following steps.
  - Inline cache check: The code checks, whether the expected receiver class in the `compiledICHolderOop` matches the current receiver's class. If the check fails, program execution continues in the runtime's `handle_wrong_method_ic_miss`.
  - Patch caller's call site: If there is a compiled version of the callee available, the call site is reset to point to the runtime's resolve method. The next time the call site gets executed, the resolve routine patches the call site to go to the compiled method's unverified entry point.
  - Create adapter area: The arguments are moved and the original stack pointer is saved.
  - Jump to callee in interpreted mode.
- C2i_tail_call_entry point: This entry is called from either a static or polymorphic virtual call site that performs a sibling tail call. The `methodOop` passed in `ebx` contains the correct callee target. The code performs the following steps.

- Patch caller's call site: If there is a compiled version of the called method, the entry point patches the static call site with the verified tail call entry point of the callee's method.
- Move arguments onto caller's caller frame. Because the call is a sibling call, there is enough space in the caller's caller (caller's incoming parameter area). The code that performs the move is similar to the code at the method's tail call entry point, i.e. parameters are moved onto the caller's caller using the caller's frame pointer (see Section 4.8.6). If the caller's caller is interpreted, the arguments are moved to the interpreted-to-compiled argument adapter area.
- Remove the caller frame.
- Create adapter area on top of the stack (now the caller's caller) and move arguments according to the interpreted parameter passing convention.
- Jump to callee in interpreted mode.

Figure 4.25 shows how the execution stack changes when executing this adapter.

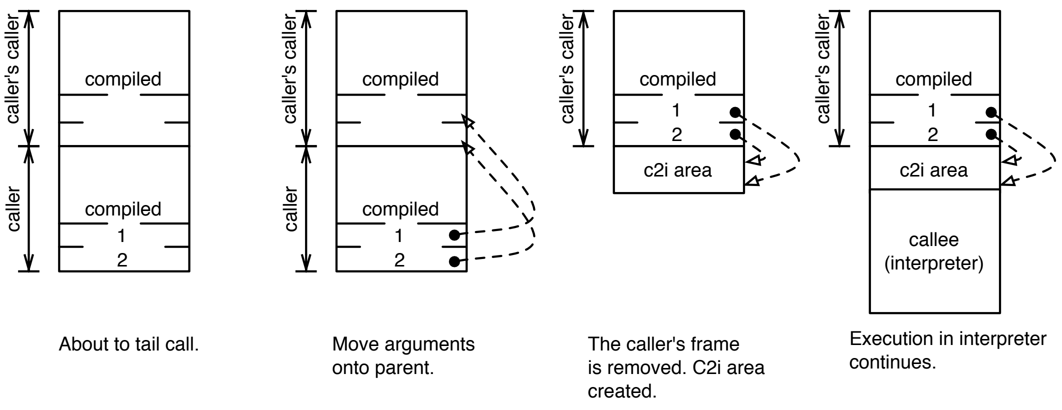

caller's caller
caller
compiled
compiled
1
2
About to tail call.
Move arguments onto parent.
c2i area
The caller's frame is removed. C2i area created.
callee (interpreter)
Execution in interpreter continues.

Figure 4.25: Stack during execution of the c2i tail call adapter runs

- C2i_unverified_tail_call_entry point: This entry is called from a monomorphic call site that performs a sibling tail call. The expected and actual receiver types have to be checked, before the caller frame is removed. The code performs the following steps.
  - Inline cache check.
  - Patch caller's call site if a compiled version of the callee exists.
  - Continue at c2i_tail_call_entry point after the patching instructions.
- C2i_not_sibling_tail_call_entry point: This entry is called from a static or polymorphic virtual call site that performs a non-sibling tail call. Because the tail call is not a sibling tail call, it is not guaranteed that the caller's caller frame has enough

room for the parameters. Similar to the method's non-sibling entry points the caller's caller frame has to be an interpreted frame, so it can be extended to hold the parameters. The code performs the following steps.

  – Patch caller's call site. The static call site is patched with the verified non-sibling tail call entry point if compiled code exists.
  – Check if caller's caller (parent) is interpreted. If it is, the arguments can be moved to the extended stack frame using the parent's last stack pointer. The code is similar to the code in the method's non-sibling tail call entry (see Section 4.8.6). If the parent frame is not interpreted the code continues at the c2i_entry point. The caller frame is not removed. In a series of tail calls the next tail call therefore has an interpreted parent frame.
  – Create adapter area (now on top of the caller's caller) and move arguments.
  – Jump to callee interpreted mode.

- C2i_unverified_not_sibling_tail_call_entry point: This entry is called from a monomorphic call site that performs a non-sibling tail call. The expected and actual receiver types have to match, before execution continues performing the tail call. The following steps are performed.
  – Inline cache check.
  – Continue at the c2i_not_sibling_tail_call_entry after the patching instructions.

4.8.8 Client Compiler

We modified the client compiler, so that it emits call sites that point to the corresponding resolve runtime method, e.g. if it is a static sibling tail call, the site calls to `resolve_static_tail_call`. Therefore changes in the HIR and LIR invocation instruction handling are necessary. To support tail calls a compiled method has additional entry points that perform the argument shifting and removal of the caller frame. The code of those entry points is stored at the end of the compiled method after the exception handler code stubs.

HIR

The HIR represents a method invocation instruction by an instance of the class `Invoke`. It stores the bytecode, the instruction computing the receiver, the instructions computing the arguments, the signature, a vtable index and a pointer to an object that represents the target method. Every HIR instruction is a subclass of the `Instruction` class, which contains among other things an integer value that stores a set of flags. To support marking a call as tail call a `TailCallFlag` is added. The constructor of `Invoke` sets this flag, if the call is a tail call.

The HIR is built by abstract interpretation over the bytecodes. The `GraphBuilder` maintains a data structure `ValueStack`, which simulates the state of the current operand stack, the locals and monitors. It builds HIR by iterating over the instructions in a basic block manipulating the `ValueStack` object. An instruction that puts a value on the operand stack, e.g. `iconst_1` causes that a corresponding HIR instruction is pushed on the operand stack in the `ValueStack` object. An instruction that uses operands from the operand stack as inputs gets them from the `ValueStack` object's operand stack. Thereby instructions are linked together.

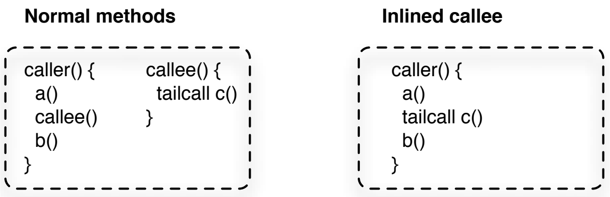
```
iterate over bytecode in basicblock:
switch(bytecode) {
case iconst_1       : valuestack.operandstack.ipush(new Constant(1))); break;
case iadd:          : result = new ArithmeticOp(add, vs.operandstack.ipop(),
                                 vs.operandstack.ipop());
                      valuestack.operandstack.ipush(result); break;
}
```

If one of the invocation bytecodes is encountered, the `GraphBuilder` calls its `invoke` method. This method performs the following steps.

- Check for monomorphic target: If the call is a virtual or interface call, the compiler analyses the receiver type and the target method to check, whether the target can be statically bound. The target can be statically bound, if one of the following conditions hold.
  - Receiver type is final. Hence there can be no actual receiver type that is a subtype of it and overrides a method.
  - Target method is declared as final and hence no sub class can override it.
  - Receiver class is a leaf class. The compiler calls the runtime to perform class hierarchy analysis. If the receiver class is a leaf class, this fact is recorded as a dependency of the current (caller) method. During class loading these dependencies are checked. Since there is only one possible actual receiver type, the call can be treated as static.
  - Target method has only one implementor. Again, this is determined through class hierarchy analysis and a dependency is recorded.

  If the compiler has determined that a virtual call can be statically bound, it changes the bytecode it passes to the `Invoke` object to `invokespecial`.
- Try inline call. If the call is identified as having a monomorphic target in the previous step, the compiler tries to inline the body of the called method. To test whether inlining may take place, the compiler checks several conditions, e.g.

whether the maximum inline depth is reached. A method that is inlined and contains a tail call may prematurely exit the method in which it is inlined. Figure 4.26 illustrates the problem. If the code of the callee is inlined in the caller, the tail call exits the caller, before the method `b` gets executed. Hence inlining is disabled for methods that contain a tail call. If the call is itself a tail call, it is guaranteed



```
caller() {        callee() {
  a()               tailcall c()
  callee()        }
  b()
}
```

```
caller() {
  a()
  tailcall c()
  b()
}
```

Figure 4.26: A method containing a tail call can't be inlined

that no code executes after the call returns, i.e. it is guaranteed that there is no method `b`. In this cases inlining can and is performed, even if the called method contains a tail call.

- Create an `Invoke` object if inlining has failed and append it to the instructions in the current basic block.

LIR

After the construction of the HIR is finished, the compiler creates the LIR, replacing each HIR instruction by possibly a couple of LIR operations. All LIR operations are instance of a subclass of `LIR_Op` and have at least a code that represents the kind of operation and a result operand. Operands can be of different type. There are virtual register, fixed register, stack slot, constant and address operands.

The LIR represents a method invocation by an instance of the class `LIR_OpJavaCall`. This object holds a pointer to the target method, the receiver operand, the address of the call, the result operand, the parameter operands and a pointer to a `CodeEmitInfo` object. The data contained within the `CodeEmitInfo` object is used for safepoint information generation. To support tail call generation a boolean flag is added that indicates, whether a call is a tail call.

The `LIR_Generator` object has one method per HIR instruction, which implements the traversal of this instruction. The method handling an `Invoke` instruction is called `do_invoke` and is modified to handle tail calls. The method performs the following steps.

- Visit the arguments. This step traverses the input arguments of the invocation, thereby building LIR operands containing the result of recursively generating LIR code for the parameter `HIR` subtrees.
- Compute `CodeEmitInfo`: The safepoint information is calculated, i.e. the current bytecode index and the state of local variables, operand stack and monitors.
- Load arguments. The arguments are moved to locations on the stack or in registers according to the calling convention. The corresponding `lir_move` operations are emitted.
- Create the call site. Depending on the bytecode of the `Invoke` instruction the corresponding `LIR_OpJavaCall` object is emitted. If a call is a tail call, the initial target address points to a different runtime resolve method and the tail call flag is set to true on that object.

After the `LIR` is generated, a linear scan register allocator runs assigning real registers to the virtual registers. It also completes the safepoint information with the real location of values on the stack or in registers. Following that machine code is emitted. For every LIR operation the corresponding machine code is emitted. The object performing this transformation is an instance of the class `LIRAssembler`. There are different versions of this object depending for which platform the VM is generated.

The method handling a LIR Java call is called `emit_call(LIR_OpJavaCall *call)` It generates the call site depending on the `LIR_Code` of the `call` object. The method is modified, so that it emits the protection domain token `move` instruction if the call is a virtual tail call.

```
emit_call(LIR_OpJavaCall * call) {
  ...
  if (call->code == lir_icvirtual_call) {
    if (call->is_tail_call())
      asm.movoop(Address(rsp, 0), (jobject)Universe::non_oop_word());
    asm.movoop(rax, (jobject)Universe::non_oop_word());
    asm.call(call.addr()); // The runtime resolve entry.
    store_relocation_info(virtual_call_type, tail_call_type, address_of_call_instr);
  }
}
```

For certain instructions, e.g. allocation of an object, only a fast path is emitted inline. If the fast path fails, execution continues in so called slow case stubs, which are emitted at the end of a method. Exception and deoptimization handlers are also emitted at the end of the method. To support tail calls the method entry points (see Section 4.8.6) are emitted following the above stubs. The address of the entry points is stored in the `nmethod` object.

```
emit_code_epilog(LIR_Assembler* assembler) {
  // generate code or slow cases
  assembler->emit_slow_case_stubs();
  ...
  assembler->emit_exception_handler();
  assembler->emit_deopt_handler();
  // Static tail call entry point.
  assembler->emit_static_tail_call_stub();
  ...
}
```

4.8.9 Server Compiler

The server compiler builds a program dependence graph. Similar to the client compiler, this graph contains special nodes for method invocation instructions. Those nodes also have to be adapted to support the generation of tail call call sites. The program dependence graph first contains platform independent nodes, which are called ideal nodes. When instruction selection is performed, these nodes are replaced by platform dependent MachNode nodes. Both of these node types for call instruction nodes are changed to support tail calls. After the compiler has finished compiling the method, it generates the special tail call entry points at the end of the method.

Ideal node graph

The compiler builds the ideal node graph using abstract interpretation over the bytecodes of a method. The object that constructs the graph is called `Parser`. To build the graph, it iterates over the bytecode and manipulates the state in the current `SafepointNode` object. The current state of operand stack, locals and monitors is stored in a `SafepointNode` object. The inputs of the `SafepointNode` represent the operand stack, locals and monitors. If the operand stack is empty, none of the operand stack inputs is connected to a node. If the operand stack is filled with two items, two of the operand stack inputs point to a subclass of `Node`, which represents the calculation of their value. The interpretation of the bytecodes manipulates the current `SafepointNode`, similar to how the `ValueStack` object is manipulated in the client compiler. The code below shows a simplified version of the `do_one_bytecode` function, which performs the abstract interpretation.

```
switch(bc()) { // Current bytecode.
  // Push updates current SafepointNode to include new top element on operand stack.
  case iconst_1: push(new ConINode(1));
  case iadd: Node* a = pop(); Node * b = pop();
             push(new AddINode(a,b));
}
```

The ideal graph nodes representing a method invocations are `CallDynamicJavaNode` and `CallStaticJavaNode`. They are modified, so that they can store whether a call is a tail call and what kind of tail call, i.e. sibling or non-sibling. The static node is lowered to a static call site, while the dynamic node is transformed to a virtual call site. The following code shows their constructors.

If the `do_one_bytecode` function encounters one of the invoke bytecodes, it calls the `do_call` method. This method either emits one of the above ideal nodes, inlines the call or creates bimorphic call sites. A bimorphic call site is similar to a monomorphic call site with two possible targets. The receiver type check is executed at the call site. If the check is successful, the call proceeds to a static target. If both type checks fail, the code either continues at a call site that performs dynamic dispatching or control is passed to the runtime, which then causes a recompilation of the method. The following code shows how a bimorphic call site looks like.

```
if (recvr.type == classA)
   call classA.method
else if (recvr.type == classB)
   call classB.method
else
   // Dynamic dispatch.
   target = recvr.type.mtable[method_offset]
   call target
   // Or deoptimize.
   call runtime
```

The probable receiver types are computed using profiling information from the interpreter. The `do_call` method performs the following steps to generate the nodes for a call.

- Check for monomorphic target. If the call is an `invokevirtual` or `invokeinterface` call, the compiler tries to determine whether only a unique target method is possible. The conditions under which a virtual call is treated as a static call are similar to the client compiler's. If the check is successful, the call is treated as a not virtual.
- Create call generator. A call generator is an object that creates an ideal node subtree corresponding to the method call. They all share the same interface `JVMState* generate(JVMState*)`. Therefore they take the state of the ideal graph before the call, generate ideal graph nodes according to the invocation instruction and return the state after the call. The subtree can be as simple as a `CallJavaNode` or can be the complete graph of an inlined method. Depending on the type of call and user specified VM flags one of the following call generators is created.

- `WarmCallGenerator` inlines the target method based on profiling information. The compiler uses it for calls that have a known target. Because of the premature method exit problem described in Section 4.8.8 this generator is not used for inlining methods that contain a tail call, except if the call to be inlined itself is a tail call.
- `PredictedCallGenerator` generates a bimorphic call site. It is used for virtual call sites, whose profiling information shows that a major receiver exists and at most two receiver types were encountered. The profiling information maintains the probability that an actual receiver is encountered at a call site. If this probability is high enough, a major receiver exists. Normally the compiler tries to inline the static calls in the bimorphic call site.

```
if (recvr.type == classA) // Profiling showed 90% of invocations go to classA.
  inline call to classA.method
else if (recvr.type == classB) // If profiling shows a second receiver type.
  inline call to call classB.method
else
 call recvr.type.mtable[method_offset]
```

  If the called method contains a tail call, the compiler disables inlining, except if the call is a tail call.
- `VirtualCallGenerator` generates a `CallDynamicJavaNode` and is used for call sites that are virtual. Its `generate` method is changed, so that it passes the appropriate address of the runtime resolve method to the nodes constructor.

```
if (is_tail_call() && is_sibling()) {
  target = SharedRuntime::get_resolve_virtual_tail_call_stub();
} else if (is_tail_call()) {
  target = SharedRuntime::get_resolve_not_sibling_virtual_tail_call_stub();
} else { // Not a tail call.
  target = SharedRuntime::get_resolve_virtual_call_stub();
}
CallDynamicJavaNode *call =
  new CallDynamicJavaNode(tf, target, ..., is_tail_call, is_sibling);
```

- `DirectCallGenerator` generates a `CallStaticJavaNode` and is used for call sites with a non virtual target. The `generate` method is changed, so it passes the appropriate address of the runtime resolve method to the nodes constructor.

• Generate the call site using the call generator. The call generator's `generate` is called, which causes the creation of the subgraph that corresponds to the invocation.

MachNode graph

After the compiler has finished building the ideal graph and has performed platform independent optimizations on it, instruction selection for the target platform happens. The compiler replaces ideal graph nodes by MachNode nodes. To support tail calls the nodes representing method invocations are changed to include boolean values, whether a call is a tail call and whether a call is a sibling call. The function that performs the matching is modified, so that it transfers this additional information from the ideal graph node to the MachNode node.

Code generation

After the compiler has built a control flow graph and performed register allocation, it emits code for each MachNode node. The code is stored in an architecture descriptor file. Each node has a corresponding entry in this file. The entry contains an assembler template of the code that is to be generated for the node. The template for the dynamic MachNode call nodes is modified to pass the security token on the stack and to pass the caller's frame pointer in register `esi`.

```
emit(MachCallDynamicJavaNode* n) {
  asm.lea(esi, [esp-frame_size]);
  if (n->is_tail_call())
    asm.movoop(Address(rsp, 0), (jobject)Universe::non_oop_word());
  asm.movoop(eax, (jobject)Universe::non_oop_word());
  asm.call(call.addr());
  store_relocation_info(virtual_call_type, tail_call_type, address_of_call_instr);
}
```

After the compiler finished generating the method's code, it creates special stubs needed for the current method. Similarly to the client compiler, the server compiler is modified to emit the tail call entry points and store them in the `nmethod` object.

4.8.10 Stack Compression

If the VM disables a tail call because the protection domain of caller and callee do not match, the execution stack grows. If this happens in a sequence of tail calls, the execution stack becomes full. Note that there are usually at most a handful different protection domains in a program. Hence the protection domains in the stack frames repeat themselves. For correct security behaviour, only stack frames containing distinct domains are required. Therefore, the stack can be compressed by removing the stack frames containing duplicate information.

The VM checks on every method entry, compiled and interpreted, whether the execution stack is full. If the stack is full, it calls a runtime routine that throws a `StackOverflow` exception. To prevent this from happening, the modified VM checks on entry to this runtime routine, whether the execution stack can be compressed by leaving some stack frames out. If this is possible, it replaces the original stack frames by a series of deoptimized stack frames containing no superfluous frames and resumes execution. This guarantees that a series of tail calls always executes in bounded stack space, where the bound is proportional to the number of different protection domains that exist in the running program. As long as the number of different protection domains multiplied by the maximum stack frame size in the program is below the maximum execution stack size, tail call optimization in the sense of Section 2.7 is guaranteed.

Deoptimization is normally used to replace one compiled stack frame by one or several interpreted frames. The following data structures are involved during the replacement of stack frames.

- `UnrollInfoBlock`: It stores the number of bytes that have to be removed to remove the deoptimized frame, the sizes of the interpreter frames that are created and the number of interpreter frames to be created.
- `vframeArray`: Stores an array of vframeArrayElements which represent the interpreted frames. A `vframeArrayElement` stores the method, the current bytecode index, monitors, locals and operand stack of the interpreted frame that is to be created.

When deoptimization is required, control enters a deoptimization runtime stub. The code in the stub first calls a function that builds a structure representing the interpreted frames the `UnrollInfoBlock` and the `vframeArray` containing the data. Then it removes the stack frame that is to be deoptimized and replaces it by a series of skeletal interpreter frames using the information in the `UnrollInfoBlock`. The interpreter frames contain no correct values at this point. Next the code calls a function that fills the skeletal interpreter frames with values for locals, operand stack and monitors. This step uses the `vframeArray` contained in the `UnrollInfoBlock`, which stores this values. Finally execution continues in the topmost interpreter frame.

To support compression of the stack the same infrastructure is used. Instead of calculating `UnrollInfoBlock` and `vFrameArray` for only one compiled frame, this information is calculated for the whole (compressed stack). The rest of the deoptimization process stays the same, i.e. the same functions are used. Figure 4.27 illustrates this process.

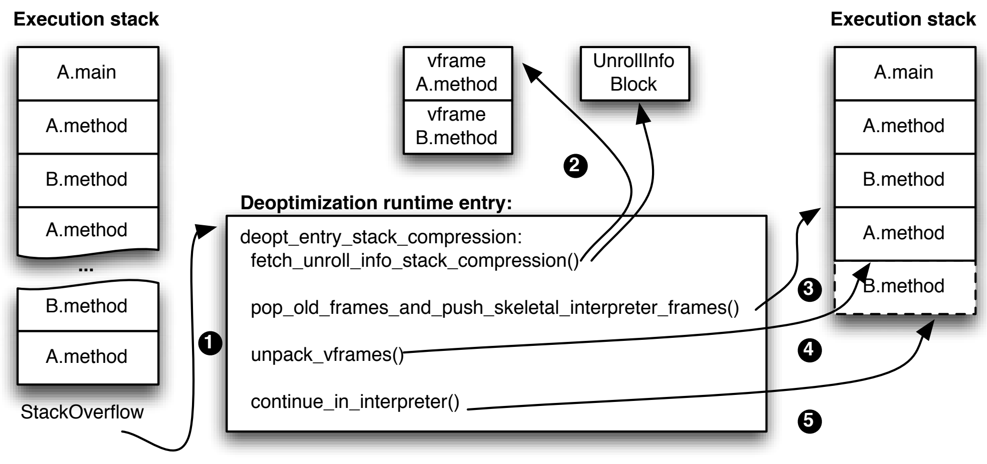


Figure 4.27: Overview of stack compression process

In addition to the existing classes that handle deoptimization, a utility class `StackCompressor` is added to help identifying, whether an execution stack can be compressed and to create a `vFrameArray` containing a representation of the compressed stack. It has one static function `can_compress`, which determines whether the stack can be compressed by looking for sequential frames that are tail calling. If there are such frames, the function returns true, assuming that one of those frames is removable. Note that this is not necessarily the case, because if the two frames contain different protection domains and no other frame on the stack contains the protection domain they cannot be removed. This unlikely case is treated when a compressed representation of the stack is built. To build a compressed representation of the current thread's stack, the deoptimization code creates an `StackCompressor` object and calls its `compress_frames` method, which returns a `vFrameArray` object.

When VM detects a stack overflow, control continues in one of two runtime stubs responsible for throwing a `StackOverflow` exception. Which of the two runtime stubs depends on whether the called method is executed in the interpreter or is a compiled method. Those runtime stub are modified, so that they continue execution in a special deoptimization entry point, if the stack is compressible. To determine whether the stack is compressible, the runtime stubs call the utility function `can_compress`. If the function returns false, the normal `StackOverflow` exception is thrown, otherwise the runtime stub continues execution at the deoptimization entry responsible for compressing the stack.

The code at this entry first calls the function `fetch_unroll_info_stack_compression`, which builds the `vframeArray` and creates an `UnrollInfoBlock` object corresponding to the compressed execution stack. After this function returns, execution continues at the normal deoptimization path and replaces the stack frames using previously computed information. This corresponds to steps 3, 4 and 5 in Figure 4.27.

To build the compressed representation the `fetch_unroll_info_stack_compression` function creates a `StackCompressor` object and calls its `compress_frames` function. The `StackCompressor` object maintains an array of `FrameInfo` objects. A `FrameInfo` object is a representation of a stack frame like the `vframeArrayElement`. Instead of using regular pointers (oops) to refer to objects, it uses `Handle` objects to encapsulate the pointers. This is necessary because during the stack walking a safepoint might occur, when an interpreted frame is queried for its content. At this point the garbage collector might move objects, making the values in the oops invalid. If `Handle` objects are used, the garbage collector can update the pointers within them.

The code within the `compress_frames` function walks the stack and builds an array of `FrameInfo` objects. To create a `FrameInfo` object, the code uses the VM's representation of the current stack frame `javaVFrame`, which can be queried for the locals, monitors, current bci and objects on the operand stack. Note that a `javaVFrame` does not actually contain those objects. It computes their values dynamically using the debugging information for compiled frames or the interpreter runtime for interpreted frames. Hence the need to use a special object to store those values. The following code shows the algorithm used.

```
compress_frames() {
  javaVFrame previous_frame;
  for each javaVFrame curr on the current stack do
    if ( ! previous_frame.is_tail_call()) or
         ! curr.is_tail_call() or
         ! previous_protection_domains.contains(curr.protection_domain()))
      FrameInfo f = new FrameInfo(curr);
      frameInfoArray.push(f);
    }
    previous_protection_domains.add(curr.protection_domain());
    previous_frame = j;
  end
  vframeArray array = vFrameArray::allocate(frameInfoArray, ...);
  return array.
}
```

The topmost frame on the stack is treated specially. Normally the `javaVFrame` contains the state of frame after the method invocation, i.e. the parameters are popped of the operand stack. This is wanted behaviour except for the topmost frame because we want to re-execute the failed invocation. The code that builds the `FrameInfo` reconstructs the parameters from the stack frame and corrects the operand stack values retrieved from the `javaVFrame`.

After the `vFrameArray` is built using the `FrameInfo` array, control resumes in the `fetch_unroll_info_stack_compression` function. It uses the array to compute the `UnrollInfoBlock` and returns both to the deoptimization code. The code uses this data to replace the stack frames and finally resumes execution in the topmost frame.

4.9 Jumping Tail Calls

The call sites showed so far always use a `call` instruction to continue execution at the method entry point. Modern processors use the `call` and `ret` instructions to predict the return address. If those instructions do not match, as it is the case in a series of tail calls, performance can degrade. A series of tail calls in our implementation issues many calls but only the last method in the series returns. This invalidates the internal return address stack, which the processor uses to predict the next instruction after a return. A mispredicted return causes performance hit. If the call instruction is replaced by a `jmp` instruction, the internal state is preserved and the internal return address stack is still valid, when a series of tail call returns. Hence the actual tail call sites have the following form.

```
sub esp  4 // Make room for return address
mov [esp] return_label
jmp tail_call_entry_point
return_label:
...
```

The `call` instruction is replaced by a `jmp` preceded by two instructions that safe the return address on the stack. The return address on the stack is needed because the runtime functions requires it, e.g. for changing the state from a monomorphic to a polymorphic call.

4.10 Optimizing Sibling Calls

The implementation described so far allows for general tail call optimization in the Java HotSpot^{TM} VM. This was the main goal of this thesis. When evaluating this implementation's performance compared to approaches used with VM's that do not support tail call optimization, results showed that for polymorphic calls the implementation performed worse than a trampolined version, if the method has many parameters. This is due to the overhead of moving the arguments at the call target, effectively moving arguments twice on every call. Therefore a second prototype was implemented, which moves the arguments before the call. This prototype does not support stack compression. If there is a protection domain mismatch, an exception is thrown. The following changes to the existing implementation are made.

- Server and client compiler sibling tail call sites move arguments directly onto the caller's incoming argument area. The code that builds the LIR for moving arguments and the code for building the MachNode graph is modified, so that it moves the arguments to the appropriate place in the caller's caller instead of to the caller's outgoing argument area.
- The verified and unverified sibling tail call entry points do not move arguments.
- The verified and unverified sibling compiled-to-interpreter adapters do not move arguments.
- The runtime is modified to correct the oop mapping, if it detects that a sibling tail call has moved the arguments on to the caller's incoming parameter area.

The prototype could be extended to support stack compression by adding additional entry points to every method, which move the arguments from the caller's incoming argument area to the caller's outgoing argument area. If the tail call is disabled, control is transferred to these entry points before it continues at a normal method entry. Due to the limited time frame of a master thesis this is left as a future improvement.

5 Evaluation

The optimization is implemented in version b44 of Sun's OpenJDK 7 for the IA-32 architecture. To evaluate performance of tail calls, we compare the execution times of a program that performs tail calls in our implementation to the same program that performs normal calls instead and to a modified version of the program that uses a trampoline to guarantee that the stack does not overflow. The program contains a method that performs recursive calls that can be tail call optimized. Several versions of the program with varying number of parameters for this method and different kind of call sites are compared. While this does not show performance of real programs, where the fraction of tail call instructions is likely less (and hence the impact of a speedup or slowdown is less), it gives a good estimation of the worst slowdown or best speedup, we can expect from using the tail call optimizing implementation.

The test machine is an Apple Mac Pro with two Intel Core2 Xeon Processors running at 2 GHz. Each processor has 2 cores and 4 Megabyte L2 cache. The machine is equipped with 3 Gigabyte main memory and uses OpenSolaris 2008.11 as operating system.

5.1 Program

The program is designed to measure the overhead of tail calls. The core is a method `test`, which repeatedly invokes a method `tailcaller`. This method performs the recursive calls.

```java
void test(int recursionDepth, int repeat) {
  int result;
  for (int i = 0 ; i < repeat; i++) {
    result = tailcaller(recursionDepth, 0, ...numberArgs);
    //or   = object.tailcaller(recursionDepth, 0, ...);
  }
}
```

To measure the execution time, the current system time is taken before and after this method is called. The program is parameterized by the recursion depth to be measured.

```
void main(String args[]) {
  int recursionDepth = parameter;
  int repeat = 90000000 / recursionDepth;
  ...// Warmup
  long start = System.currentTimeMillis();
  test(recursionDepth, repeat);
  long end = System.currentTimeMillis();
  println(end-start);
}
```

The function that performs the tail call is either a static method or a method of an object. It performs an addition or subtraction on each argument before calling a second function `tailcallee`. The first argument of those functions is the counter of the recursion depth. It is decreased on every recursive call. If it reaches zero the functions return.

```
int tailcaller(int recDepth, int arg2, .. ., int argn) {
  if (recDepth==0) return arg2;
  else
    return tailcallee(recDepth-1, arg2+1, .. ., argn +1);
}
int tailcallee(int recDepth, int arg2, .. ., int argn) {
  if (recDepth==0) return arg2;
  else
    return tailcaller(recDepth-1, arg2+1, .. ., argn +1);
}
```

To measure polymorphic call sites, two global variables are used to ensure the call site sees two distinct receiver classes. Those variables are initialized before measurement of the execution time starts.

```
class Test1 {
  int tailcaller(int recDepth, .. .) {
     if (recDepth==0) return arg2;
     else {
       if (recDepth%3==0) target = instanceTest1;
       else target = instanceTest2;
       return target.tailcallee(recDepth-1);
  }}}
```

To compare the tail calling implementation to alternative techniques, a trampolined version of above methods is used. The principle employed is the same as explained in Section 2.6. For efficiency we do not allocate new continuation objects, every time a tail call is performed. Instead a `Context` object is used, which holds the parameters and the result. This object is passed to the `apply` method. The `apply` method returns instances of subclasses of the `Procedure` class, the tail called method. These instances are initialized once before the test is run.

```
class Context {
  int recDepth; int arg2; ... int argn; int result;
}
abstract class Procedure {
  Procedure apply(Context c);
}
class TailCaller extends Procedure {
  Procedure apply(Context c) {
    if (c.recDepth==0) {
      c.result=c.arg2;
      return null;
    }
    c.recDepth++;
    c.argn +=2;
    return tailCalleeInstance; // class TailCallee extends Procedure ...
  }
}
int tailcaller(int recDepth, .. ., int argn) {
  Context c = context; // Global Variable.
  Procedure p = tailCallerInstance;
  c.recDepth = recDepth;
  c.argn = argn;
  do { // Trampoline.
    p = p.apply(c);
  } while (p!=null);
  return c.result;
}
TestCaller testCallerInstance; TestCallee testCalleeInstance;
```

The trampolined version is inherently polymorphic. Therefore we can only measure the above version, i.e. there is no differentiation between static and polymorphic call sites.

The program is tested with `tailcaller` methods, which have two to eight arguments. There are three versions of every program, where the recursive call sites are either static, monomorphic or polymorphic. For every of this tests we use runs with recursion depth 1, 10, 50, 100, 500 and 1000. The tests are run using the client and server compiler. The initial tail call implementation, which moves the arguments at the method entry and the optimized tail call implementation, which moves the arguments at the call site, are compared. Results of the server compiler are not reported, because they exhibit similar behaviour as the client compiler results. The same holds for monomorphic calls. They show the same behaviour as polymorphic calls. For brevity of presentation we leave both out. The following configurations are used.

- Base: The client compiler is used and no tail call optimization is performed. Every call increases the size of the execution stack.

- Trampoline: The client compiler is used and the trampolined version of the program is executed.
- Entry: The client compiler is used with tail call optimization enabled. Sibling tail calls move the arguments to the caller's outgoing area at the callsite and on method entry from there onto the caller's incoming area.
- Callsite: The client compiler is used with tail call optimization enabled. Sibling tail calls move the arguments at the call site as described in Section 4.10.

The program is verified to be executing in compiled code by using debugging flags. Each measurement is repeated 10 times and the arithmetic mean of the results are reported. No garbage collection is happening, while the time is measured. Inlining is turned off. Results are reported as speedup in percent relative to the Base configuration.

5.2 Static Sibling Calls

Figure 5.1, 5.2, 5.3 and 5.4 show the results of running the programs when both `tailcaller` and `tailcallee` have the same number of parameters ranging from two to eight. In the configurations that perform tail call optimization, the call continues at the sibling tail call method entries.

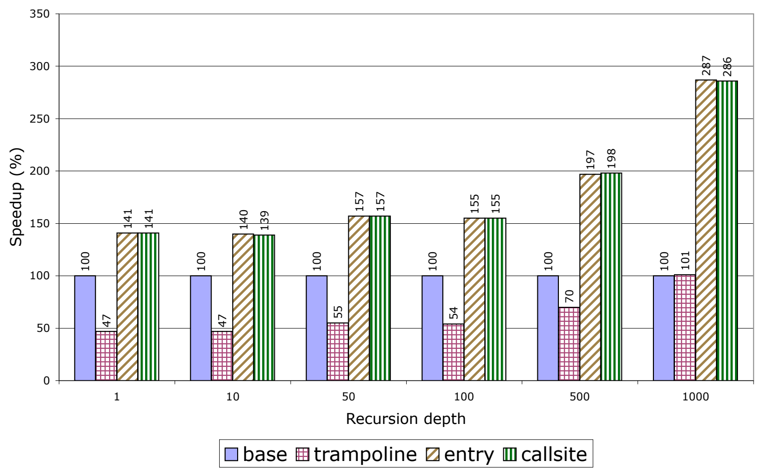


Figure 5.1: Static call with two arguments

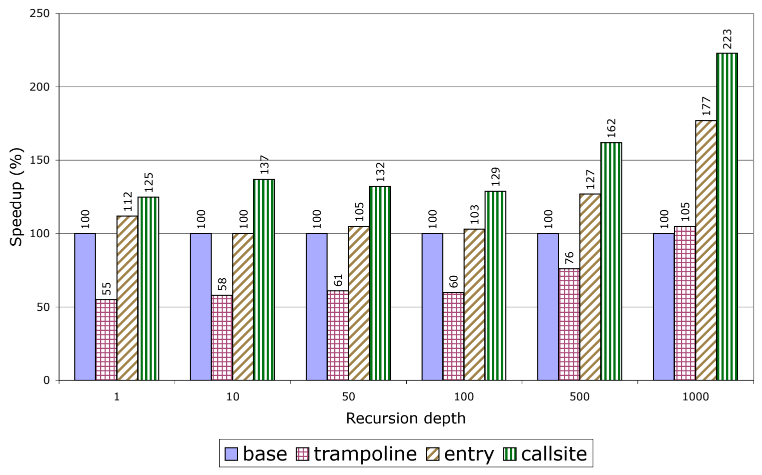


Figure 5.2: Static call with four arguments

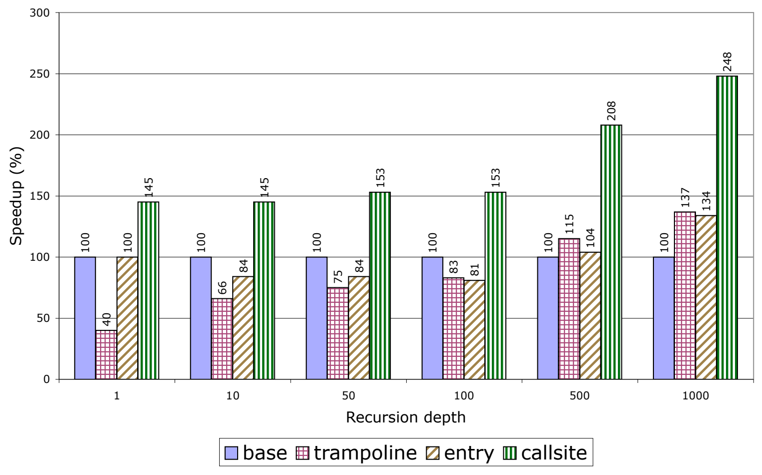


Figure 5.3: Static call with six arguments

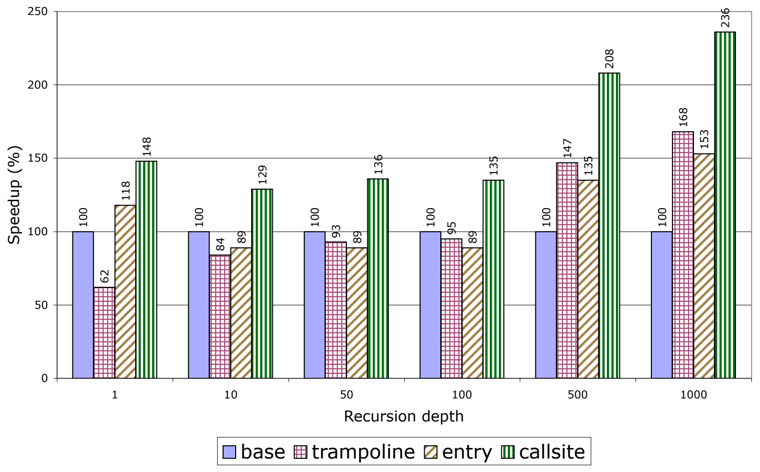


Figure 5.4: Static call with eight arguments

Faster method prolog The code at method entry of sibling tail calls creates the frame of the callee by subtracting the callee's frame size from the frame pointer and storing this value in the stack pointer. The old frame pointer is already on the stack. Hence one instruction less is needed compared to the normal method entry, which first pushes the old frame pointer to the stack and then builds the new frame by subtracting from the stack pointer. This explains the 41% advantage of Entry and Callsite over Base at recursion depth two with two parameters (all parameters passed in registers).

Moving arguments at the call site The configuration Callsite is always faster than normal method execution, the trampolined version and the tail call version that moves arguments twice. Callsite is typically around 20% faster and up to two times faster than Entry (six arguments recursion depth 500). This implies that on architectures with few registers like IA32 this optimization is beneficial. We can expect that in real programs, running in the second prototype, i.e. Callsite, static sibling calls that cause tail call optimization do not pose a performance overhead when compared to programs that do not perform tail call optimization or use trampolines to achieve it.

All arguments in registers When the program is executed with two parameters, the configuration Callsite and Entry are equally fast. This is because for static calls the first two integer parameters are passed in registers. No arguments are passed on the stack.

From this we can conclude that on architectures, where most of the parameters can be passed in registers, e.g. on Sparc and on x86-64, the optimization described in Section 4.10 is not necessary because programs rarely use methods that have more parameters than would fit into the parameter registers and the overhead of moving arguments on the stack seldom occurs. With increasing parameter size configuration Entry becomes slower relative to Callsite. This is due to the overhead of moving the arguments twice.

Normal calls vs. moving arguments twice Normal calls increase the size of the call stack. After a certain recursion depth, the memory used for the execution stack does not fit into the processors first cache, the level one (L1) cache. Data has to be evicted to the L2 cache. This causes an overhead in execution time for programs executing with configuration Base.

This effect can be observed when comparing execution times of Entry to Base at recursion depths 500 and 1000 for calls that have six or more parameters. Although Entry has the additional overhead of moving arguments twice it is faster than Base. For calls with recursion depth below that we see an overhead of up to 23% (six arguments recursion depth 100) compared to Base. The time that Entry spends extra for moving the arguments outweighs the time advantage that Entry has because the stack does not grow and less L1 cache misses occur. If the argument size is increased however the stack of Base grows faster and the overhead that Entry has due to moving decreases, e.g. to 13% for 8 arguments and recursion depth 100.

We verified the claim that L1 cache misses indeed happen more often by taking the assembler output of the two programs and compiling it into executables. We observed cache behaviour of this executables with the Unix tool `oprofile`[36]. This tool allows reading the processors internal performance counters. The program corresponding to Base showed significant (e.g. 40 times more for recursion depth 500 and six parameters) more cache misses than the program corresponding to Entry.

From this we conclude that with programs containing mostly deep recursions, i.e. greater than 500, Entry poses no overhead. If many tail call optimized calls happen that do not recurse that far, we can expect a worst slow down of up to 23% compared to a program that performs normal calls. Note that these numbers only apply, if the program consists mostly of tail calls. If the fraction of tail calls within the program decreases the actual slow down decreases.

Trampolined calls Trampoline shows a significant overhead when compared to the other configurations at low recursion depths. This is due to the costs of the loop, which repeatedly calls the next procedure. The actual call to the method is a virtual call resulting in additional overhead. At deep recursions (500 to 1000) the cache effect described in the above paragraph results in Base being slower than the trampolined version, e.g. at recursion depth 500 with six parameters Trampoline is 10% faster than Entry. This is due the overhead of Entry moving the arguments twice.

In a real program when automatic conversion is used, every call that is not a tail call has to use a trampoline, e.g. if `tailcaller` contains calls to two other methods `a` and `b` then the body of the `Tailcaller.apply` method contains two trampolines for calling to `A.apply` and `B.apply`. The trampoline has to be used for every call, because in general it is unknown if the called methods contain tail calls and hence need a trampoline. The overhead of trampolines in such cases is therefore much higher than in this example where only one trampoline exists.

To validate that claim, we modified the tailcaller method to include one call to a method `calc` that adds its four parameters. With this modified version the Trampoline configuration includes a trampoline in the `Tailcaller.apply` method, which calls to the method `Calc.apply`. Trampoline is already 20% slower than the program executing in Entry when using a recursion depth of 500 or 1000. Hence we believe that in most real programs the general overhead of trampolines outweighs the time advantage for deep recursions. The disadvantage that Entry exhibits with less deep recursions, is compensated by the advantage of Entry on deep recursions and regular calls, which do not tail call.

5.3 Non-Sibling Calls

To show how non-sibling calls that are tail call optimized influence performance, the program is executed in a variation, where `tailcallee` has two more arguments than `tailcaller`. The call from `tailcaller` to `tailcallee` hence is a non-sibling call. In tail call optimizing implementations the non-sibling tail call entry is reached. Figure 5.5 and 5.6 show the execution times of the different configurations. An overhead in Callsite and Entry compared to Base is expected, because non-sibling tail calls cause execution to be continued in interpreted mode, every time a series of tail calls starts. The deeper the recursion the less impact this has on total execution time because more time is spent recursing and the less often program execution continues in the interpreter. Every time code enters the non-sibling entry, the caller's caller is checked whether it is an interpreted frame. This causes additional runtime overhead. The compiler reserves a minimum of four stack slots for every call. Every call with two or four parameters is a sibling call. Therefore we only show the results of calls with six and eight parameters in this section.

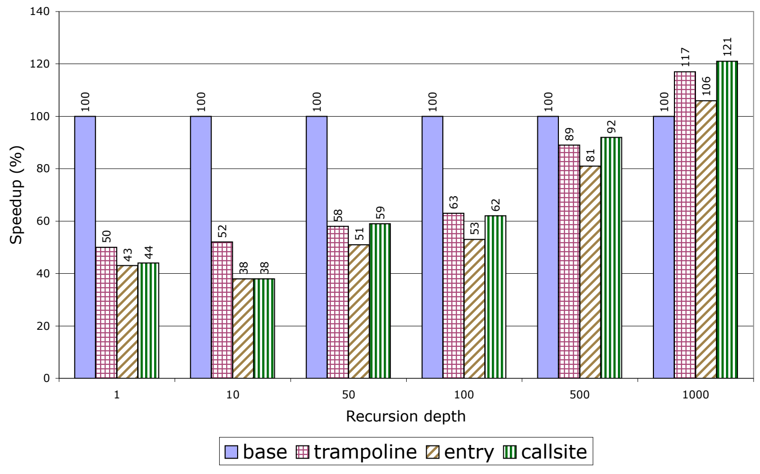


Figure 5.5: Non-sibling calls six arguments

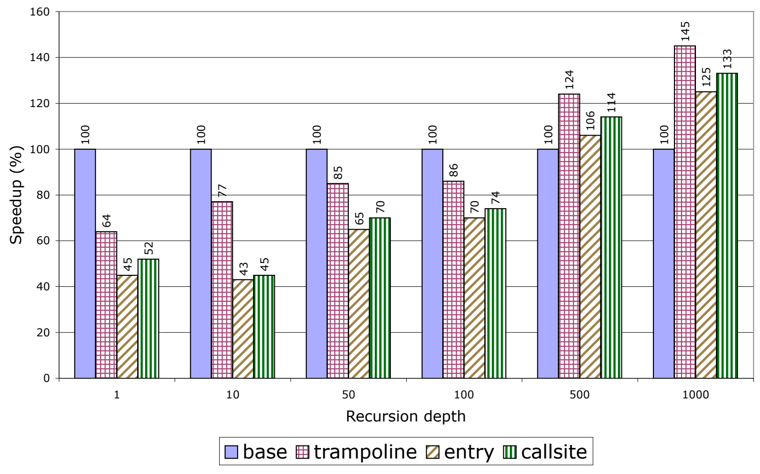


Figure 5.6: Non-sibling calls eight arguments

At recursion depth 500 Entry is 24% slower than Base and Callsite is 9% slower when the six parameter version is used. At recursion depth 1000 both are faster. The trampoline version matches and exceeds the speed of the tail calling versions, because it does not have to resort to the interpreter. Non-sibling tail calls in Callsite also pay a performance penalty when compared to Base, because arguments are moved twice for non-sibling calls.

This overhead can be prevented by setting the JVM flag `MinOutgoingArgsSlotSize` to a value that is equal to the biggest incoming argument stack slot size required by any (or any frequently executed) method in the program.

5.4 Polymorphic Calls

Figure 5.7, 5.8, 5.9 and 5.10 show the results of running the polymorphic version of the program with two to eight arguments. All calls are sibling calls. The calls to `tailcaller` and `tailcallee` go through a vtable stub, which performs the dynamic dispatch. This vtable stub performs a protection domain check in configuration Callsite and Entry.

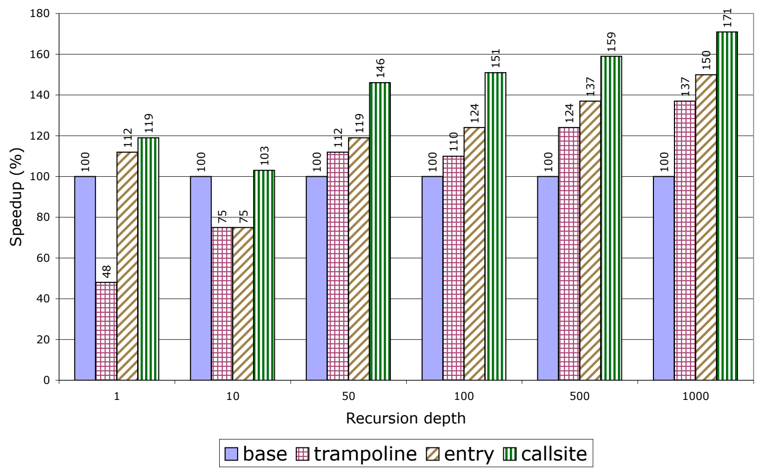


Figure 5.7: Polymorphic call with two arguments

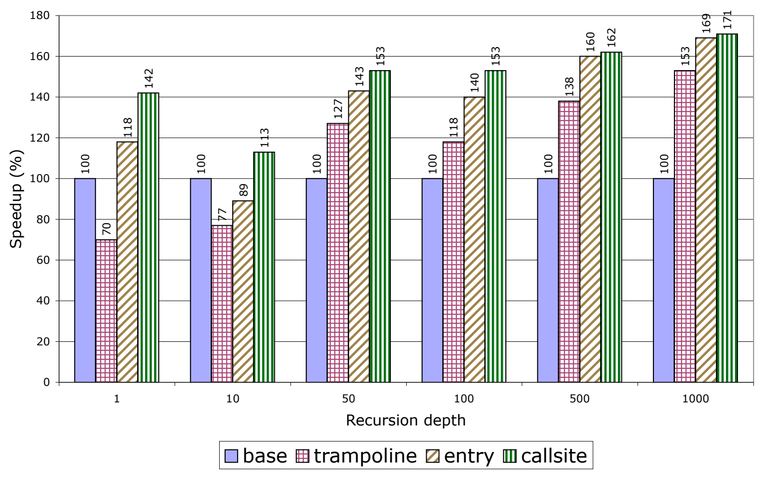


Figure 5.8: Polymorphic call with four arguments

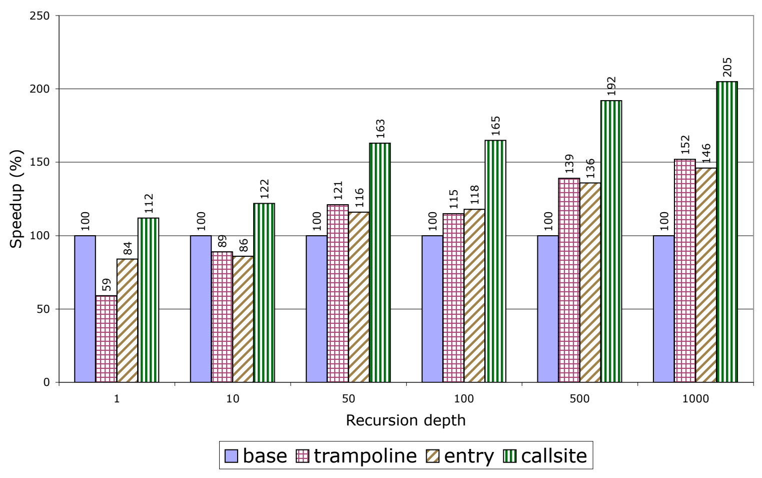


Figure 5.9: Polymorphic call with six arguments

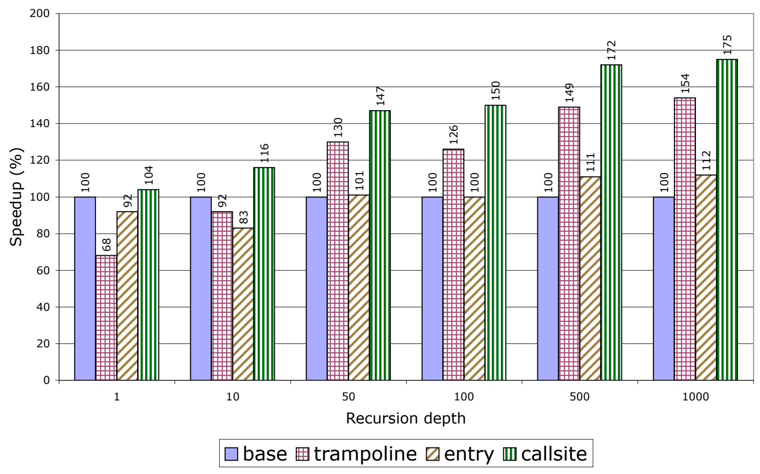


Figure 5.10: Polymorphic call with eight arguments

Faster method prolog The advantage that Callsite and Entry exhibit due to the shorter method prolog can be also observed for polymorphic calls. When the recursion depth is one and the program has calls with two or four parameters, both configurations are faster than Base. For the programs where six or eight parameters are used, this advantage is not sufficient to counterbalance the additional overhead that Entry has for moving the arguments twice. Callsite is always faster than Base.

Moving arguments at the call site In correspondence to static calls Callsite is also always faster than the other configurations. It has the advantage of cache locality and a shorter method prolog when compared to Base. Compared to Entry, it does not have to move the arguments a second time. This advantage makes up for up to 57% faster execution time (recursion depth 1000 with eight arguments). We can again conclude that this optimization is beneficial, if the executed programs contain many short methods with many parameters that perform tail calls.

Normal calls vs. moving arguments twice We can observe the overhead that Entry has compared to Base at recursion depth one and ten, because it moves the arguments to the caller's incoming area at method entry. At deeper recursion depths the cache locality benefit and the shorter prolog of Entry outweighs that overhead. Here, Entry is faster than Base. In real programs the outcome depends on whether most tail calls recurse beyond a depth of ten or not.

Trampolined calls The usage of a trampoline poses an overhead at recursion depth 1 when compared to the other configurations. At recursion depth ten Trampoline is already faster than Entry when the program is run with functions with six or eight parameters. This is because Trampoline does not move arguments twice. The disadvantage that Trampoline has when compared to static calls, because it uses a polymorphic call site, does not weigh in. At recursion depth 500 and with eight parameters Trampoline is 34% faster than Entry. This disadvantage of Entry motivated to build the second prototype Callsite.

Non-sibling calls Figure 5.11 shows the performance of non-sibling polymorphic calls, when functions with eight arguments are used. We can observe the same characteristics as with static calls. Both Callsite and Entry are slower at low recursion depths. The deeper the recursion the less impact the time spent in the interpreter has. Trampoline is faster than Callsite and Entry because it does not have to resort to the interpreter.

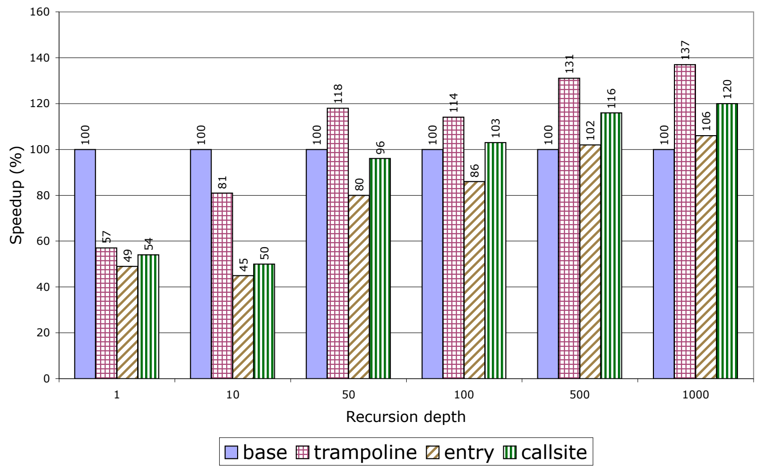


Figure 5.11: Polymorphic non-sibling calls with eight arguments

5.5 Stack Compression

If stack compression is enabled, the implementation tries to compress a series of tail calls, if the stack overflows. The compress operation uses the deoptimization infrastructure. Hence it is slow. We try to measure a worst case scenario again using our test program with two, four and eight parameter versions. The program is run once using a recursion depth of 1000 and once using a recursion depth of 20000. At 20000 the stack overflows in all three versions if stack compression is turned off. The program with methods having two parameters is approximately 60 times slower when stack compression happens, the one with four parameters is 50 times slower and the one with eight parameters is 20 times slower. The slowdown improves with increasing parameter size because the frame size of the methods increases. Less frames are to be compressed. Note that this slowdown only happens, if the recursion depth causes the stack to overflow. In the conclusions we propose an improvement that should help lower the slow down.

6 Related Work

Probst [47] describes the addition of a tail call calling convention to the GNU Compiler Collection (GCC). The standard C calling convention requires the caller to remove the arguments after a call. Compilers implement this either by popping the arguments of the stack after the call instructions or by using a fixed frame size for a method that provides enough space for all outgoing function parameters. Because of this 'caller pops arguments' convention the c calling convention can not support tail calls between two function where the called function has more arguments than the caller - a non-sibling call. Such a call extends the caller's caller frame to make room for the additional parameters of the callee. When the callee returns the caller's caller stack pointer has a wrong value. The tail call calling convention proposed by Probst solves this problem by using a 'callee pops arguments' convention. Every function removes its arguments on return. Suppose we have a sequence of tail calls `f(a)` calls `g(a,b)` calls `h(a,b,c)`. The call to `f(a)` causes the caller's caller stack pointer to be extended by one stack slot. `f(a)` tail calling `g(a,b)` causes the caller's caller stack pointer again to be extended by one stack slot (to make room for the additional argument of `g`). The same happens between the call from `g` to `h`. Finally `h` returns popping three parameters of the stack resulting in a proper alignment of the caller's caller stack pointer. This convention was implemented for IA-32 and Alpha. The changes were not integrated into the official version of GCC. The same approach could be used for implementing non-sibling tail calls in the Java HotSpot$^{\text{TM}}$ VM, if the assumption that frames have a fixed size was not so wide spread in the code base of the VM.

Bauer [5] describes the implementation of indirect sibling tail calls for GCC, which had not been supported up to the publication of his thesis. Indirect sibling calls are tail calls, where the call target is computed dynamically. This less radical change became part of the official version of GCC. GCC currently does not support non-sibling tail calls. Sibling calls are implemented as described in Section 2.4. A tail call moves the arguments on to the caller's caller outgoing arguments area, removes the caller's stack frame and jumps to the callee. This approach makes it unnecessary to have special entry points to a function that handle tail calls. Since the programming languages that GCC supports do not feature a stack based security mechanism, a call that is a tail call can always remove the caller's stack frame.

IBM's Java Just-in-Time compiler version 3.0 [54] uses tail recursion elimination as described in Section 2.6 to optimize self recursive static method invocations. The method call is replaced by a branch to the beginning of the method and the parameters are set accordingly. This is not sufficient to guarantee general tail call optimization.

The LLVM compiler infrastructure [34] is similar to a Java VM in that it also provides a virtual instruction set, which is executed by the LLVM infrastructure. Instead of bytecodes that operate on an operand stack the instruction set represents a virtual load store architecture with three operand instructions. The result of instructions is stored in variables that model an infinite register set. All variables in a LLVM program are in SSA-form. The instruction set contains general purpose instructions found in modern processors such as control flow instructions, binary arithmetic operators, logical operators and comparison operations. In addition it contains more high-level constructs such as memory allocation instructions, function calls and exception handling.

The LLVM infrastructure provides an in-memory graph based representation of the program on which optimizations are defined and which provides the basis for code generation. This representation is similar to HotSpot^{TM}'s client compiler's HIR in that it is a control flow graph of basic blocks containing instructions. Instructions refer to their inputs through pointers. There are various backends that transform this representation to machine code either at compile time or just-in-time. The LLVM instruction set supports tail call's through a 'tailcall' prefix and a special calling convention 'fastcc'. If a call is in tail call position, caller and callee are marked as using the 'fastcc' convention, the backend can optimize the tail call removing the caller's frame. The author of this thesis implemented tail calls for the x86 32bit and 64bit and powerpc 32bit backends. Similar to Probst the fastcc calling convention causes functions to pop their arguments on return. LLVM also provides an optimization pass defined on the platform independent graph that performs tail recursion elimination.

Peyton Jones et al. [46] describe a portable assembly language called *C–*. It is designed to replace the programming language C as a portable backend language. Many language implementations were using C as target when C– was created. It's syntax is similar to the sytnax of the C but features only a limited number of data types. It is different from C in that it provides a runtime interface, which allows writing runtime services such as garbage collection, exception handling and debugging. C– supports tail calls by a special function call instruction `jump`. A C– backend converts such an instruction to a tail call, which replaces the caller's stack frame by the callee's frame. How this is accomplished is not further specified. C– has runtime support for walking the stack but no built-in support for a security access mechanism based on stack frames. If such a mechanism is built, the implementor has to deal with the fact that the tail call removes a stack frame possibly loosing information.

The .Net framework [8] provides a virtual machine—the *common language runtime* (CLR)—that has an instruction set, the Microsoft Intermediate Language MSIL, that operates on an operand stack similar to Java bytecode. MSIL features a `tailcall` prefix that can precede a call instruction. The call is then executed by the CLR as a tail call removing the caller's stack frame. However the documentation [39] says that the stack frame is not removed, if control is transferred from untrusted code to trusted code. Such a transition is equivalent to a protection domain mismatch in the JVM.

Clements and Felleisen [12] describe an abstract machine that can optimize all tail calls while still maintaining correct security information in the presence of stack inspection. The proposed machine stores a table of permissions with each frame. When a tail call is performed the table of the caller's caller is updated to include the permissions of the caller. When the stack is inspected these tables are taken into account. Hence the permissions of the removed caller frame are not lost. Using such a scheme within the Java HotSpot$^{\text{TM}}$ VM incurs an overhead for every tail call that involves differing protection domains even if the stack does not overflow. Hence we decided to lazily compress the stack.

Schinz and Odersky [50] describe how to implement tail call optimization on a JVM that does not support it. They transform the program to use trampolines as described in Section 2.6. The disadvantage of this method compared to supporting tail calls natively within the JVM is the performance overhead and the additional implementation work that is required.

7 Conclusions

The presented changes to the Java HotSpot$^{\text{TM}}$ VM allow for general tail call optimization in the presence of the Java access security mechanism. A series of tail calls can execute in bounded stack space not causing the stack to overflow. A language implementation can rely on the fact that a series of tail calls does not cause a stack overflow exception. This was the main goal of this thesis.

The following components of the Java HotSpot$^{\text{TM}}$ VM were modified to handle tail calls

- The semantics of Java bytecode was augmented to allow marking method invocations as tail calls using the `wide` prefix.
- The bytecode verifier was changed to check the conditions under which a tail call is valid.
- The interpreter was modified to handle tail call invocation instructions.
- The client compiler was modified to emit special call sites for tail calls that include the protection domain token. It also emits code for the different tail call method entry points.
- The server compiler was modified to emit special call sites for tail calls and emit code for the tail call method entry points.
- The runtime was modified to link the appropriate entry points at call sites.
- The deoptimization infrastructure was extended to allow compressing the execution stack when a series of tail calls causes the stack to overflow.

The evaluation shows that for sibling calls performance is faster than a normal method invocation. Non-sibling calls currently cause a performance overhead on calls that do not recurse deep because control is often transferred into the interpreter. The implementation provides a flag the user can set `MinOutgoingArgsSlotSize`, which helps to minimize the effect of non-sibling tail calls. If this flag has a value other than zero, every stack frame reserves an outgoing argument area of at least the size of the value of this flag. Hence every call to a method that requires at most this many stack slots is a sibling tail call and does not pay the performance penalty.

The existing prototype already shows good performance especially when compared to alternative methods on JVMs that do not support tail call optimizations. However the following enhancement could be applied to further improve performance:

- Usage of adapter frames for non-sibling tail calls. When a non-sibling tail call happens and the caller's caller frame is not interpreted it causes control to continue in an interpreted frame. This incurs a significant performance hit. Instead of resorting to the interpreter, the non-sibling method entry could create an adapter frame where the arguments can be moved. Every subsequent tail call could then extend this adapter frame. The creation of such an adapter frame is cheap (a few instructions) compared to transferring control to the interpreter.
- Eagerly compressing the stack. Currently if the protection domain check fails, the tail call is disabled. If the recursion is deep enough, the stack overflow and is compressed by the implementation. This compression is slow because it has to walk the whole stack analyzing the frames content. In many cases the protection domains repeat themselves after few frames (e.g. as shown in Figure 4.6, where the protection domain repeats itself every second frame). This observation can be used to improve the worst cases performance for such cases. Instead of disabling the tail call if the caller's and callee's protection domain mismatch a check could be performed, which inspects the last n caller's protection domains for a match. If an equal domain is found for one of those methods the tail call can be performed. If the protection domain is not found the tail call is disabled. While this decreases performance for calls that never cause a StackOverflow because of the overhead caused by this check, the worst cases performance is better because stack compression happens less or never (if there are never more then n frames between two differing protection domains).
- Inlining methods that contain tail calls. We currently disable inlining for calls to methods that contain tail calls (see Section 4.8.8). This can negatively effect performance. An improved version would also inline non-tail calls to methods that contain tail calls. To maintain correct execution behaviour, the implementation would disable tail calls in such inlined methods. This transformation does not invalidate tail call optimization because inlining does not cause the creation of a new frame.

If stack frames in compiled code are not extendable, special cases to handle non-sibling tail calls have to be introduced in the JVM. In our implementation every sibling tail call entry point has a corresponding non-sibling tail call entry point. Hence there are two extra entry points to a compiled method (`nmethod`) and two extra entry points to the compiled-to-interpreted adapters. Extra logic within the JVM is needed to create the initial extendable frame in a series of non-sibling tail calls. In our implementation this logic causes a run of the method in the interpreter. An alternative technique would be to use a special extendable adapter frame. All these aspects complicate the implementation of tail calls.

An alternative specification that eases the implementation of tail calls could limit tail call optimization to sibling tail calls. The bytecode verifier would reject all other tail calls. A sibling tail call in this context would be a tail call from a method `f` to `g` where the signature of the parameters of `g` *fits into* the signature of the parameters of `f`. The definition of *fits into* has to guarantee that the stack space required for the parameters of `g` is less or equal to the stack space required for the parameters of `f` on all virtual machines that implement the specification.

A possible definition would be to require that for each n occurrences of parameters of type X in the signature of `g`, there are at least n occurrences of parameters of type X in the signature of `f`. For example the following call would be a sibling tail call.

```
int f(int a, byte b, int c, byte d) {
  return g((int)a, (int)b, (byte)c, (byte)d);
}
```

But the following call would not be a sibling tail call because the signature of `f` contains only one integer parameter.

```
int f(byte a, int b) {
  return g((int)1000+a, (int)b);
}
```

Programming language compilers that output bytecode can work around the sibling tail call restriction by emitting additional artificial parameters to satisfy the sibling requirement. For example the previous signature could be changed by the language compiler to include the required second integer parameter.

```
int f(byte a, int b, int artificial) {
  return g((int)1000+a, (int)b);
}
```

The disadvantages of this restriction are

- Method signatures contain parameters that are never accessed in the method.
- Slower code. Depending on how the compiler handles parameters and how stack slots are managed, additional moves for the artificial parameters are introduced. In the following example the call to `f` can cause extra moves for the `artificial` parameter.

  ```
  int f(byte a, int b, int artificial) {
  ```

```
  if (...) return g((int)1000+a, (int)b);
  return f(a, b, artificial);
}
```

The server compiler in the Java HotSpot$^{\text{TM}}$ VM views stack slots as an extension to the machine registers. Hence it recognizes that it does not need to move the artificial parameter in the example above. The client compiler handles stack slots differently and therefore emit a series of moves. If `f` is called from another function, there are always additional moves for the artificial parameter.

```
for (int i = ....) {
  res f(a,b, /*artificial move*/0);
}
```

The changes to the Java HotSpot$^{\text{TM}}$ VM are available at the multi-language Da Vinci Machine project [49], which purpose is to experiment with features required for languages other than Java. The author hopes that they will form the basis for bringing tail calls to the Java HotSpot$^{\text{TM}}$ VM.

List of Figures

A Bibliography

[1] Ole Agesen. GC points in a threaded environment. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1998.

[2] Ole Agesen and David Detlefs. Mixed-mode bytecode execution. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2000.

[3] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 207–222, New York, NY, USA, 1999. ACM.

[4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, New York, NY, USA, 1998. ACM.

[5] Andreas Bauer. Übersetzung funktionaler Sprachen mittels GCC - Tail Calls. Master's thesis, Technische Universität München, 2003.

[6] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 129–140, New York, NY, USA, 1998. ACM.

[7] Per Bothner. The Kawa language framework. `http://www.gnu.org/software/kawa/`.

[8] Don Box and Ted Pattison. *Essential .NET: The Common Language Runtime.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] Gilad Bracha and Graham Hamilton. *JSR 202: classfile specification update. http://jcp.org/en/jsr/detail?id=202.* Sun Microsystems, Inc., 2006.

[10] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 98–105, New York, NY, USA, 1982. ACM.

[11] C.J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[12] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[13] Xavier Clerc. Ocaml-java project. `http://ocamljava.x9c.fr`.

[14] Clifford N. Click. *Combining analyses, combining optimizations.* PhD thesis, Rice University, Houston, TX, USA, 1995.

[15] Clifford N. Click and Michael Paleczny. A simple graph-based intermediate representation. In *Papers from the ACM SIGPLAN Workshop on Intermediate Representations*, pages 35–49, New York, NY, USA, 1995. ACM.

[16] William D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185. ACM, 1998.

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

[18] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

[19] David Detlefs and Tony Printezis. A generational mostly-concurrent garbage collector. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2000.

[20] 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*, 9(3):319–349, 1987.

[21] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java^{TM}Language Specification, The 3rd Edition.* Addison-Wesley Professional, 2005.

[23] Robert Griesemer. Generation of virtual machine code at startup. *OOPSLA99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, 1999.

[24] Jr. Guy L. Steele. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *Proceedings of the 1977 annual conference*, pages 153–162, New York, NY, USA, 1977. ACM.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 4 edition, 2007.

[26] Rich Hickey. Clojure. `http://clojure.org`.

[27] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag.

[28] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43, New York, NY, USA, 1992. ACM.

[29] Erik Huelsmann and al. *Armed bear common lisp.* `http://common-lisp.net/project/armedbear`.

[30] Jim Hugunin, Barry Warsaw, Samuele Pedroni, Brian Zimmer, and Frank Wierzbicki. The jython project. `http://www.jython.org`.

[31] Intel Corporation. *Intel IA-32 Architecture Software Developerś Manual: Basic Architecture*, 2003.

[32] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action.* Manning Publications Co., Greenwich, CT, USA, 2007.

[33] 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):1–32, 2008.

[34] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. `http://llvm.org`.

[35] Xavier Leroy. Java bytecode verification: An overview. In *Proceedings of the International Conference on Computer Aided Verification*, pages 265–285, London, UK, 2001. Springer-Verlag.

[36] John Levon. *OProfile manual.* Victoria University of Manchester, 2004. `http://oprofile.sourceforge.net/doc/index.html`.

[37] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[38] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[39] Microsoft Corporation. *NET Framework 3.5*, February 2009. `http://msdn.microsoft.com/en-us/library/w0x726c2.aspx`.

[40] Scott G. Miller. Sisc: A complete Scheme interpreter in Java. Technical report, 2003.

[41] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[42] Charles Nutter, Thomas Enebo, Ola Bini, and Nick Sieger. Jruby. `http://jruby.codehaus.org`.

[43] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolayv Mihaylo, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

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

[45] Eduardo Pelegrí-Llopart and Susan L. Graham. Optimal code generation for expression trees: an application burs theory. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 294–308, New York, NY, USA, 1988. ACM.

[46] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C–: A portable assembly language that supports garbage collection. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, pages 1–28, London, UK, 1999. Springer-Verlag.

[47] Mark Probst. Proper tail recursion in C. Master's thesis, Technische Universität Wien, 2001.

[48] Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

[49] John Rose. Da Vinci Machine project, February 2009. `http://openjdk.java.net/projects/mlvm/`.

[50] Michel Schinz and Martin Odersky. Tail call elimination on the Java Virtual Machine. In *Proceedings of the ACM SIGPLAN BABEL Workshop on Multi-Language Infrastructure and Interoperability*, pages 155–168. Elsevier, 2001.

[51] Michael D. Schroeder and Saltzer Jerome H. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278– 1308. IEEE Computer Society, 1975.

[52] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Proceedings of the Second International Symposium on Static Analysis*, pages 366–381. Springer-Verlag, 1995.

[53] IEEE Computer Society. *IEEE Standard for the Scheme Programming Language*. The Institute of Electrical and Electronic Engineers, Inc., New York, USA, ieee std 1178-1990 edition, 1991.

[54] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[55] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, 1992.

[56] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, New York, NY, USA, 1984. ACM.

[57] Dan. S Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63. IEEE Computer Society, 1998.

[58] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.

[59] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141, New York, NY, USA, 2005. ACM.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.