TailCalls

Patch name: tailc.patch

> **Prototype**
>
> Work in progress.

Tail call support in Davinci

Tail call optimization guarantees that a series of tail calls executes in bounded stack space. No StackOverflow exception happens.

After applying the tail call patches Hotspot supports tail call optimization for all method invocations that are marked as tail call. The optimization is performed by the VM if the invocation is marked as a 'tail call' and the necessary conditions for tail calls are met. That is the VM supports *hard tail calls*. If the VM can not guarantee that the innvocation will be optimized it throws an exception. Note that the VM will **not** recognize normal calls that could be tail call optimized and perform the optimization on them.

To enable tail call optimization the VM is started with the flag -XX:+TailCalls.

java -XX:+TailCalls TailCallExample

Tail call optimization is implemented by replacing the caller's frame with the callee's frame if the java security mechanism allows it. If removing the caller's frame is not possible, the VM proceeds in two possible ways depending on whether the 'TailCallStackCompressions' flag is set or not.

- TailCallException is thrown. If the VM detects that the security information between caller and tail called callee differs it throws a TailCallException. The security information differs if the class holding the caller method has a different protection domain than the class holding the callee. This behaviour is enabled by default e.g the flag -XX:+TailCallsStackCompression is not set.
- Tail call is defeated. The stack frame of the caller is left on the stack. If the recursion depth is deep enought the stack becomes full. At this point, just before throwing a StackOverflow exception the VM tries to 'compress the stack' by removing superfluous stack frames. To enable this behaviour the flag -XX:+TailCallsStackCompression has to be set.

Marking a call as a 'tail call'

To mark a method call as tail call the invocation bytecode instruction has to be prefixed with a 'wide' (196) bytecode.

```
public static int tailcaller(int);
  Code:
   0:        iload_0
   1:        ifne        8
   4:        iload_0
   5:        iconst_1
   6:        iadd
   7:        ireturn
   8:        iload_0
   9:        iconst_1
   10:        isub
   11:        wide
   12:        invokestatic        #2; //Method tailcaller:(I)I
   15:        ireturn

}
```

After applying the langtools-tailcall-goto.patch (to the langtools directory) we can mark calls as tail calls using the 'goto' prefix before a method invocation in java. Above bytecode example is generated by javac from following java code.

```
public class Simple {
  public static int tailcaller(int x) {
    if (x==0) return x+1;

    return goto tailcaller(x-1);
  }
}
```

Generating code

There are two bytecode verifiers in the jdk. The old type-inference verifier and the new type-checking verifier. The type-checking verifier is used if the class file format major version is >= 50. To use this verifier, a valid stackmaptable attribute has to be emitted. Tail calls are currently only supported if the new verifier is used. Hence, to use tail calls the generated class file must have a version number >= 50.0 and must contain a valid stackmaptable attribute.

The default behaviour of the jvm is to fall back to the old verifier if the new verifier fails (e.g because the stackmaptable attribute is invalid). The jvm has an option to disable this behaviour.

```
java -XX:-FailOverToOldVerifier -XX:+TailCalls GeneratedClass
```

Implementation

Some implementation details about the prototype can be found in Arnold Schwaighofer's thesis:

http://www.ssw.uni-linz.ac.at/Research/Papers/Schwaighofer09Master/

There currently exist two branches

- tailc-eager.patch: Compiled code (by server, client compiler) moves the callee's arguments to their actual position at the call site as part of the argument lowering.
- tailc-lazy.patch: Compiled code moves the callee's arguments to their actual position at the callee's method entry. Hence arguments are moved twice. First to the outgoing argument of the caller area at the call site. Later the tail call method entry code moves the arguments to the caller's incoming argument area.

Stack walking and frame elision

A successful tail call removes the frame of the caller **C1** and replaces it with the callee frame **C2**. This process can repeat an arbitrary number of times, with each successive frame **C(k)** performing recursive calls before it tail-calls the next frame **C(k+1)**. Unless the thread loops infinitely, the sequence of tail-called frames ends eventually when one callee **C(Max)** returns to its caller **P0** instead of performing a tail call. The frame **P0** returned to may be called the *parent*, and is second-youngest frame on the stack at the point of every tail call in the sequence.

If some **C(k)** is interrupted and the stack is inspected in the debugger, the earlier frames **C(1)..C(k-1)** will not be visible unless the system has taken special care to record their historical presence for the benefit of the debugger. It might be reasonable to define debugging parameters which request the JVM to record (at extra cost) the presence of some of those historical frames.

Thread stacks can also be viewed with the API call `Throwable.getStackTrace`. The same considerations apply as for debugger display of frame, except that there must be a default setting in production mode (with no debugging) which is generally agreed upon. Recording no historic frames at all is almost certainly the correct default setting. If historic frames are recorded at all in production modes, then they would probably require special presentation in the stack trace. It is unlikely that historic frames will be retained or displayed in any initial version of tail call support.

A very specialized version of stack walking is performed for a limited number of system methods like `Class.forName`, which are sensitive to their immediate caller, and behave as if that caller (as a `Class` object) were passed as an extra invisible parameter. In recent versions of the JDK, these methods are clearly marked using the internal annotation `@CallerSensitive`, and can be reliably processed as special cases. An attempt to link to one of these methods from a tail call site should fail with a linkage error.

The API call `AccessController.getContext` also gives a view derived from the control stack. This view cannot omit the effects of historical frames, if the presence of those frames would affect access control checks. The information content of an access control context is a set of protection domains, each one derived from the class loader of a method in a pending frame. (There are other details, such as user-supplied protection domains and stack boundaries, but they are irrelevant to this discussion.) The set of protection domains is unordered and indifferent to repetitions. (That is, it is really a set.) The most important property of this set is that, if a stack frame is contributing to the current computation, the protection domain from that stack frame must be present in the set. Crucially, this must be true even of the historical frames **C(1)..C(k-1)**.

Options for implementing access control tracking

Suppose the protection domains for the various frames **C(1)..C(k)** are **PD(1)..PD(k)**.

There are several basic requirements:

1. Any protection domains that apply to **P0** and its callers (and/or other surrounding context) must be retained through all execution of **P0** and **C(1)..C(k)**.
2. For any recursive call from **C(k)** that performs access control checking, the protection domains **{PD(1)…PD(k-1)}** must also be available for access checks.
3. During the same recursive call from **C(k)**, **PD(k)** must also be available.
4. At every tail call, a new call to **C(k+1)**, the protection domain **PD(k)** must be added to the existing set **{PD(1)…PD(k-1)}**.

Requirements 1 and 3 do not need any special provision, since they just restates the normal effect of pending frames on all normally recursive JVM calls.

Requirement 2 implies that there is a place where the set **{PD(1), …, PD(k-1)}** accumulates. (It is initially empty, during **C(1)**.) Without tail calls, it would naturally accumulate as the pending stack frames **C(1)..C(k-1)** accumulate, but the precise effect of a tail call is that these frames are only past history.

Requirement 4 implies that the history actually accumulates in the provided place. Call this accumulated set **A(k) = {PD(1)…PD(k-1)}**. Thus, **A(k)** is the set of protection domains which is recorded during the execution of **C(k)**, and which records the historical effects of the previous **C(i)**.

These requirements can be weakened slightly by observing that A(k) does not need to contain a duplicate entry for **PD(k)**, since **C(k)** already provides that protection domain. This observation leads to a simple, localized optimization. The last entry **PD(k-1)** can be held back from **A(k)**, as long as **PD(k-1)=PD(k)**. When setting up the next call to **C(k+1)**, the held-back **PD(k-1)** must be added to **A(k+1)**, unless **PD(k)=PD(k+1)**, in which case the protection domain can continue to be held back. This implies that there can be a "fast path" for calls from **C(k)** to **C(k+1)**, in the common case where the tail-caller and tail-callee are in the same protection domain.

(The current mlvm prototype detects "shifts" where **PD(k)PD(k+1)** and throws an exception. These exception paths are where summarization must take place instead.)

A likely place to maintain **A(k)** would be an extra hidden frame between the parent frame **P0** and the first tail-called child **C(2)**. Thus, when taking down **C(1)** during the first tail call, the hidden frame (initialized with **A(2)**) would be pushed above **P0**.

For optimized code which inlines a sequence of **C(i)**, the compiler must record an equivalent to the **A(i)** sets. Again, this could be done by extra hidden frames.

It is likely that the sets **A(i)** will generally be small and stable through a long series of tail calls. The worst case would be a loop through a long list of nodes, each of which is defined in a distinct protection domain, an unlikely scenario. More likely is a user protection domain and one or more system-level domains for library types. In that case the loop will quickly stop adding new elements to the set. It is likely that a direct vector on the stack of protection domain references is an effective representation.

Another possibility is to keep the frames **C(i)** until some consolidation point is reached (say, a stack depth limit) and "compress" the stack when it overflows. The historical frames will need to be marked so that they can be hidden as required.

In the case of a megamorphic virtual or interface tail call, the check for protection domain shifting (**PD(k)PD(k+1)**) must be incorporated into the method dispatch. For monomorphic call sites, the check for shifting can be done once, at link time. In all cases, if the check detects a shift, there must be a slow path which locates **A(i)** stored nearby and adjoins the caller's protection domain to the set (if it is new to the set). There must also be an even slower path which creates a recording point if needed.

Further discussion of the problem of access control may be found in a paper by Clements and Felleisen, linked here: http://mail.openjdk.java.net/pipermail/mlvm-dev/2013-July/005417.html