Efficient Coroutines for the Java Platform *

Lukas Stadler† Thomas Würthinger† Christian Wimmer§

†Johannes Kepler University Linz, Austria §University of California, Irvine
{stadler, wuerthinger}@ssw.jku.at cwimmer@uci.edu

Abstract

Coroutines are non-preemptive lightweight processes. Their advantage over threads is that they do not have to be synchronized because they pass control to each other explicitly and deterministically. Coroutines are therefore an elegant and efficient implementation construct for numerous algorithmic problems.

Many mainstream languages and runtime environments, however, do not provide a coroutine implementation. Even if they do, these implementations often have less than optimal performance characteristics because of the tradeoff between run time and memory efficiency.

As more and more languages are implemented on top of the Java virtual machine (JVM), many of which provide coroutine-like language features, the need for a coroutine implementation has emerged. We present an implementation of coroutines in the JVM that efficiently handles a large range of workloads. It imposes no overhead for applications that do not use coroutines and performs well for applications that do.

For evaluation purposes, we use our coroutines to implement JRuby fibers, which leads to a significant speedup of certain JRuby programs. We also present general benchmarks that show the performance of our approach and outline its run-time and memory characteristics.

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

General Terms Algorithms, Languages, Performance

Keywords Java, virtual machine, coroutine, stack frame, activation, optimization, performance

1. Introduction

Coroutines are a well-known programming concept that allows subroutines to run in an interlocked way. A coroutine can pass control to some other coroutine using a *yield* statement.

Coroutines have also often been seen as inferior alternatives to full-blown threads, because of the manual context switching. However, there are situations in which manual context switching makes sense or is even desired (e.g., producer/consumer problems, discrete event simulation, and non-blocking server implementations). None of the top five languages of the TIOBE programming languages index [20] (C, Java, C++, PHP and Basic) supports coroutines without additional libraries.

Runtime environments such as the Java virtual machine (JVM) make it hard for library programmers to implement coroutines on top of them. The high level of abstraction and the focus on portability prevent access to the underlying machine, which is needed in order to implement low-level features such as coroutines. A coroutine system for a JVM also needs to interface with other subsystems, such as garbage collection, which is only possible from inside the JVM.

We propose a coroutine system for JVMs that uses a combination of two implementation strategies: Separate stacks are allocated for coroutines up to a certain number, and a stack copying approach is employed to allow large amounts of coroutines to be allocated.

Our system incurs only minimal changes to the JVM. Apart from the additional stack management code, it only requires a small amount of new code in the garbage collection system. Most importantly, no modifications to the existing just-in-time compilers and the interpreter were required.

We implemented our system for the Java HotSpot™ VM on the Windows and Linux platforms, which shows that it is feasible for JVMs and managed runtimes in general. This paper contributes the following:

- We present a coroutine management algorithm that combines the advantages of the two most common implementation techniques.
- We introduce a Java API for coroutines.
- We outline the modifications of the Java HotSpot™ VM that are necessary to implement coroutines.
- We show that for languages with coroutine-like features, the modified JVM vastly increases the performance of implementations that run on top of it.
- We report the run-time and memory characteristics of our algorithm.

2. System Overview

We developed our implementation as an extension to the Java HotSpot™ VM [15]. 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-Language Virtual Machine Project* (MLVM) [16].

2.1 The Java Virtual Machine

A Java virtual machine loads, verifies, and executes Java bytecodes. It needs to adhere strictly to the semantics defined by the Java vir-

* This work was supported by Oracle.

PPPJ '10, September 15–17, 2010, Vienna, Austria.

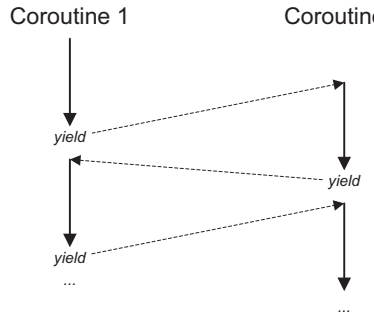

Figure 1. Interlocked coroutine execution

tual machine specification [8]. 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* [12], 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, most JVMs today 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* or *stack 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.

In order to conform to the specification, JVMs need to implement automatic memory management (garbage collection). The Java HotSpot™ VM contains a sophisticated memory management system that performs *precise garbage collection*, which requires the exact size and layout of an object and all object pointers within it to be known to the runtime system. Every object is preceded by a two-word header that contains a pointer to the class of the object and additional information, such as locking and garbage collection bits.

Garbage collection marks all objects that are reachable from a set of *root pointers* as alive. The VM has to consider a number of sources for these root pointers, e.g., activation frames, JNI handles, and compiled code.

2.2 Coroutines

Coroutines [10] are a concept that was first described in 1963 by Conway [3] as "each module ... may be coded as an autonomous program that communicates with adjacent modules as if they were input and output subroutines". They are present in numerous programming languages, such as Go, Icon, Lua, Perl, Prolog, Ruby, Tcl, Simula, Python, Modula-2, and many others. The runtime environments of some of the most popular modern programming languages, such as C# and Java, however, do not provide coroutine functionality out-of-the-box.

From a language design perspective, coroutines are a generalization of subroutines. When they are invoked, they start execution at their first statement. A coroutine can transfer control to some other coroutine (typically using a *yield* statement). The state of the local variables at the transfer point is preserved. When control is transferred back, the coroutine resumes the preserved state and continues to run from the point of the transfer. At its end or at the encounter of a return statement a coroutine dies, i.e., passes control back to its caller or some other coroutine as defined by the semantics of the coroutine system.

In contrast to other programming language concepts (e.g., continuations) coroutines are not strictly defined. Coroutines may not even be recognizable as such, and depending on their semantics and expressiveness they are called generators, coexpressions, fibers, iterators, green threads, greenlets, tasklets, or cooperative threads.

Coroutines that can only yield from within their main method are called *stackless*. They can be implemented by compile-time transformations but are only useful for a limited range of applications. If coroutines can yield from subsequently called methods, they are called *stackful*.

Only coroutines that are not tied to a specific language construct (like for-each loops) are considered to be *first-class* coroutines.

Asymmetric coroutines are bound to a specific caller and can only transfer control back to this caller. *Symmetric coroutines* can transfer control to any other coroutine [10]. The target coroutine can either be specified explicitly (*yieldTo*) or implicitly (*yield*). Using *yield*, control is passed to a scheduler that selects some other ready coroutine to run.

An implementation of coroutines is a useful feature for a virtual machine for a number of reasons:

- Coroutines are present in many new programming languages such as Ruby [4] and Go [5]. Language implementations that rely on an underlying virtual machine, such as a JVM, need to emulate coroutines if they are not available. The most common ways to emulate coroutines are using synchronized threads and compile-time transformations, both of which have significant drawbacks.
- Coroutines are a natural control abstraction for many problems. For example, producer/consumer problems can often be implemented elegantly with coroutines.
- In contrast to heap-allocated state objects, coroutines allow the compiler to fully optimize local variables, e.g., to put them into processor registers. They can thus benefit from the fact that their state is not accessible from the outside, while heap-allocated state objects lead to code that is compiled as if such access was possible.
- Coroutines provide an easy way to inverse a recursive algorithm into an iterative one. For example, a callback-based SAX parser [11] can be converted into a coroutine that returns one XML element every time it transfers back to its caller.
- Every coroutine is constrained to a single thread. Within their thread coroutines do not need to be synchronized because the points of transfer can be chosen such that no race conditions occur.

3. Implementation

Coroutines have been incorporated into many different languages. Depending on the architecture of the underlying runtime system different implementation techniques have been chosen.

The most common techniques for languages that use the CPU-supported stack management are:

Separate coroutine stacks. A separate stack area is allocated for each coroutine. This was very simple in older operating systems (like DOS), but the introduction of exception management, the need for correctly handling stack overflows, and other stack-specific algorithms have made allocating memory that can function as stack space more difficult.

Copying parts of the stack. The stack data of every coroutine is copied to and from the stack as needed. While this approach is simple to implement, it also has two main disadvantages:

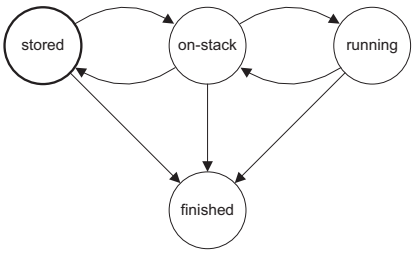

Figure 2. Coroutine states and transitions

Firstly, the coroutine transfer is much more expensive (and its costs depend on the current number of stack frames). Secondly, in garbage-collected environments the data that has been copied from the stack can contain root pointers that need to be visited during garbage collection, which means that the garbage collector needs to be aware of them.

In programming languages that represent the stack as a linked list of stack frames it is trivial to implement coroutines by storing the pointer to the topmost frame of the coroutine. Such systems need special garbage collection algorithms to free obsolete frames.

3.1 Additional Stacks

In order to provide optimal performance, our coroutine implementation creates separate stacks for coroutines. On an ideal CPU (ignoring caching effects) the switching between coroutines that lie on separate stacks is always a cheap constant-time operation, because only few CPU registers need to be saved and restored.

The required memory is requested from the operating system's memory management system. A continuous area of memory is allocated. The end of the stack is protected by guard pages to allow the runtime system to handle stack overflow errors. These guard pages are configured in such a way that any access to them will lead to an operating system exception. The JVM handles these exceptions and in turn throws a `StackOverflowError` at the Java level.

3.2 Stored Coroutines

Separate stacks take up significant amounts of memory and address space. A stack occupies at least 32-100 kByte of memory (see Section 5.1), due to guard pages and the operating system's allocation granularity. On 32-bit systems the address space is exhausted after roughly 25,000 stacks have been allocated. The exact number depends on the operating system's memory layout and the JVM configuration.

Even on 64-bit systems, where the address space usage is less of an issue, the memory overhead still prevents the allocation of a large number of coroutine stacks.

Only a small amount of the stack is actually used by a suspended coroutine, typically 1 to 2 kByte. In our system multiple coroutines can share a stack. A coroutine is nevertheless permanently associated with a specific stack: while it is possible to relocate Java frames (albeit expensive), it is not possible to do so for native frames. Note that relocation of frames is only a problem if the relocated frame is executed at its new position; it is not a problem to temporarily move a frame to some other location while the corresponding coroutine is suspended.

When a coroutine is created the system automatically selects a stack that the coroutine will be associated with by looking for the stack with the smallest number of associated coroutines. The programmer can also select a coroutine's stack explicitly by specifying some other coroutine that will share its stack with the first one.

3.3 Coroutine State

As coroutines are copied to and from their stacks they need to keep track of their state. Figure 2 shows the different states that a coroutine can have and the transitions that are possible.

stored In this state the stack frames of the coroutine are stored in a data object. Every new coroutine is created in this state, with a small stub data object containing a fabricated frame that starts coroutine execution. This frame is copied to the stack the first time the coroutine is called.

on-stack A coroutine whose stack frames are on a stack, but which is not currently running is in the "on-stack" state. Notice that in order to change into the "running" state the coroutine first has to be "on-stack".

running The currently running coroutine of a thread is in the "running" state. This implies that the coroutine's stack frames are located on its stack.

finished Coroutines that finish execution change into this state. Normally, a coroutine reaches the "finished" state from the "running" state, but there are circumstances when coroutines can change to "finished" directly from other states.

3.4 Data Structures

The VM uses a set of data structures to keep track of all existing coroutines (see Figure 3):

Thread is the preexisting data structure for threads. It is enhanced with a list of coroutines (`coroutine_desc_list`) and a list of coroutine stacks (`coroutine_stack_list`). Both are implemented as circular doubly-linked lists. Note that the thread's body is considered to be a coroutine itself. Therefore, the first elements of the two lists are the thread's original stack and coroutine, which live as long as the thread is alive.

CoroutineDesc holds VM-internal information that is coroutine-local (e.g., for resource and handle allocation). It also specifies which stack this coroutine belongs to (`stack`) and holds a pointer to the coroutine's off-stack data storage (`data`). Additionally it contains a `state` field which always holds the coroutine's current state.

CoroutineStack holds information about a stack area. This includes the memory address and size of the reserved space. The `current` pointer points to the coroutine that is currently active on this stack. There is only one `CoroutineStack` structure for each stack area, which can be referenced by more than one `CoroutineDesc` structure.

CoroutineData is used to store the stack contents of a coroutine while the stack is occupied by another coroutine.

Like every native thread has a mirror Java class in the form of `java.lang.Thread`, there is a Java class `Coroutine` for coroutines (see Section 4). It is needed in order to allow Java code to interact with the coroutine system. This mirror class contains a pointer to the native `CoroutineDesc` structure.

3.5 Creating Coroutines

Creating a new coroutine involves the following steps:

- Determine if a new stack should be created or not. This can either be specified explicitly (by telling the new coroutine to share the stack with an existing one), or determined automatically. If the stacks are managed automatically, the system creates a new stack for each coroutine up to a specified number and then select coroutines that share stacks using a simple hash function.

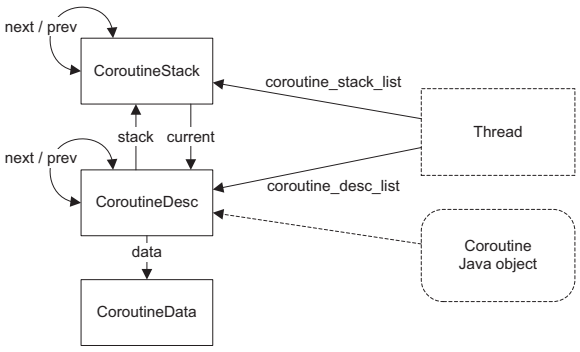

Figure 3. Native coroutine data structures

- In case a new stack is needed the system creates and initializes the `CoroutineStack` structure and allocates a block of memory from the operating system. It also secures the end of the stack with guard pages and makes further calls to register the memory block as belonging to the current thread.
- Next the `CoroutineData` structure is created and filled with a small dummy frame whose return address points to a function that initializes and starts the coroutine.
- Finally the `CoroutineDesc` structure is created and associated with the `Coroutine` Java object.

After all the native structures have been initialized the system connects the Java `Coroutine` object with the `CoroutineDesc` structure. Finally it fills some auxiliary fields and returns the newly create coroutine.

3.6 Context Switch

Switching from one coroutine to another is the most basic operation in every coroutine system. It heavily influences the overall performance and should therefore be implemented with as few instructions as possible.

An example of how the native structures are modified during a context switch is shown in Figure 4. In this case there are two coroutine stacks, one of which is shared by two coroutines. The figure shows the switch from the first to the second coroutine and from the second to the third coroutine.

Figure 5 shows the details of the context switch operation of our algorithm. It is divided into two parts: "prepare switch" and "switch". If the first one succeeds it is guaranteed that the second one succeeds as well, without any errors, exceptions, or safepoints (e.g., garbage collection runs). This separated design, which involves only a negligible overhead, was chosen in order to be able to cleanly handle out-of-memory errors while enlarging the `CoroutineData` structure.

We decided to implement the core functionality in a mixture of Java and assembly code. The assembly code is generated at JVM startup time to fit the current JVM configuration and can be called from compiled and interpreted Java code with no overhead (apart from the call/ret instructions). Depending on the operating system and the CPU, our system is comprised of approximately 100-120 assembly instructions, of which 35-50 form the fast path that only switches from one stack to the other without having to copy any stack contents (bold line in Figure 5).

The system first checks whether the target coroutine belongs to the current thread. It is not allowed to switch to another thread's coroutine, and an exception is thrown in this case.

Then it is checked if the target coroutine is in the "on-stack" state, in which case no additional preparations are necessary. If the target coroutine is not "on-stack" and its stack is currently

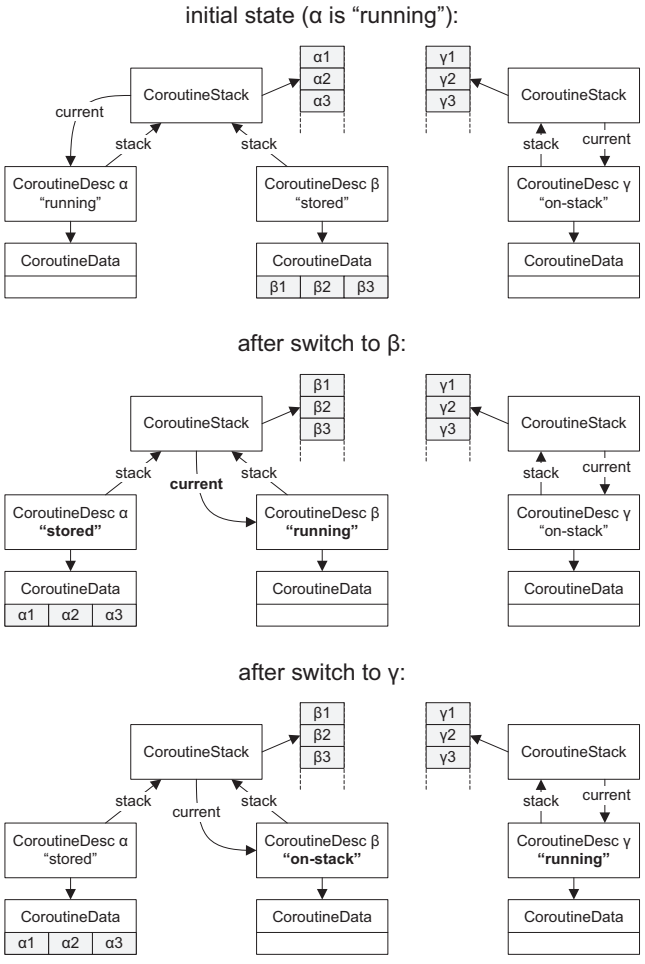

Figure 4. Data structures before and after context switch

occupied by another coroutine, the system checks if this coroutine's `CoroutineData` structure is large enough to accommodate the stack contents. The growing of the `CoroutineData` structure (if needed) is performed by C++ code which throws an exception if the system runs out of heap memory.

After the "prepare switch" part is finished it is guaranteed that the context switch will succeed, so the coroutine system now updates its Java data structures so that the target coroutine becomes the current coroutine.

Now the actual switch is performed. First, the old CPU context (i.e., program counter, stack pointer, frame pointer, etc.) is stored into the old coroutine. Then the system determines if it is necessary to rescue the old stack contents and/or restore the target's stack contents. The target coroutine might already be "on-stack", in which case no rescuing or restoring is necessary. If the target stack is not currently in use (i.e., newly created or the coroutine that last occupied it finished), then only the new contents need to be restored. If the target stack is currently in use both rescuing and restoring are necessary.

When the rescue and restore operations are finished the target CPU context is restored, which, from a VM perspective, represents the actual switch operation.

If the old coroutine has reached its end then its data structures are freed.

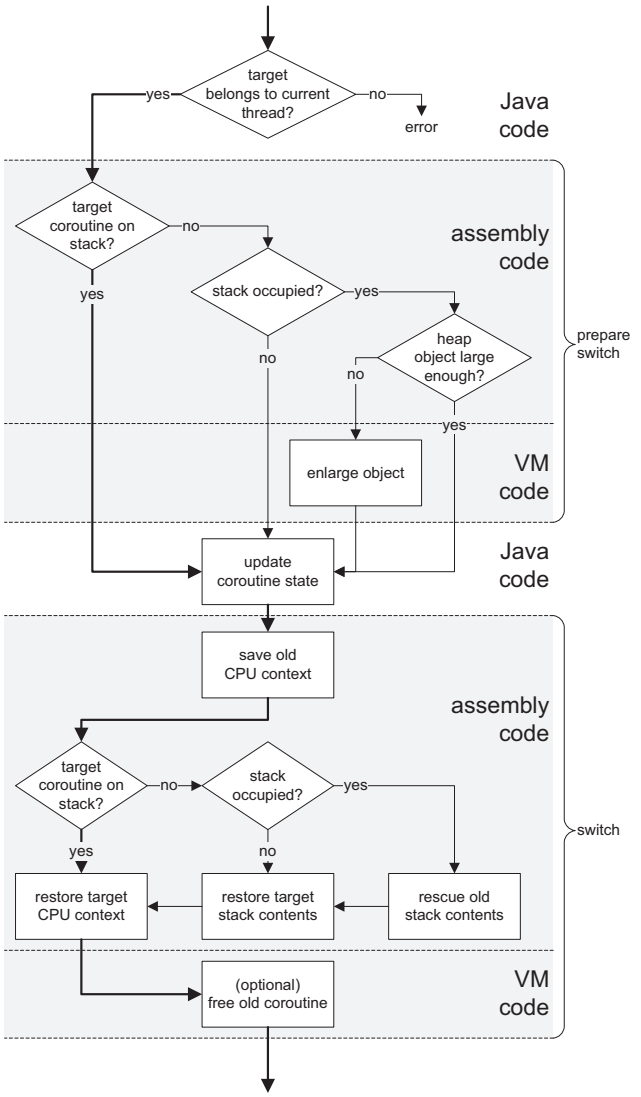

Figure 5. Coroutine context switch operation

3.7 Coroutines and Garbage Collection

The JVM needs to be able to locate the stack frames of all coroutines at all times because it needs to visit stack frames for a number of reasons, including:

- When compiling methods the just-in-time compilers create code that is valid only for the currently loaded set of classes. For example, if the compilers discover that there is only one possible target for a polymorphic call site they output a monomorphic call instead. It is also possible to inline the method in question. The compilers register the assumptions they took while compiling code with the system. If during the loading of new classes such assumptions are violated all activations of the affected methods need to be located and modified in such a way that they will resume execution in the interpreter. This process is known as *deoptimization* [7].
- During garbage collection the JVM needs to find all root pointers to the set of live objects. The stack frames' local variables and expression stack values can contain root pointers. Also, the methods that the stack frames belong to are objects that need to be kept alive.

The thread's `coroutine_desc_list` is used to iterate over all live coroutines when needed. If a coroutine is on-stack then its stack is walked in exactly the same way as ordinary thread stacks are. If a coroutine's frames are stored in its `CoroutineData`, a special-purpose stack walking method is used. This method has to take into account the displacement between the `CoroutineData` and the stack that the stack frames lie on when they are executed, i.e., it modifies each pointer that points to the stack to point to the correct position within the `CoroutineData` structure.

```
public class Coroutine {

  public Coroutine();
  public Coroutine(Runnable r);
  public Coroutine(long stackSize);
  public Coroutine(Runnable r, long stackSize);

  public static void yield();
  public static void yieldTo(Coroutine target);

  public boolean isAlive();
  protected void run();
}
```

Figure 6. Coroutine Java API: `Coroutine` class

```
public class SampleCoroutine extends Coroutine {

  public void run() {
    System.out.println("Coroutine running 1");
    yield();
    System.out.println("Coroutine running 2");
  }

  public static void main(String[] args) {
    new SampleCoroutine();
    System.out.println("start");
    yield();
    System.out.println("middle");
    yield();
    System.out.println("end");
  }
}
```

Figure 7. Example for the usage of `Coroutine`

4. Usage: Java API

A survey of 12 different programming languages with coroutine implementations showed that there is no common naming scheme for coroutines, neither in the way the corresponding language features are called, nor in the names of the methods that are used to control them. We created a Java API for our implementation that tries to be consistent with the majority of programming languages and that fits smoothly into the Java world. The API always works on the coroutines of the current thread in order to avoid having to lock data structures.

There are two main classes: `Coroutine` for symmetric coroutines, and `AsymCoroutine` for asymmetric coroutines. They can both be supplied with a stack size, which the runtime tries to provide, but which cannot be guaranteed. By specifying a stack size the user can deal with extreme cases such as coroutines that call deeply recursive methods and therefore need a large stack, or large numbers of coroutines that hardly call any other methods and for

```
public abstract class AsymCoroutine<InT, OutT>
                      implements Iterable<OutT> {

  public AsymCoroutine();
  public AsymCoroutine(long stackSize);

  public OutT call(InT input);
  public InT ret(OutT output);

  public boolean isAlive();
  protected abstract OutT run(InT input);

  public Iterator<OutT> iterator();
}
```

Figure 8. Coroutine Java API: `AsymCoroutine` class

which a large stack would be a waste of memory. In most cases, however, the automatically chosen stack size will be adequate.

4.1 Coroutine

The interface for symmetric coroutines is shown in Figure 6. It is similar to `Thread` in that it can either be supplied with a `Runnable` at construction or subclassed (overriding the `run()` method).

Each thread maintains a circular doubly-linked list of coroutines that serves as an ordered container for all live coroutines. The coroutine system inserts a newly created `Coroutine` object after the current one, which means that it is the next coroutine to be scheduled. There are two ways to switch to another coroutine:

`yield()` transfers control to the next coroutine in the list, which automatically moves the current coroutine to the end of the list.

`yieldTo(Coroutine target)` transfers control to the specified coroutine. The target coroutine must be a coroutine created within the current thread, otherwise an exception is thrown. This method behaves as if the coroutines between the current and the target coroutine were silently skipped, i.e., it does not alter the order of the coroutines within the current thread.

Both these methods are static and always act on the current thread's list of coroutines. The instance method `isAlive()` returns true if the `Coroutine` in question has not yet reached its end.

Figure 7 shows an example of a simple coroutine that produces the following output:

```
start
Coroutine running 1
middle
Coroutine running 2
end
```

4.2 AsymCoroutine

For reasons of generality, we also provide an implementation of asymmetric coroutines, called `AsymCoroutine`, shown in Figure 8. Instances of this class can only be created by subclassing `AsymCoroutine` (overriding the `run()` method).

`AsymCoroutine` objects are not part of the ordinary `Coroutine`-scheduling, they are thus only executed when they are explicitly called. They also know their caller, which allows them to return to their caller with a `ret()` call.

`AsymCoroutine`s are prepared to take an input from their caller and to return an output. The types of these input and output parameters can be specified by generic type parameters. If the input and/or output parameters are not used, the respective type parameters should be set to `Void`. The input parameter given to the first `call()` will become the `input` parameter of the run method, and the output parameter returned by the `run()` method will become the last `call()`'s return value.

```
public class CoSAXParser
             extends AsymCoroutine<Void, String> {

  public String run(Void input) {
    SAXParser parser = ...
    parser.parse(new File("content.xml"),
      new DefaultHandler() {
        public void startElement(String name) {
          ret(output);
        }
      });
    return null;
  }

  public static void main(String[] args) {
    CoSAXParser parser = new CoSAXParser();

    while (parser.isAlive()) {
      String element = parser.call(null);
      System.out.println(element);
    }
  }
}
```

Figure 9. SAX parser inversion

```
  ...
  public static void main(String[] args) {
    CoSAXParser parser = new CoSAXParser();

    for (String element: parser) {
      System.out.println(element);
    }
  }
}
```

Figure 10. SAX parser inversion using enhanced for loops

The fact that `AsymCoroutine` implements `Iterable` allows such coroutines to be used in enhanced for loops. In this case, the input parameter is always `null` as there is no way to supply the iterator with an input.

The `AsymCoroutine` interface looks as follows:

`OutT call(InT input)` transfers control to the coroutine that this method is called on. The calling coroutine can either be a `Coroutine` or `AsymCoroutine` instance, and it is recorded as the caller of the target coroutine. `call()` passes an input parameter of type `InT` to the called coroutine and returns the coroutine's output parameter, which is of type `OutT`. A coroutine can only be called if it is not currently in use. This means that a coroutine cannot directly or indirectly call itself. Invoking `call()` on a `AsymCoroutine` that is not alive leads to an exception.

`InT ret(OutT output)` suspends the current coroutine and returns to the caller. The output parameter of type `OutT` is passed to the calling coroutine, and the next time the current coroutine is called `ret()` will return the input parameter of type `InT`.

`boolean isAlive()` returns true if the `AsymCoroutine` in question has not yet reached its end.

It is important to note that trying to `yield()` to another `Coroutine` generates an exception if the current coroutine is an `AsymCoroutine`.

Figure 9 shows an example `AsymCoroutine` that inverts a SAX parser [11] to return one XML element at a time. Figure 10 contains

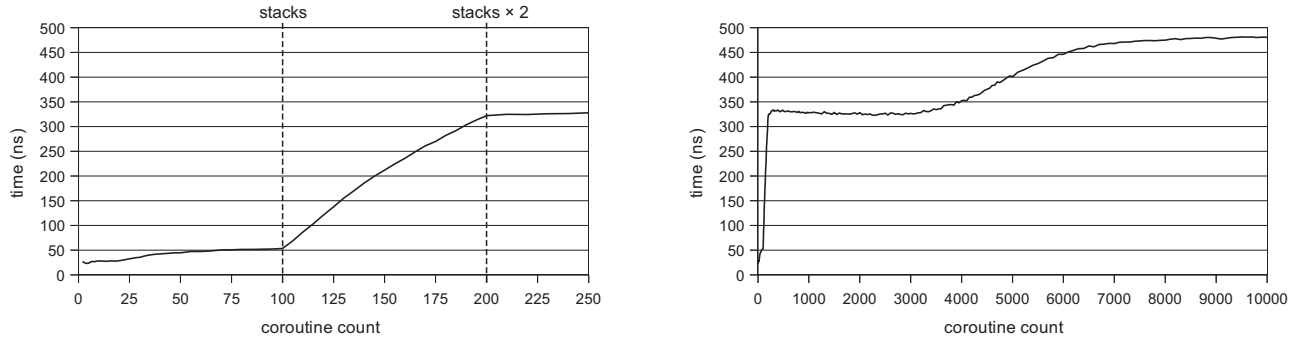

Figure 11. average time for one "switch" operation, relative to the active coroutine count

an alternative version of the main method that uses an enhanced for loop.

The system also includes a subclass of `AsymCoroutine` with Void input and output parameter types which provides `Thread`-like construction using a `Runnable` and parameterless `call()` and `ret()` methods.

5. Evaluation

All tests and benchmarks presented in this section were performed on a computer with a quad-core Intel i5 750 CPU at 2.67 GHz and 8 GByte main memory running Windows 7.

5.1 Memory Consumption

Coroutines consume different amounts of memory depending on their number of active stack frames. Unless a coroutine has a deeply nested chain of method activations at the time it yields control, it uses no more than 2 kByte of space (roughly 1 kByte used by the VM plus at least 32 bytes per Java frame). The size of the various control structures is negligible compared to the size of the stack contents. We managed to run 1,000,000 concurrent coroutines on a 32-bit machine, at which time the address space was exhausted.

The size of the coroutine stack depends on the reservation and allocation granularity of the operating system and the requested stack size. The stack size also has to account for a number of so-called guard and shadow pages that are used to detect stack overflows. These additional pages take up between 24 and 92 kByte, depending on the operating system. This leads to a minimum stack size of 32 kByte on Linux, 64 kByte on Windows and 100 kByte on Solaris, which allows the allocation of roughly 60,000, 30,000, or 20,000 stacks on a 32-bit machine.

5.2 Execution Time

The execution time of the various coroutine operations depends on factors such as the cache layout of the CPU, the operating system and others. Nevertheless an approximation can be given.

On our system, creating a new coroutine takes 1.5 μs if a new stack needs to be created, and 0.3 μs if not. Creating a new thread takes roughly 2.5 μs. The first call to the coroutine has to set up a number of data structures and takes 3 μs, whereas starting a thread takes 60 μs.

Switching between coroutines is the most important operation and takes 20 ns in the best case. This is less than twice the time it takes to make a polymorphic method call.

Figure 11 shows the average time for one "switch" operation in relation to the total amount of active coroutines. The benchmark we used to create these graphs creates the given number of coroutines, switches to all coroutines and computes the average time per switch. The left graph is a magnification of the right one and shows the behavior for small numbers of coroutines. The coroutine system was configured to allow a maximum of 100 distinct coroutine stacks per thread.

The graphs illustrate the factors that influence the performance of the switch operation:

- Up to 25 coroutines the switch time is dominated by the raw execution time of the assembly and Java code.
- From 25 to 75 coroutines the size of the working set outgrows the first-level CPU cache.
- Between the number of stacks (100) and two times the number of stacks (200) the execution time degrades towards the time for the copy operation.
- At around 4,000 coroutines the size of the working set starts to outgrow the second-level CPU cache.

The time per switch stays at 480 ns even for very large numbers of coroutines (1,000,000). These times depend heavily on the machine the benchmarks are executed on, but they nevertheless show that there are no abrupt performance degradations

5.3 JRuby fibers performance

In order to determine the effect of JVM-level coroutine support on language implementations that run on top of the JVM, we evaluate the performance of different Ruby [4] environments. Ruby is a general purpose object-oriented programming language that draws concepts from a number of programming languages including Perl, Smalltalk, Eiffel, Ada and Lisp. The high flexibility and programming frameworks such as Ruby on Rails [19] have made the language very popular, consistently ranking in the top 15 of the TIOBE Programming Community Index [20] in recent years. The latest version of the Ruby language (1.9) implements a feature called *fibers*, which provides a simple interface to symmetric and asymmetric coroutines.

The Ruby community [18] maintains a C implementation that is called *Matz's Ruby Interpreter* (MRI) after the creator of the Ruby language (Yukihiro Matsumoto). It is the most widely-used Ruby implementation and as such serves as a baseline for our benchmarks. MRI implements fibers with a simple copying approach.

JRuby [17] implements the Ruby language on top of a JVM and is the second-most widely-used Ruby implementation. It implements fibers by creating a thread for each fiber and synchronizing the threads in such a way that the threads behave as if they were fibers. We modified JRuby to use our JVM coroutines instead of threads to implement fibers.

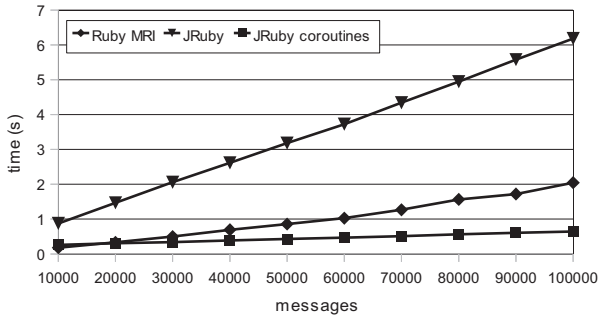

Figure 12. Ruby benchmark: 5 fiber chain

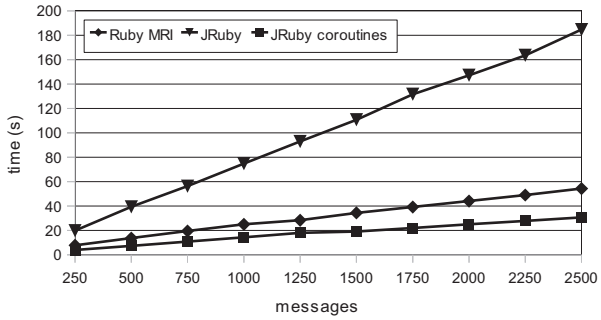

Figure 13. Ruby benchmark: 5000 fiber chain

In order to assess the performance of these implementations we used a simple benchmark that passes messages through a chain of fibers. Tests with different numbers of fibers and messages showed that for all test sizes a large speedup is attained by the use of coroutines.

Figures 12 and 13 show a comparison of the performance of MRI, JRuby without modifications, and JRuby with coroutine support. Figure 12 shows the runtime of a benchmark that passes different numbers of messages through a chain of 5 fibers, and Figure 13 shows the same benchmark for a chain of 5000 fibers.

The graphs show that our combined approach works especially well when each coroutine has its own stack (which is the case with 5 fibers) and still works reasonably well when it needs to resort to copying (which happens with 5000 fibers).

JRuby with coroutines is roughly 2 times faster than MRI, which in turn is about 3.5 times faster than JRuby without coroutines. On average, JRuby with coroutines is 7 times faster than JRuby without coroutines.

6. Related Work

Most of the publications on coroutines deal with the operational semantics and the connection to other research areas, while the actual implementation details are often omitted.

Weatherly et al. [21] implemented coroutines in Java for simulation tasks. Their implementation copies data to and from the stack for each context switch. The focus of their work lies on the connection to the simulation infrastructure, and they hardly touch the intricacies of the coroutine semantics in the Java environment.

Ierusalimschy et al. [6] describe how the Lua virtual machine handles coroutines. Their implementation uses a separate stack for each coroutine. The Lua virtual machine is interpreter-only, and the interpreter functions *stackless*. This means that the actual stack data can be stored at an arbitrary memory position, completely circumventing the problem of dealing with native stacks.

Bobrow and Wegbreit [1] show an implementation of coroutines that stores a spaghetti stack containing stack frames of multiple coroutines into a single stack segment. This is achieved by jumping back and forth on the stack and testing if enough free space is available for a new stack frame each time a subroutine is called. While this approach can be efficient, it suffers from a number of drawbacks. It also violates the assumption made by modern operating systems that there should always be a certain amount of available stack space beneath the current stack frame.

Pauli and Soffa [13] improve upon Bobrow and Wegbreit. They introduce a dynamic and moving reentry point for coroutines, instead of the static one used by the original algorithm. They also show how to apply the algorithm to statically scoped languages and conclude with a detailed analysis of different variations of the algorithm. However, all models described by Pauli and Soffa still incur a significant overhead for each method call.

Mateu [9] makes the case that a limited version of continuations can be implemented via coroutines. He presents an implementation that heap-allocates frames and uses a generational garbage collector to free obsolete frames. The performance measurements show that it is difficult to find the optimal size of this frame heap because the performance is dominated by the tradeoff between CPU cache efficiency (small heap) and fewer collections (large heap).

Carroll [2] shows an example of a real-world problem that would benefit from a coroutine implementation in Java. Two consecutive parsers are used to parse RDF (which is based on XML). Using two ordinary parsers is only possible if they are executed in different threads. Carroll shows that a significant performance increase is possible if the two parsers are combined into a single thread. In this case this is achieved via inverting the RDF parser, which is undesirable from a design perspective. The need for manual inversion could be avoided if real coroutines were available.

7. Future Work

An additional state “serialized” (added to the states shown in Figure 2) would be an interesting extension to the approach presented in this paper. Coroutines in this state could move to other threads and possibly also to other JVMs. If such serialized coroutines were cloneable the system would be functionally equivalent to a system supporting first-class continuations [14], and as such be applicable to problems that are only solvable by continuations.

8. Conclusions

Coroutines are an elegant solution for many problems and as such have reappeared in a number of modern programming languages.

In this paper we presented a coroutine implementation approach that allocates stacks up to a certain limit and then starts copying data to and from these stacks. We showed the implementation of this approach for the widely used Java HotSpot^{TM} VM.

The main contribution of our approach is to show that the two most widely-used coroutine implementation strategies can be combined, resulting in an implementation that performs well over a wide range of different workloads. We also introduced a Java API for coroutines.

We gave estimates for the memory usage of our coroutines and the run time of the most important operations on them. We also analyzed the performance of our approach in the context of language implementations, which showed that for languages that provide coroutine-like features JVM-based implementations can benefit significantly from our coroutines.

Acknowledgments

This work was performed in a research cooperation with Oracle (formerly Sun Microsystems). We would like to thank John Rose and Dave Cox for their continuing support.

References

[1] D. G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16(10):591–603, 1973.

[2] J. J. Carroll. Coparsing of RDF & XML. Technical Report HPL-2001-292, Hewlett Packard Laboratories, Nov 2001. URL http://www.hpl.hp.com/techreports/2001/HPL-2001-292.pdf.

[3] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963. doi: 10.1145/366663.366704.

[4] D. Flanagan and Y. Matsumoto. *The Ruby programming language*. O'Reilly, 2008. ISBN 9780596516178.

[5] Google. Go programming language, 2009. http://golang.org/.

[6] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005. doi: 10.3217/jucs-011-07-1159.

[7] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1): Article 7, 2008. doi: 10.1145/1369396.1370017.

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

[9] L. Mateu. An efficient implementation for coroutines. In *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 230–247. Springer, 1992.

[10] A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):6:1–6:31, Feb. 2009. doi: 10.1145/1462166.1462167.

[11] P. Murray-Rust, D. Megginson, T. Bray, et al. *Simple API for XML*, 2004. http://www.saxproject.org/.

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

[13] W. Pauli and M. L. Soffa. Coroutine behaviour and implementation. *Software—Practice and Experience*, 10(3):189–204, Mar. 1980.

[14] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose. Lazy continuations for Java virtual machines. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*, pages 143–152. ACM, 2009. doi: 10.1145/1596655.1596679.

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

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

[17] The JRuby Community. *JRuby*, 2010. http://jruby.org/.

[18] The Ruby Community. *Ruby Community Website*, 2010. http://www.ruby-lang.org/.

[19] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006. ISBN 0977616630.

[20] TIOBE Software BV. Tiobe programming community index, April 2010. http://www.tiobe.com/tpci.htm.

[21] R. M. Weatherly and E. H. Page. Efficient process interaction simulation in Java: Implementing co-routines within a single Java thread. *Winter Simulation Conference*, 2:1437–1443, 2004.