Lazy Continuations for Java Virtual Machines

Lukas Stadler* Christian Wimmer† Thomas Würthinger*
Hanspeter Mössenböck* John Rose§

*Johannes Kepler University Linz, Austria †University of California, Irvine §Sun Microsystems, Inc.

{stadler, wuerthinger, moessenboeck}@ssw.jku.at
cwimmer@uci.edu john.rose@sun.com

ABSTRACT

Continuations, or 'the rest of the computation', are a concept that is most often used in the context of functional and dynamic programming languages. Implementations of such languages that work on top of the Java virtual machine (JVM) have traditionally been complicated by the lack of continuations because they must be simulated.

We propose an implementation of continuations in the Java virtual machine with a lazy or on-demand approach. Our system imposes zero run-time overhead as long as no activations need to be saved and restored and performs well when continuations are used. Although our implementation can be used from Java code directly, it is mainly intended to facilitate the creation of frameworks that allow other functional or dynamic languages to be executed on a Java virtual machine.

As there are no widely used benchmarks for continuation functionality on JVMs, we developed synthetical benchmarks that show the expected costs of the most important operations depending on various parameters.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters, run-time environments*

General Terms

Algorithms, Languages, Performance

Keywords

Java, virtual machine, continuation, stack frame, activation, optimization, performance

1. INTRODUCTION

With the increasing success of dynamic languages such as Ruby or Python, the interest in continuations has been renewed. Also, web frameworks such as RIFE [12] or the PLT Scheme web server [8] propagate the use of continuations for web development. Continuations, or more precisely *first class continuations*, can be seen as 'the rest of the computation', or the rest of the program as it would run from the point where the continuation was created.

In a more practical sense a continuation is the state of a running thread that can be stored, possibly modified, and reinstated. Continuations can be categorized as follows [5]:

First-class continuations store the full state of the computation and can be reinstated any number of times from anywhere in the program.

Delimited continuations (also known as partial or composable continuations) turn a part of a continuation into a function that can be called any number of times. Scheme, for example, uses `reset` and `shift` [4] to create delimited continuations.

One-shot continuations (or exit continuations) work as non-local exit statements and can basically be seen as a `return` that exits multiple method scopes at once.

Mobile continuations (which can be resumed on a different thread or even on a different virtual machine) can in the context of this paper be seen as a special case of delimited continuations. The implementation presented in this paper is primarily concerned with first-class continuations, although it deals efficiently with one-shot continuations as well and could easily be expanded for delimited continuations.

In the context of Java VMs, storing a continuation can be seen as storing all stack frames of the current thread with their local variables and intermediate computation results. However, in most situations it is unnecessary to store all stack frames, because when a continuation is reinstated some of the frames on the stack are still the same as in the continuation and do not have to be restored. We therefore propose a lazy approach where a stack frame is only stored in a continuation if it would otherwise be destroyed or modified. In other words, we store a stack frame only when the control flow returns to the method to which the frame belongs.

We implemented our approach for the interpreter and the client compiler of the Java HotSpot™ VM, but in principle it can also be implemented for other Java VMs or managed runtimes of other languages. This paper contributes the following:

- We present an algorithm to lazily store the contents of a continuation. This algorithm exploits the fact that many continuations share activation frames.

© ACM, 2009. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the Proceedings of the 7th International Symposium on Principles and Practice of Programming in Java, pp. 143–152.
PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.
http://doi.acm.org/10.1145/1596655.1596679

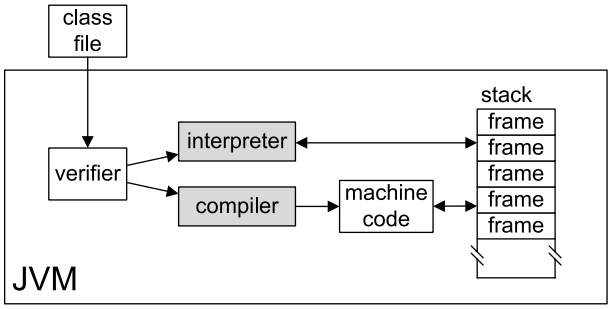

Figure 1: System overview

- We show the modifications of the Java HotSpot™ VM needed to implement the algorithm.
- We outline the characteristics of our algorithm with an evaluation of the run-time behavior of our implementation.

This paper consists of the following parts: Section 2 provides a brief description of all the relevant parts of the Java HotSpot™ VM. Section 3 describes the actual details of our implementation, followed by an evaluation of how this implementation can be used in Java code or by language frameworks in Section 4. Section 5 provides estimations for memory consumption and execution time. Section 6 presents related work, and Section 7 concludes this paper.

2. SYSTEM OVERVIEW

We developed our implementation as an extension to the Java HotSpot™ VM [14]. It is part of a larger effort to extend this JVM with first-class architectural support for languages other than Java (especially dynamic languages), called the *Da Vinci Machine Project* or *Multi-Lanugage Virtual Machine Project* (MLVM) [15].

2.1 Java Virtual Machine

A Java virtual machine (JVM) loads, verifies, and executes Java bytecode. It needs to adhere strictly to the semantics defined by the Java virtual machine specification [9]. The Java HotSpot™ VM in particular can either interpret the bytecodes or use one of its two just-in-time (JIT) compilers, called *client compiler* [7] and *server compiler* [11], to compile bytecodes into optimized machine code. It decides at run time on a per-method level if the compilation overhead is justified, because in normal applications most of the execution time is concentrated in a few frequently called methods, known as *hot spots*.

In order to achieve maximum performance, all JVMs use the CPU-supported stack to manage local variables and expression stacks. All the information belonging to one execution of a method is called an *activation frame*. The stack containing these activation frames grows and shrinks in defined directions, and obsolete activation frames are overwritten automatically without the need for explicit management. In addition to the local variables and expression stacks, the stack also holds the *bytecode index* (bci) or *program counter* (pc) for each activation frame such that it can be restored upon a return instruction.

Figure 1 gives an overview of how bytecode is loaded into the JVM and how it interacts with the stack. The interpreter and the compiler contain the main parts of our implementation.

```
Queue<Continuation> queue;
public void yield() {
  Continuation cont = new Continuation();
  if (cont.capture() == Continuation.CAPTURED) {
    queue.add(cont);
    queue.remove().resume(null);
  }
}
public void coroutine(int n) {
  int i = 0;
  while (true) {
    println("coroutine " + n +
            " iteration " + i);
    yield();
    i++;
  }
}
public void start() {
  for (int i = 0; i < 10; i++) {
    Continuation cont = new Continuation();
    if (cont.capture() == Continuation.CAPTURED) {
      queue.add(cont);
    } else {
      coroutine(i);
      // never reached because of while loop
    }
  }
  queue.remove().resume(null);
}
```

Figure 2: Coroutines implemented via continuations

2.2 Continuations

Continuations can be seen as dynamic labels, comparable to a label jumped to by a `goto` statement. The creation of a continuation works like a `fork` on the thread, but the new context is stored in the continuation instead of run in parallel. A prime example for continuations is the `call-with-current-continuation` (short: `call/cc`) construct of the Scheme programming language [1]. `call/cc` creates a function that can restore the thread to the state after the call to `call/cc`. Scheme was one of the first programming languages to establish the concept of continuations. There are two main operations that define continuations:

capture creates a continuation: the *dynamic* execution context of the current thread is stored.

resume reinstates the given continuation. This means that the thread resumes execution at the instruction following the capture call that created the continuation. The stack of activation frames is restored to exactly the same state as it was after the continuation was captured.

A complete implementation of continuations, like the one presented in this paper, allows resume to be called *multiple times* on the same continuation and regardless of the current state of the stack. Many other implementations restrict continuations to one resume call that has to take place as long

as all activation frames of the continuation are still present on the stack.

Figure 2 shows an example that uses continuations to implement coroutines [10]. Coroutines are a form of explicit, cooperative multitasking where multiple tasks run in parallel using only one operating system thread. The `yield` method captures the current continuation, enqueues it, and resumes the continuation of the first waiting coroutine, thereby switching to this coroutine. The `start` method creates ten coroutines by creating their initial continuation, and then starts the first one. Continuations are initially resumed at the `capture` call in the `start` method and enter the else branch of the if statement. It is important to note that the `capture` method returns `CAPTURED` whenever it created the continuation, and the value passed to `resume` if it was resumed. The API used to capture and resume continuations is explained in detail in Section 4. The example produces the output shown in Figure 3.

The `while` loop of each coroutine runs independently from the others, i.e., the local variable `i` is independently managed. After each loop iteration, the execution is paused and control is transfered to the next coroutine. The coroutines therefore run intermingled, i.e., ten instances of the `while` loop are managed at the same time, but only the stack frame of one of them is active. The other nine stack frames are saved in continuation objects. Therefore, the coroutines do not run in parallel, as it would be when each of them was started in its own thread. This reduces the overhead of thread creation and scheduling, which is crucial if the number of coroutines is high.

Neither capturing nor resuming continuations is defined by the Java virtual machine specification, and up to now only research projects have implemented continuations in the context of a Java virtual machine. Continuation support necessarily breaks JVM invariants. For example, code that is usually executed only once can be run multiple times, and `finally` blocks can be skipped. This means that any method that is included in a continuation needs to be aware of this and needs to be marked (see Section 4.3). Also, methods and blocks holding locks via the `synchronized` keyword cannot be included in a continuation because it is not clear if and when the runtime system should release and reacquire these locks.

In general, a call site in a method may be associated with matched pairs of set-up and take-down actions, to be executed whenever control enters and leaves that point, either via normal control flow or via continuation processing. This structure may be observed explicitly in the Scheme DYNAMIC-WIND function, which surrounds the computation within a call site with matched pairs of "before" and "after" actions. In the JVM bytecode architecture, the take-down actions are specified by "finally" blocks, but there are no explicit "initially" blocks to make the matching set-up actions explicit. When a continuation exits a JVM frame, it might often be reasonable to execute the "finally" blocks associated with the pending call, but there is no way to be sure of the programmer's intent, and there is certainly no corresponding way to determine the matching set-up actions intended by the programmer. Because the present resume functionality is based on bitwise copying of stack frame images, it does not address these problems at all. Language frameworks built using a continuation implementation need to deal with them.

```
coroutine 0 iteration 0
coroutine 1 iteration 0
[...]
coroutine 9 iteration 0
coroutine 0 iteration 1
coroutine 1 iteration 1
[...]
```

Figure 3: Output of coroutines example

```
Continuation alpha = new Continuation();
Continuation beta = new Continuation();
void main() {
  a();
}
void a() {
  b();
}
void b() {
  alpha.capture();
  // ...
  beta.capture();
}
```

Figure 4: Example source code

3. IMPLEMENTATION

In the context of stack-based activation frame management, capturing a continuation means to store all or part of the stack such that it can be restored later on, because some activation frames might have been modified or overwritten. The runtime system needs to provide methods and data structures for storing and retrieving activation frames.

The Java program of Figure 4 captures two continuations. The two continuation objects are created in advance, and the actual capturing is done via the `capture` method. To simplify matters we use this stripped-down example throughout most parts of this paper instead of the one in Figure 2.

Figure 5 shows the evolution of the runtime stack of the example code over time. The gray areas visualize when an activation frame can possibly be modified, which always applies only to the topmost frame. A simple implementation of continuation capturing could store the stack contents by performing a complete copy of the whole stack contents every time a continuation is created. The resulting self-containing continuation objects are shown on the right side of Figure 5. This simple implementation suffers from a number of drawbacks:

- The immediate costs for creating a continuation always depend on the current stack depth (up to the delimiting frame).
- In practice many continuations share all but the few topmost stack frames, and the duplication of identical frames wastes memory and run time.
- Because it is not known if a stack frame has already been modified, all stack frames need to be restored, even though this is unnecessary.

In order to minimize the memory and run-time costs, the actual saving of activation frames needs to be delayed as

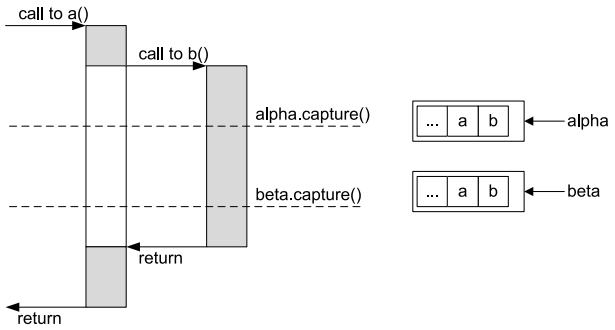

Figure 5: Simple continuation capturing

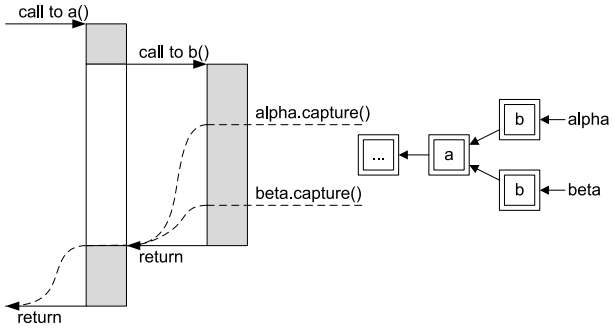

Figure 6: Optimized continuation capturing

long as possible. Frames need not be saved immediately when it is guaranteed that they will be saved before they are modified. Modifications to an activation frame cannot occur before the execution returns to the frame in question.

Intercepting the actual return instruction allows us to solve all of the aforementioned drawbacks because we can store activation frames just before they are modified. By storing activation frames in this "lazy" manner, all continuations that use a particular activation frame are automatically accumulated so that only one copy of this frame needs to be stored. Figure 6 shows how our algorithm represents continuations as a tree structure. In this example both continuations share all activation frames starting at the "a" frame.

3.1 Data Structures

The contents of a continuation are stored in so-called *activation objects*, three of which are shown in Figure 7. A continuation is represented by a reference to its first activation object, which is called the continuations *topmost activation object*. Activation objects contain the following fields:

next frame This field is used to denote the next activation object in the continuation, and thus it creates a linked list of activation objects. When multiple continuations share activation objects the linked lists merge into a tree structure.

method The method of the activation frame that is contained in this activation object.

bci / pc The pointer to the next instruction, either as a bytecode index for interpreted methods, or a program counter for compiled machine code.

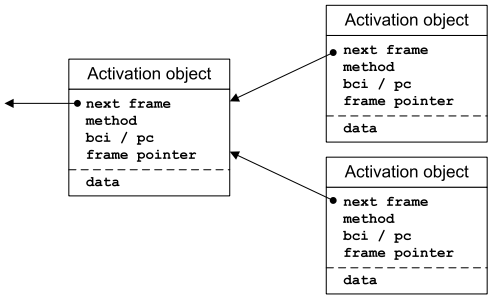

Figure 7: Layout of the activation objects

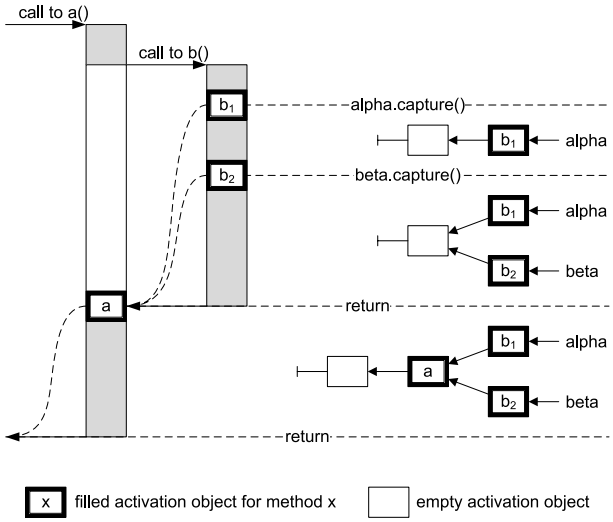

Figure 8: Evolution of the inverse tree structure

frame pointer The position of the activation frame on the stack when it was last seen on-stack. This information is needed for joining continuations (see Section 3.3).

data The data field stores a plain copy of the stack contents of the activation frame. Copying byte-by-byte is fast and simple. If we need to inspect the contents of the activation object (for garbage collection, etc.) the JVM methods for inspecting activation frames are used. During garbage collection all object pointers contained in the activation frame data need to be visited because otherwise the referenced objects might be reclaimed. If referenced objects are moved, the stack frame must be adjusted to contain their new addresses. The drawback of byte-by-byte copying is that activation frames from compiled methods and activation frames from interpreted methods need to be treated differently because they have a different structure.

An activation object can exist without actually containing any data. In this case it acts as a *skeleton* object that can later be filled with the data of the activation frame.

Figure 8 shows how the activation objects evolve into the structure shown in Figure 6. In this case the implementation recognizes that both continuations refer to the same activation frame for the method `a`, and thus can share the activation object. When `alpha.capture` is called, a new activation object is created and the frame of method b at

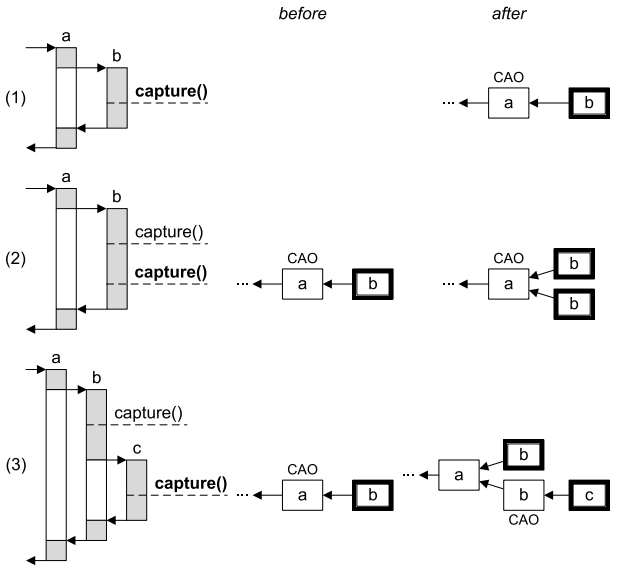

Figure 9: Capturing continuations

position b_1 is stored into it. Since there is no activation object for the caller's frame yet, we create a skeleton object and link b_1's activation object to it. When `beta.capture` is called, we create another activation object and fill it with the frame at position b_2. Since there is already an activation object for the caller (the skeleton object), we link b_2's activation object to it, thus creating a tree. When method `b` returns, we intercept this event and fill the skeleton object of `a` with the corresponding frame contents and in turn create a new skeleton object for `a`'s caller.

Having a skeleton object for the next activation frame that needs to be stored (i.e., the caller's frame) serves two purposes:

- Skeleton objects represent activation objects that are still unmodified on the stack. If a continuation is resumed and it uses a skeleton object, the runtime knows that it does not need to restore this frame.
- If the skeleton object did not exist, the runtime would have to maintain a set of activation objects that share the caller's activation frame. The `next` pointers of the objects in this list would have to be changed to the caller's activation object as soon as it is created.

3.2 Capturing Continuations

In our approach the frames of a continuation are copied lazily at the latest possible time. We just copy the current frame, and when the control flow returns to the caller we copy the caller's frame as well. As explained above, the caller's activation object already exists when the callee's activation object is filled, so the question arises where to store the reference to the caller's activation object. As a simple solution we store it in the thread object of the current thread and refer to it as the *thread's caller activation object* (CAO). In Figure 8 the CAO is always an empty activation object, although this is not necessarily true in all cases.

When instructed to capture the current continuation the `capture` method creates an empty activation object for the current frame and *patches* its own return address, i.e., it replaces it with the address of a small call-site-specific trampoline. When `capture` returns, the trampoline code is executed, which fills the activation object with the frame of the method that called `capture`, and then jumps back to this method. This is explained in more detail in Section 3.3.

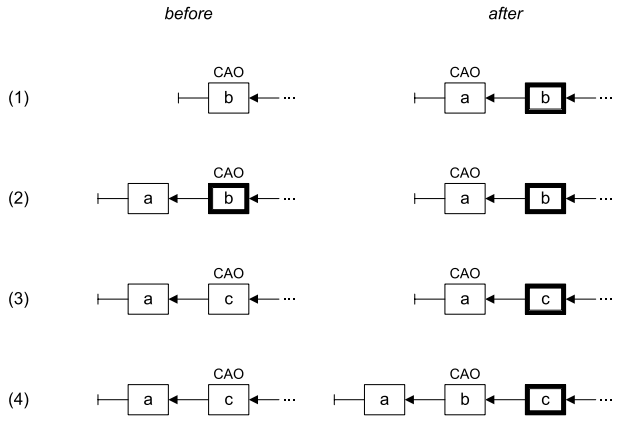

Figure 10: Storing activation frames

Linking activation objects can be more complicated than in the simple case above. Figure 9 shows the three cases that can occur. In case (1) no continuation was created so far, thus there is no CAO yet. `capture` not only creates an activation object for b's frame but also an empty skeleton object as the CAO. In case (2) there is already a continuation, thus the CAO already exists. `capture` creates an activation object for b's frame and links it to the existing CAO. In case (3) there is also already a continuation, but it was created in the caller of the method where we now create the new continuation. `capture` not only has to create an activation object for c's frame but also an activation object for b's frame, which becomes the CAO. Note that there may be more not yet stored activation frames in between, for which activation objects would be created one-by-one on return.

3.3 Storing Activation Frames

The runtime creates a trampoline for each call site within a compiled method that are marked as `@Continuable` (see Section 4.3). There is only one instance of the trampoline for interpreted methods because interpreter activation frames are verbose enough so that all information can be deduced from the frame itself. The trampoline code that is executed during a patched return to a method M has the following tasks:

- It fills the CAO, which is guaranteed to exist, with the frame contents of M.
- It possibly creates an empty skeleton object and installs it as the CAO.
- It patches the return address of M with the address of another trampoline.

Figure 10 shows the four different cases that can occur:

- In the simplest case (1) the `next` pointer of the CAO is empty, and this also implies that the CAO is not yet filled. The CAO is filled now and a new one is created and appended to the list. Finally the CAO is advanced.

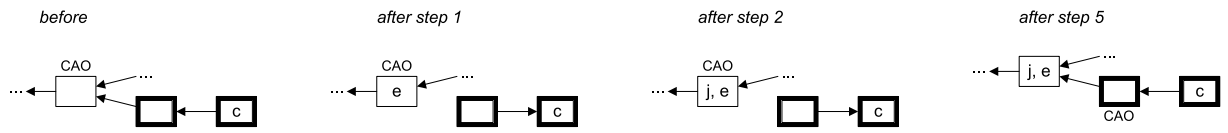

Figure 11: Resuming continuations (simple case)

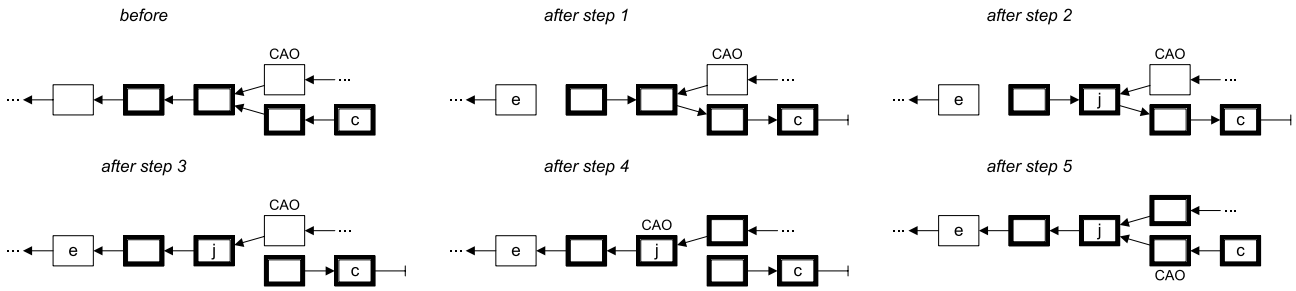

Figure 12: Resuming continuations (complex case)

- In (2) the CAO is already filled. This happens when a continuation has been resumed, i.e., the activation frame on the stack has been created from the activation object in question. Under this premise the `next` pointer can never be null, and the pointer in the thread is simply advanced to the next object without creating any new objects.

- In the most complex cases (3) and (4) the CAO is empty and its `next` pointer is not null. First the CAO is filled, and then the frame pointer stored in the next activation object (here "a") is compared with the frame pointer if the caller's activation frame. If it is the same (3), the CAO is simply advanced, otherwise (4) a new skeleton object is inserted just after the CAO; this object becomes the new CAO.

Whenever the algorithm creates a new activation object, it also needs to store an activation frame into it later. Therefore it needs to patch the return address to this activation frame. If no new object has been created then no patching is needed. This guarantees that the return address has already been patched to the call-site-specific trampoline whenever an activation frame of interest for a continuation gets active.

3.4 Resuming Continuations

In the tree of activation objects, each continuation can be seen as one concrete path from the topmost activation object of the continuation (i.e., a leaf) to the root. This path is described by traversing the continuation using the `next` pointers of the activation objects. As the current point in the execution of a program can also be seen as a continuation, and thus as a list of activation objects, reinstating a continuation can be performed by transforming the current list of activation frames to the continuation's list of activation frames. We achieve the smallest number of changes by searching for the nearest common ancestor of the CAO and the topmost activation object of the continuation.

Figures 11 and 12 show two examples of transformations that resume a continuation. A few activation objects are marked: "c" is the topmost frame of the continuation to be reinstated, "j" is the object at which the CAO and the continuation are joined and "e" is the first empty activation object ("skeleton") when the list of activation objects of the continuation is traversed.

In order to resume the continuation, and thus transform the activation stack, all or some of the following steps are performed:

1. Traverse the activation objects starting from the topmost frame of the continuation until an empty activation object is encountered. During the traversal the `next` pointers are modified such that the list is reversed. Additionally the activation objects are marked as being traversed. During this step "e" is discovered.

2. Traverse the activation objects starting from the CAO of the thread until either "e" or an activation object marked during step 1 is hit. This leads to "j".

3. If "j" and "e" are not the same object (as in Figure 12) then the activation objects between these two do not need to be restored, thus we can restore the `next` pointers for these objects.

4. The CAO is advanced towards "j". Frames are removed from the stack, and if the corresponding activation objects are still empty skeletons, the contents of these frames are saved in their activation objects.

5. Now that the thread is at the join point "j", the activation objects from "j" to "c" are reinstated onto the stack and the thread now points to the activation object after "c".

Note that after a continuation is resumed the thread now points to previously filled activation objects. These filled objects can quickly be skipped during capturing without the need to fill them again (see case (2) in Figure 9).

4. API FOR CONTINUATIONS

We have designed the API of our implementation with two main goals in mind: to allow for an efficient implementation and to be easy to use for a language implementation framework. All methods are accessible via a single

```
public class Continuation {
  public static final Object CAPTURED;
  public native Object capture();
  public native void resume(Object retVal);
}

public @interface Continuable {
}
```

Figure 13: API for continuations

class `Continuation`, which makes it easy for frameworks to generate calls. Figure 13 shows the API for our continuation implementation: the `Continuation` class and the `@Continuable` annotation.

4.1 Method Capture

The method `c.capture()` associates the continuation `c` with the current thread state. It can be called on the same `Continuation` object multiple times, and the object always refers to the continuation of the last `capture` call. As long as `capture` has not been called, the `Continuation` object is in an empty state, and calling `resume` on it leads to a runtime exception. The return value of the `capture` call depends on the situation: if a continuation was just created then the `CAPTURED` marker object is returned, and if a continuation was resumed then the value that was given to `resume` is returned.

4.2 Method Resume

The method `resume` allows a continuation to be reinstated. It takes one parameter: an object that is returned by the `capture` call that created the continuation. If the resumed continuation is invalid, a runtime exception is thrown. Continuations can be invalid for the following reasons:

- `capture` has never been called on the `Continuation` object, and thus there is no continuation to resume.
- The continuation to resume belongs to another thread. Migrating continuations to another thread is not implemented in our current model.
- The activation frame storage code encountered an invalid activation frame as explained in Section 4.4.

Additionally, a runtime exception is thrown if `CAPTURED` is passed as a return value.

A call of the method `resume` never returns to the code after the call, i.e., code that is located immediately after a call of `resume` is always dead code. Instead, execution resumes after the call of the method `capture` where the continuation was defined. Code that should be executed after resuming a continuation must therefore be placed after the call of `capture`.

4.3 Annotation @Continuable

The runtime has to assume that including methods in continuations that are not aware of this is unsafe and likely leads to undefined behavior. Thus the runtime needs a way to determine if a method is designed with continuation in mind. We introduce the annotation `@Continuable` to mark methods as safe for use within continuations. If a type (class, interface, or enum) is marked with `@Continuable` then all methods implemented within it (but not inherited methods or methods added by subclasses) are considered safe.

4.4 Invalid Activation Frames

An activation frame is considered invalid if one of the following applies:

- Its method was not marked with the `@Continuable` annotation and thus is not safe to be used within continuations.
- The activation frame currently holds a lock that was acquired via the `synchronized` keyword. Methods that are declared `synchronized` always give rise to invalid frames. Methods that use a `synchronized` block can lead to both valid and invalid activation frames, depending on the current position.
- The activation frame belongs to a native (JNI) method. The JVM has not enough information about native method activation frames to store and restore them. This also applies to some reflective methods such as `Method.invoke` that, depending on the implementation, use native calls.

The activation frame storage mechanism (see Section 3.3) checks if an activation frame is valid before storing it in an activation object. If it is not valid then the activation object is marked as invalid, and any attempt to resume a continuation containing it will fail. Furthermore the return address of the invalid activation frame is not patched and thus no more frames are stored for the invalid continuations.

As long as the invalid activation frame does not need to be stored, the continuations are not invalid, because the invalid activation frame is still on the stack and thus not restored. This means that while a `Method.invoke` is an invalid activation frame, the method that it calls can still use continuations. All these continuations become invalid as soon as the control flow returns to the `Method.invoke` method.

Invalid activation frames can be used on purpose to implement *delimited continuations*. Continuations that have been captured with an invalid activation frame on the stack will be automatically and reliably invalidated as soon as the execution returns to the invalid activation frame.

5. EVALUATION

All tests and benchmarks presented in this section were performed on a computer with a dual-core Intel Pentium D processor at 2.8 GHz and 4 GByte main memory running Microsoft Windows XP.

5.1 Memory Consumption

Activation frames, when stored on the heap, take up more space than they do when stored on the stack because they need to be wrapped in an object that is manageable by the garbage collector. Storing and restoring a large number of continuations likely puts a large strain on the memory system so it is important to characterize the impact of our approach with respect to memory consumption.

Compiled method size

For each compiled method that is marked with the `@Continuable` annotation, about 40 bytes are needed

for the call-site-specific trampoline and for some additional control information.

Continuation size

An activation frame stores the local variables, intermediate expressions, and some additional data such as return addresses. The size of an activation frame, and thus the size of a continuation, depends on the optimizations the compiler performs. Despite this the size of a continuation can be approximated as (in `words`): 8 + 8 * [number of stackframes] + [number of local variables] + [number of expression stack elements]

Comparison to the copy-all approach

While our approach uses more memory for single continuations than the simple copy-all approach, it benefits from the fact that activation objects can be, and in practice will be, shared among multiple continuations. If we assume 8 words of bookkeeping information per activation object and an average activation frame size of 16 words then simple calculation shows that for practical stack depths (5-50) the break-even point in memory consumption is reached as soon as the reuse of stack frames reaches 27-33% on average.

5.2 Execution Time

For measuring the performance of continuations we designed a synthetic benchmark that performs a number of recursive method calls where each method has a certain number of local variables and then captures a continuation. Then the methods return and their corresponding activation frames are stored. Afterwards the benchmark resumes the continuation.

We performed this benchmark to determine the performance of our implementation depending on two parameters:

Stack depth

We expect the time of a benchmark run to be proportional to the number of activation frames it needs to process because the runtime executes the frame storage code for each individual activation frame.

Local variable count

The number of local variables should also have a linear influence on the execution time as it determines how much data needs to be copied for each activation frame.

Figure 14 shows the execution time for capturing one continuation consisting of 1 to 50 stack frames (x-axis) and for three different numbers of local variables (different lines in the chart). This test shows a linear increase of run time when the number of stack frames or local variables is increased. The numbers include the time necessary for allocating the continuation and frame objects (we verified that no garbage collection was necessary during the measurement). The costs of storing an activation frame is roughly 100 ns plus 6 ns per local variable, which amounts to about 7 times the costs of a plain method call. For comparison, the time necessary for plain method calls is shown in Figure 15.

Figure 16 shows the costs of resuming continuations. They are lower than for capturing a continuation because no memory allocation is necessary. The time for restoring an activation frame is roughly 60 ns plus 2 ns per local variable, which amounts to about 3.5 times the costs of a plain method call.

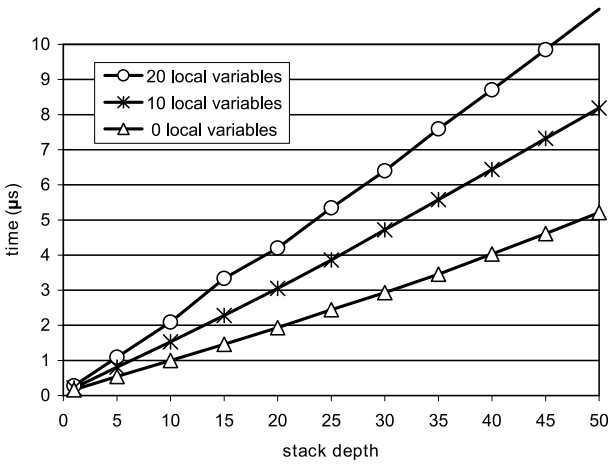

Figure 14: Costs of capturing a continuation

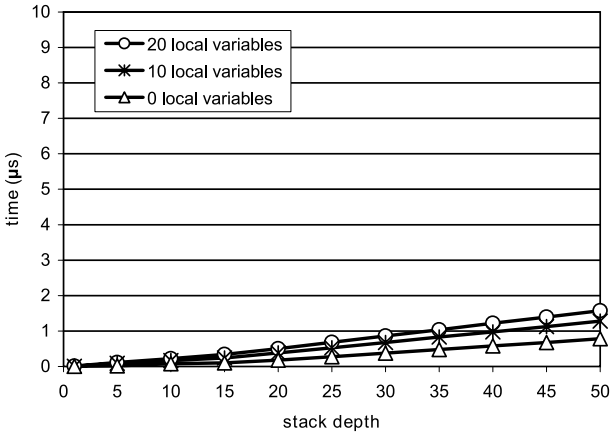

Figure 15: Costs of plain method calls

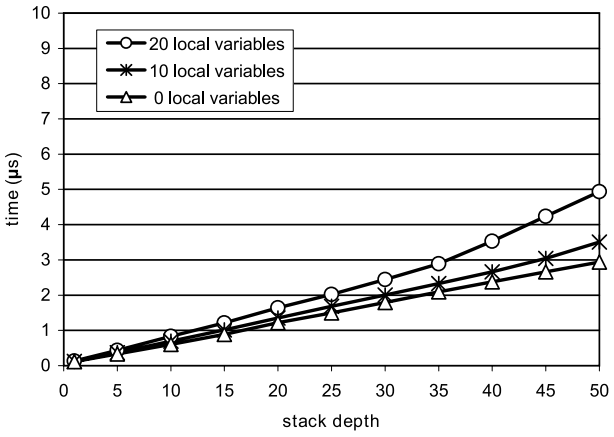

Figure 16: Costs of resuming a continuation

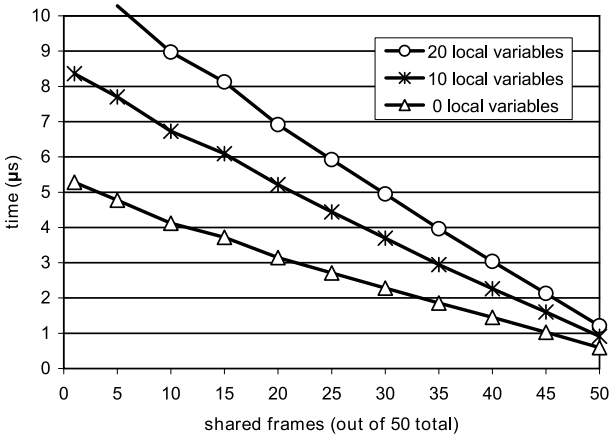

Figure 17: Benefit of shared activation frames

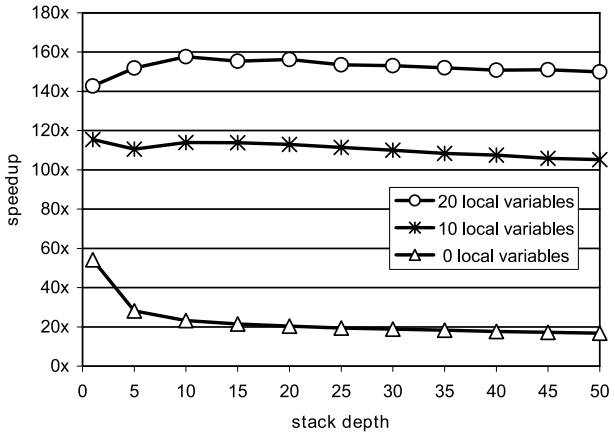

Figure 18: Speedup compared to Javaflow library

Figure 17 shows the costs of capturing continuations that share activation frames. In this case, the costs *per continuation* drops significantly. In this series of tests, 50 activation frames are on the stack, and some of them are shared among the 10 continuations created in each test. If no activation frames are shared the execution time is equal to the time needed to capture 50 frames in Figure 14.

Figure 18 shows a comparison of our implementation with the Apache Commons Javaflow implementation for continuations [2], which implements continuations via bytecode instrumentation. All code to capture the continuation is inserted into the methods whose stack frames are stored, which introduces a high overhead. We ran the same benchmarks as above, but this time using the Javaflow library instead of our continuation library. The results show that our VM-based implementation provides a significant speedup of one to two orders of magnitude.

6. RELATED WORK

Dragoş et al. implemented first-class continuations for the Ovm realtime Java virtual machine [5]. Their implementation works for the Java-to-C++ (j2c) execution engine, which is an ahead-of-time compiler. The usage of C++ as a backend forces them to do conservative garbage collection for the continuation objects. Their algorithm takes the copy-all approach and they claim an execution time for capture and resume of roughly 10 times a normal call. This conforms to the worst case results for our implementation, but their algorithm does not benefit from shared activation frames.

The Apache Commons Javaflow library [2] implements continuations without JVM support using bytecode instrumentation. The instrumentation can be performed in advance or by a special class loader, which adds complexity either to the build process or to the application itself. All affected methods are modified such that they can distinguish between normal execution, continuation capturing, and continuation resuming, which adds runtime overhead even when no continuations are used. The local variables, expressions, and position are stored in a thread-local parallel stack. This approach is not tied to a specific Java virtual machine and allows continuations to be serialized. Its performance cannot compete with VM-based implementations and it was not designed with heavy use of continuations in mind.

The RIFE Java web application framework [12] also implements continuations via bytecode instrumentation. It works similar to the Javaflow library, but it allows continuation capturing only within a specific method (`processElement`), so that there is always only one activation frame per continuation.

Many runtime environments for languages that require continuations implement them via the simple copy-all approach. One example is the original Ruby environment [13] which as of version 1.9.1 implements continuations via a copy-all approach. The native stack is copied and restored within C functions, which is very fragile to differences between C compiler implementations.

Compilers for continuation-aware languages that output Java bytecode often compromise on supporting only one-shot continuations. One example of this limited approach is the Scheme-to-Java compiler BCD Scheme [3]. This implementation approximates continuations by exceptions that are thrown when the continuation is resumed. The `call/cc` function catches the exception and returns the result that is contained within it.

Hieb et al. describe the more sophisticated implementation of continuations in the Chez Scheme environment [6]. Their approach splits the stack into segments that are allocated on the heap and lazily copied when needed. It provides runtime characteristics similar to those of our approach, but it introduces additional overhead even when continuations are not used. It also requires more radical changes to the runtime system.

7. CONCLUSIONS

The implementation of many functional and dynamic languages requires continuations. Many languages rely on efficient continuations in order to deliver acceptable performance.

In this paper we presented a new approach for capturing continuations in a lazy way and showed its implementation for the widely used Java HotSpot™ VM. While there are continuation libraries that work without changes to the Java virtual machine these libraries cannot achieve the performance of a VM-based implementation.

The main contribution of our approach is the lazy capturing of a thread's state and an optimized way of resuming this state by restoring only those stack frames that differ from the current thread state. We showed that our lazy algorithm

performs well and with predictable run-time costs and that it can benefit from common patterns of continuation usage such as sharing of activation frames.

8. REFERENCES

[1] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[2] Apache Commons. *Javaflow*, 2009. http://commons.apache.org/sandbox/javaflow/.

[3] B. D. Carlstrom. Embedding Scheme in Java. Master's thesis, Massachusetts Institute of Technology, 2001.

[4] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–160. ACM Press, 1990.

[5] I. Dragoş, A. Cunei, and J. Vitek. Continuations in the Java virtual machine. In *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. Technische Universität Berlin, 2007.

[6] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77. ACM Press, 1990.

[7] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot^{TM} client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):Article 7, 2008.

[8] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431—460, 2007.

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

[10] A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):Article 6, 2009.

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

[12] *RIFE Java web application framework - Continuations*, 2009. http://rifers.org/wiki/display/RIFECNT/Home.

[13] *Ruby Programming Language*, 2009. http://www.ruby-lang.org.

[14] Sun Microsystems, Inc. *The Java HotSpot Performance Engine Architecture*, 2006. http://java.sun.com/products/hotspot/whitepaper.html.

[15] Sun Microsystems, Inc. *Da Vinci Machine Project*, 2009. http://openjdk.java.net/projects/mlvm/.