JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Serializable Coroutines for the HotSpot™ Java Virtual Machine

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Software Engineering

Eingereicht von:
Lukas Stadler, BSc.

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Linz, Februar 2011

Abstract

Coroutines are non-preemptive light-weight 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 trade-off 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. This thesis presents 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.

The implementation of two advanced concepts, namely thread-to-thread migration and serialization of coroutines, is also explained along with the API that is used to control them.

For evaluation purposes, coroutines were used to implement JRuby fibers, which lead to a significant speedup of certain JRuby programs. This thesis concludes with general benchmarks that show the performance of the approach and outline its run-time and memory characteristics.

The work presented in this thesis was performed in a research cooperation with, and sponsored by, Oracle (formerly Sun Microsystems).

Kurzfassung

Coroutinen sind nicht-präemptive, leichtgewichtige Prozesse. Im Gegensatz zu Threads müssen sie nicht synchronisiert werden, da die Ausführungsreihenfolge explizit und deterministisch festgelegt wird. Dadurch ermöglichen sie eine elegante und effiziente Implementierung vieler Algorithmen.

Allerdings stellen viele verbreitete Laufzeitumgebungen und Programmiersprachen keine Coroutinen zur Verfügung, und existierende Coroutinen-Implementierungen bieten oft nur unzureichende Kompromisse zwischen Laufzeit- und Speichereffizienz.

Dadurch, dass es mehr und mehr Programmiersprachen-Implementierungen für Java VMs gibt, von denen viele Coroutinen-ähnliche Funktionen anbieten, ergibt sich eine vermehrte Nachfrage nach einer Coroutinen-Implementierung. Diese Arbeit präsentiert eine Implementierung, die für einen weiten Bereich verschiedener Aufgaben performant ist, wobei Programme, die Coroutinen nicht benutzen, nicht verlangsamt werden.

Auch zwei weitergehende Konzepte, das Migrieren von Coroutinen zwischen Threads und das Serialisieren von Coroutinen, werden zusammen mit dem benötigten API beschrieben.

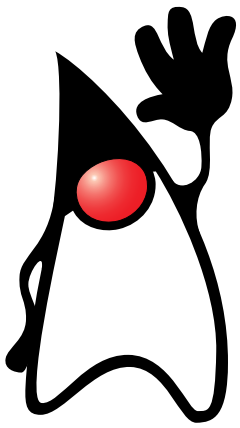
Zur Evaluierung wurden JRuby fibers mittels Coroutinen implementiert, was zu einer starken Steigerung des Programmdurchsatzes führte. Die Arbeit schließt mit einer Evaluierung der Laufzeit- und Speichercharakteristika der Implementierung.

Die hier vorgestellte Forschungsarbeit wurde im Rahmen einer Kooperation mit und mit Unterstützung von Oracle (vormals Sun Microsystems) erstellt.

Contents

Chapter 1

Motivation



This chapter shows why a coroutine implementation in a Java Virtual Machine is useful and who will benefit from it. It also introduces two advanced concepts, namely thread-to-thread migration and serialization/deserialization.

Coroutines are a programming language concept that allows for explicit, cooperative and stateful switching between subroutines. They are a powerful and versatile concept that provides a natural control abstraction for many problems. For example, producer/consumer problems can often be implemented elegantly with coroutines [7].

Within a thread, coroutines do not need to be synchronized because the points of control transfer can be chosen such that no race condition can occur.

Coroutines also provide an easy way to inverse recursive algorithms into iterative ones. For an example of this see Figure 3.6 on page 24.

Coroutines can be simulated by putting the state of a method execution into an object, similar to how closures [16] are usually implemented. In this scheme local variables are stored in heap objects while the task is suspended, so that they can be restored later

on. Only coroutines allow the compiler to fully optimize local variables, e.g., put them into processor registers or stack slots. They can thus benefit from the fact that their state is not accessible from the outside, while heap-allocated objects lead to code that is compiled as if such access was possible.

Because of this versatility many new programming languages, such as Ruby [11], Go [13] and Lua [15], incorporate coroutines as essential language or runtime environment features. Language implementations that rely on an underlying virtual machine (VM), such as a Java Virtual Machine (JVM), need to emulate coroutines if they are not available. The most common ways to do so are using synchronized threads and compile-time transformations, both of which have significant drawbacks.

Thus it can be expected that a native coroutine implementation for a production-quality JVM, such as the HotSpot™ Virtual Machine, will provide significant value to language implementation, framework and application developers.

Simple coroutines, which are constrained to one thread, are useful for many use cases, but sometimes even more functionality is required:

Thread-to-Thread Migration
: For server applications that use coroutines to manage user sessions it might be beneficial to resume a coroutine in a different thread than the one in which it was created.

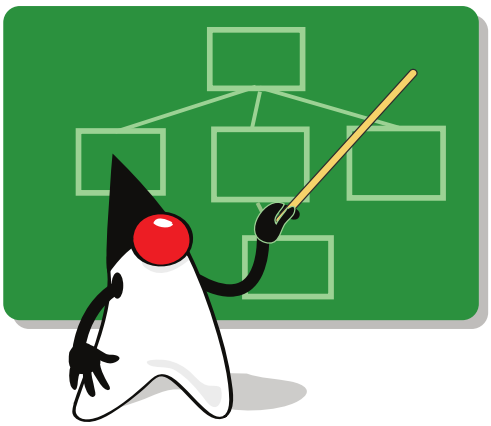
Serialization/Deserialization
: In other situations it might even be necessary to move a coroutine into permanent storage, like a database. This is useful for use cases like long-lasting transactions, checkpointing or agents that move between VMs.

A coroutine system that includes these concepts provides enough functionality for a large number of use cases.

The rest of this thesis is organized as follows: Chapter 2 provides an overview of the state of the art concerning Java and coroutines. Then, Chapter 3 puts these two pieces together and explains how coroutines fit into the Java world, including the necessary APIs. Chapter 4, which is the main chapter of this thesis, shows the actual implementation details of the basic coroutine implementation, the thread-to-thread migration and the serialization and deserialization. In order to illustrate how coroutines work in practice Chapter 5 presents some common use cases and usage patterns. Chapter 6 contains the evaluation of the coroutine implementation presented in this thesis, followed by an outlook to related work in Chapter 7. Chapter 8 provides proposals for further work, followed by conclusions in Chapter 9.

Chapter 2

State of the Art



This chapter provides an overview of Java, Java Virtual Machines and coroutines. A brief introduction to the Java programming language is followed by a description of the HotSpot™ Java Virtual Machine. Then this chapter introduces the basic concepts of coroutines, along with a short history of coroutines. It ends with an overview of common coroutine implementation techniques.

2.1 Java

Java[4] is a general-purpose object-oriented programming language developed by Sun Microsystems[1]. After its introduction in 1995 it quickly became one of the most successful and most widely-used programming languages, consistently ranking at the top of programming language rankings like the TIOBE programming language index [34]. Figure 2.1 shows a simple "Hello World!" implemented in Java.

[1] now a subsidiary of Oracle Corporation

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Figure 2.1: "Hello World!" in Java

While it was originally designed to run on interactive TV appliances, Java's success is tightly coupled to the success of the World Wide Web[2]. The ability to transfer code over a network and to execute it in a controlled environment, without letting it compromise the surrounding system (also known as *sandboxing*), played a crucial role in the adoption of Java *applets* as interactive elements in HTML pages. New technologies such as powerful JavaScript virtual machines diminished the importance of Java applets, but the large number of Java installations facilitated by the applet technology jump started the success of Java in many other areas.

Java gained widespread adoption in business applications because of a number of security-related features. Java pointers (called *references*) do not support pointer arithmetics, which is a major source of errors in lower-level languages. Java also does not allow explicit object deallocation, it instead provides a *garbage collector* that automatically frees unreachable objects. This avoids a large portion of common memory leaks and security problems. The memory model is sound enough to allow for *exact garbage collection*, which is a requirement for many high-performance garbage collection algorithms. Java is also largely platform independent, although in practice a programmer still needs to be careful to achieve full platform independence.

For programmers, Java is easier to learn and to maintain than many other languages. Java's object model is simple and easy to understand, but powerful enough for most use cases. For example, multiple inheritance is limited to interfaces, which is much simpler than multiple inheritance for classes. The Java programming language is accompanied by a large runtime library, which includes support for many advanced tasks like XML-parsing, graphical user interfaces, etc. Java also has a concise, C++ - like syntax which provides familiarity for C and C++ programmers.

2.2 Java Virtual Machines

The Java Compiler (`javac`) compiles Java source code, contained in `.java`-files into platform- and CPU-independent *bytecode* stored in one or more `.class`-files.

[2] Java-enabled TV sets were, at the time, unviable due to the cost of hardware capable of running Java. Ironically, recently introduced Blue-ray TV appliances use Java for their interactive components.

A *Java Virtual Machine* (JVM) then loads, verifies, and executes these bytecodes. It needs to adhere strictly to the semantics defined by the Java virtual machine specification [19]. It is important to note that all executable bytecodes are contained within methods, and in turn all methods are contained within classes. Although there can be more than one class within a single `.java`-file there is always exactly one class per `.class`-file. Multiple `.class`-files can be packaged into a compressed Java Archive (a so-called `.jar`-file), along with other resources needed by the application. Figure 2.2 provides an overview of these artifacts and the tools that generate them.

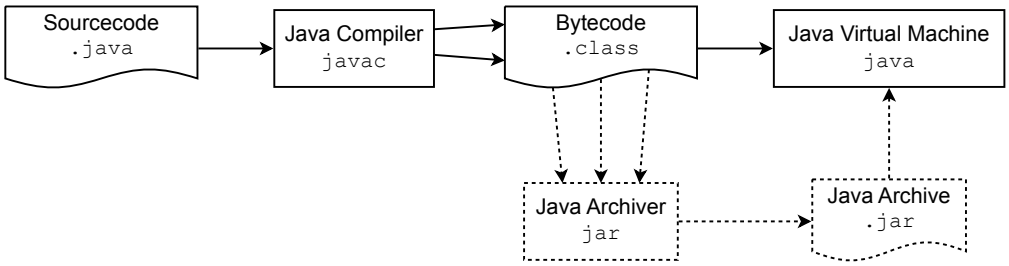


Figure 2.2: Overview of the artifacts generated by a Java system

After the classes are loaded and verified for soundness the bytecodes are executed. They can be interpreted or compiled just-in-time; most JVMs employ a sophisticated combination of both. The JVM is a stack machine that uses an expression stack instead of registers, but in practice all JVMs compile bytecode into register-based, platform-specific code.

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 return addresses of method calls in the form of a *bytecode index* (bci) or *program counter* (pc) for each activation frame, which is used to resume the caller at the correct position upon a return instruction.

In order to conform to the specification, JVMs need to implement automatic memory management (garbage collection). Garbage collection marks all objects that are reachable from a set of *root pointers* as live and discards all unreachable objects. The JVM has to consider a number of sources for root pointers, e.g., activation frames, references to Java objects from native code, and compiled code.

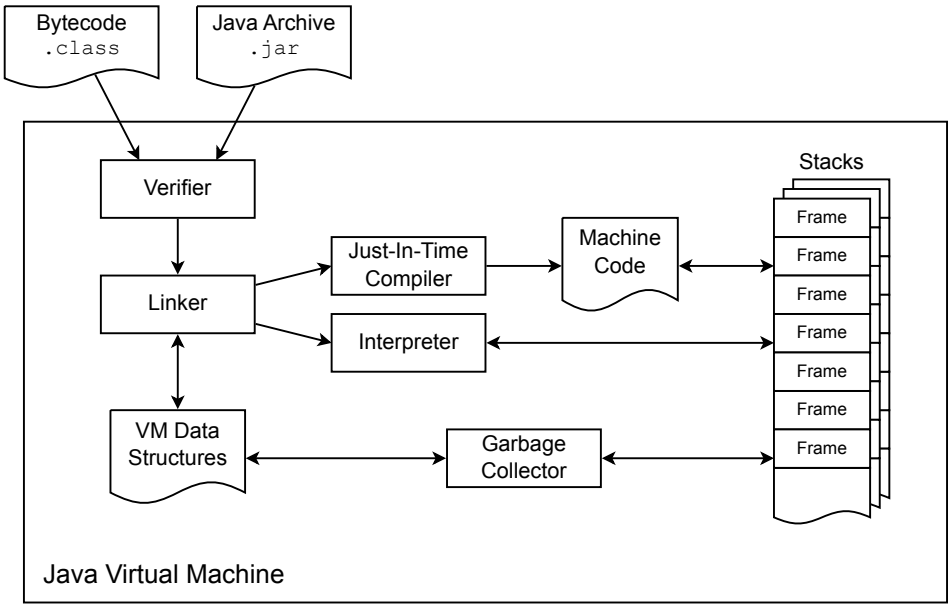


Figure 2.3: Java Virtual Machine overview

Figure 2.3 provides a simplified overview of the most important subsystems of a Java Virtual Machine. It is important to note that compiled code, interpreted code and the garbage collector interact with stack frames.

2.2.1 Java HotSpot™ VM

The Java HotSpot™ VM was introduced along with Java 1.3. It can either interpret Java bytecodes or use one of its two just-in-time (JIT) compilers, called *client compiler* [18] and *server compiler* [23], to compile bytecodes into optimized machine code. The name "hotspot" refers to the fact that it detects frequently executed methods, called *hot spots*, which are then targeted for compilation and further optimization.

The Java HotSpot™ VM also 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.

2.2.2 Thread and Stack Management

Java includes support for creating, starting and managing multiple execution threads within one application. Modern JVMs use the native threading facilities provided by the operating system, in order to achieve maximum performance.

The programmer can create threads by creating instances of the `java.lang.Thread` class, which contains all methods for managing threads. As soon as the thread is started the JVM will create the necessary native data structures, along with the actual operating system thread, so that the parallel execution can start.

Java threads are scheduled by the operating system, in a nondeterministic way. This means that there is no guarantee as to when a switch from one thread to the other occurs. There is also no way to know to which thread the system will switch, other than that only active, non-suspended threads will be considered.

Synchronization between threads is a feature that is important enough to be supported by a special keyword, **`synchronized`**. Every Java object includes a hidden lock field that is used for synchronization of threads. Every thread that wants to enter a synchronized code block must first obtain the lock of an object that is associated with that synchronized code. The lock is granted to only one thread at a time, which guarantees mutual exclusion of threads competing for critical resources.

Each thread has its own stack, which is managed by the runtime system. Normal Java code cannot interfere with stack management. The stacks are usually provided by the operating system for each thread automatically.

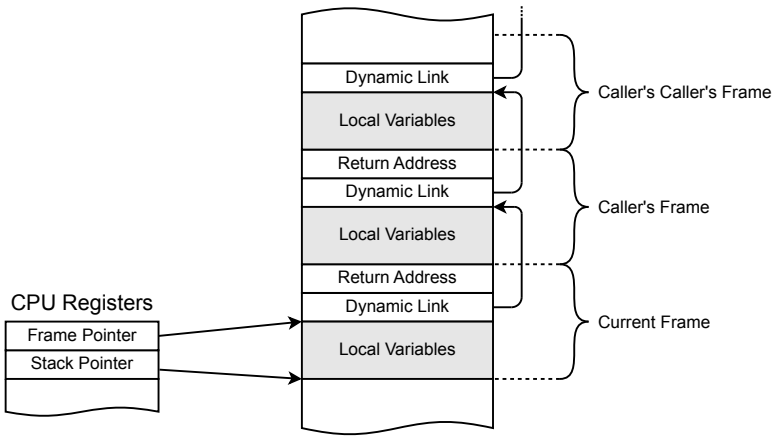


Figure 2.4: Layout of stack frames

In most systems the stack grows from higher towards lower addresses, Figure 2.4 shows the typical layout of such a stack: The CPU registers *stack pointer* and *frame pointer* define the lower and upper bounds of the current stack frame, and each stack frame contains a pointer (*dynamic link*) to the next one. It is thus possible to determine the location and size of all stack frames on the stack by following the dynamic links until the end of the stack is reached.

2.3 Coroutines

Coroutines [21] are a concept has been used since the early days of electronic data processing. The term "Coroutine" was coined in 1963 by Conway [7] 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. However, none of the top five languages of the TIOBE programming languages index [34] (Java, C, C++, PHP and Basic) supports coroutines without additional libraries.

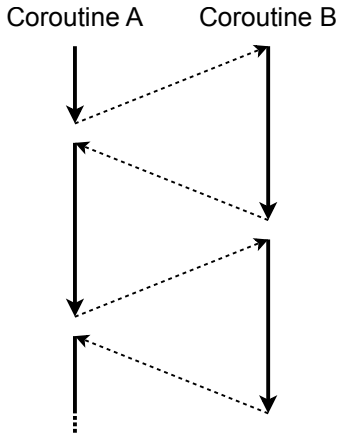


Figure 2.5: Interlocked execution of coroutines

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). Coroutines can transfer control back and forth, as shown in Figure 2.5.

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.

Coroutines have 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).

```
1 // Subroutine
2 class Container {
3   boolean first = false;
4   public String next() {
5     if (first) {
6       first = false;
7       return "Alpher";
8     } else {
9       first = true;
10      return "Bethe";
11      }
12   }
13 }
```

```
1 // Coroutine
2 class Container {
3   public String next() {
4     yield "Alpher";
5     return "Bethe";
6   }
7 }
```

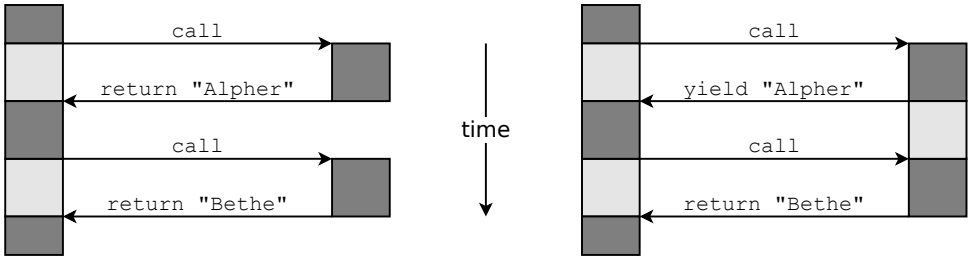


Figure 2.6: Comparison of subroutine (left) and coroutine execution (right)

Figure 2.6 shows a simple comparison between subroutine and coroutine execution[3]. The task that the methods in this example should achieve is to return *"Alpher"* the first time the method next() is called and *"Bethe"* the second time. While the subroutine needs to explicitly implement an ad hoc state machine, the coroutine can simple yield a value to the caller without destroying the method's execution frame. Thus, when the coroutine is called the second time it will resume immediately after the **yield** statement and return the second value. If a third value were to be returned by the coroutine version, it would simply need to be changed to **yield** *"Bethe"* and **return** *"Gamow"*.

2.3.1 Coroutine Characteristics

The fact that coroutines automatically preserve their execution context is sometimes called *automatic stack management* [2]. Any algorithm that uses coroutines can be re-implemented with state machines, which is considered to be *manual stack management*.

In contrast to other programming language concepts (e.g., continuations), the semantics of coroutines have no strict definition. Coroutines may not even be recognizable as such, and depending on their semantics and expressiveness they are called generators [14, 21], coexpressions [14], fibers [28], iterators [21], green threads [29], greenlets [26], or cooperative threads.

[3]Note that this example uses pseudo-code, the actual coroutine API is introduced in Section 3.2.

This section presents the most common attributes used to describe coroutine variants.

2.3.1.1 Stackful vs. Stackless

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. An example of this is C#'s implementation of iterator methods using the **`yield`** keyword, which causes the compiler to perform a transformation that turns the method into a state machine that implements the `IEnumberable` interface.

If coroutines can yield from subsequently called methods, they are called *stackful*.

2.3.1.2 First-class Coroutines

Only coroutines that are not tied to a specific language construct (like for-each loops) and that are programmer-accessible objects are considered to be *first-class* coroutines. Programming languages that limit the usage of coroutines to certain situations usually do so to allow the compiler to transform the coroutine code into ordinary sequential code.

2.3.1.3 Symmetric vs. Asymmetric Coroutines

Asymmetric coroutines are bound to a specific caller and can only transfer control back to this caller. The coroutine in Figure 2.6 is an asymmetric coroutine. The asymmetry lies in the fact that there are distinct and complementary operations for transferring control to and from a coroutine (**`call`** and **`yield`**/**`return`**). This also means that an asymmetric coroutine is bound to return to its original caller.

Symmetric coroutines, on the other hand, are not bound to return to their predecessor coroutine [21]. There is only one operation, typically also called **`yield`**, which is used to transfer control to another coroutine. Depending on the coroutine implementation the target coroutine can either be specified explicitly (**`yieldTo`**) and/or implicitly (**`yield`**). Using **`yield`**, without specifying a target coroutine, usually involves some kind of scheduler that selects a coroutine to run.

2.3.2 History

At the assembly language level coroutine techniques have been employed for a long time, although the name "Coroutine" was not used.

The name "Coroutine" was first mentioned in 1963 by Conway [7]. His coroutines were a tool for efficiently implementing multi-pass processes, like compilers. His usage of coroutines is similar to UNIX pipes, where the output of one process is used as the input of another process.

In Simula I and its successor Simula 67 [9], Ole-Johan Dahl and Kristen Nygaard introduced coroutines as a fundamental operation. They had a specific interest in using coroutines for simulation purposes, because the semantics of coroutines are more adequate for simulation than the semantics of subroutines. Thus, Simula I includes a concept similar to coroutines, called *simulation*. Simula 67 introduced the keywords `call`, `detach` and `resume`, which provide a more generic implementation of coroutines. They realized and exploited the fact that coroutines can be used for implementing other language features.

Implementations of Smalltalk [12], which first appeared in 1972, implement the execution stack as a first-class citizen, which allows for a trivial coroutine implementation.

Scheme [1], created in 1975, is one of the first languages that implemented first-class continuations. They are an even more powerful concept than coroutines and can be used as an implementation vehicle for coroutines.

Modula-2 [37], designed and developed between 1977 and 1980 by Niklaus Wirth, also includes coroutines as lightweight, quasi-concurrent processes. They are controlled by the low-level procedures `NEWPROCESS` and `TRANSFER`. The coroutines in Modula-2 are intended primarily for producer/consumer problems, similar to Conway's original design.

The standard C library includes functions named `setjmp` and `longjmp`, which can theoretically be used to implement coroutine-like features, but these rarely-used functions are highly platform dependent and awkward to use. The `setcontext` family of functions, available in POSIX C libraries, is suited for coroutine implementations. But implementations containing these functions are rare and the `setcontext` functions also have significant shortcomings, for example with stack overflow detection.

With the advent of languages like C# and Java, coroutines went out of fashion. Recently, however, there is again a growing interest, and an increasing number of languages provide coroutines. Modern languages with coroutines include Icon, Go [13], Lua [15], Perl, Python and Ruby [11].

2.3.3 Implementation techniques

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. Because of this, stacks also consume considerable amounts of memory (see Section 4.2.2 on page 30), which prevents applications from using large numbers of coroutines.

Copying 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: Firstly, the coroutine transfer becomes 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.

Continuations Every system that provides continuations can easily provide coroutines, by simply capturing and resuming continuations. The following example (syntax taken from [30]) shows how an implementation of symmetric coroutines could look like:

```
1  public class Coroutine {
2    private Continuation continuation = new Continuation();
3
4    // returns the next scheduled coroutine
5    private Coroutine next() {
6      return ...;
7    }
8
9    public void yield() {
10     if (continuation.capture() == Continuation.CAPTURED) {
11       next().continuation.resume(null);
12     }
13   }
14 }
```

In programming languages that represent stack frames as heap objects and a 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, however, need special garbage collection algorithms to free obsolete frames.

Within the constraints of the Java programming language and a JVM it is not possible to use any of these implementation techniques, because it is not possible to gain arbitrary access to the structure of the stack. JVMs also do not provide continuation

support, apart from research projects like [10] and [30]. There are, however, other techniques that can be used:

Threads as coroutines If multiple threads are synchronized in a way that allows only one of them to run at a time, and that provides explicit switching between the threads, they will appear as if they were coroutines running within one thread[4]. The drawback of this approach is that threads consume a lot of memory and that the synchronized switching between threads is an expensive operation.

Compile-time transformations A special compilation step that turns methods into state machines can be used to make methods appear like coroutines at the source level. C#'s implementation of iterator methods performs this transformation for individual methods (stackless), while frameworks like javaflow [3] perform this transformation in a way that simulates stackful coroutines. This approach leads to a significantly larger compiled code size, and the more complicated methods leave the just in time compiler with less opportunities for optimization.

An example of the transformation of a simple C# iterator method is shown in Figure 2.7.

```
using System;
using System.Collections;

class Test {

  // The returned iterator will
  // generate 0 as the first
  // element, 1 as the second, ...,
  // and 9 as the last element.
  // Then it will signal that is has
  // no more elements to offer.
  static IEnumerator GetCounter() {
    for (int c = 0; c < 10; c++) {
      yield return c;
    }
  }
}
```

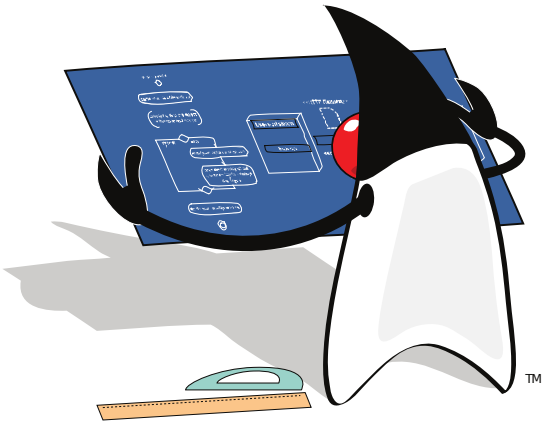
```
private class GetCounter {
  private int state = -1;
  private int count = 0;
  public int current;

  public bool MoveNext() {
    switch (state) {
      case 0:
        while (count < 10) {
          current = count;
          state = 1;
          return true;
        resume_label1:
          state = -1;
          count++;
        }
        break;
      case 1:
        goto resume_label1;
    }
    return false;
  }
}
```

Figure 2.7: A simple C# iterator method (left) and an equivalent state machine (right). Note that the simplified state machine shown here is only a pseudocode approximation. The actual result of the transformation has a length of two pages and is not representable by valid C# code.

[4]Only with limitations. Thread local variables, for example, will not work as expected.

Chapter 3

Coroutines in Java



This chapter explains the basic concepts a programmer needs to know in order to use the coroutine implementation presented in this thesis. It also describes the API that is used to implement, create, use and destroy symmetric and asymmetric coroutines. Then, this chapter shows the API for migrating coroutines between threads and closes with a description of the API used to serialize and deserialize them.

3.1 Concepts

The introduction of a new programming language concept raises a number of questions that need to be answered before the new feature can be implemented:

Should new language constructs be created?

Coroutines could be implemented more elegantly by introducing new keywords like `yield` into the Java language. The implementation presented in this thesis does not do so, because this would lead to problems with backward-compatibility and would require changes to the java source compiler (`javac`), thus making

the implementation more complex. Not introducing a new keyword follows the general effort of the Java language to avoid special-purpose syntax where possible.

How does the programmer access the new features' primitives?
Symmetric and asymmetric coroutines are the basic primitives of a coroutine implementation. The implementation shown here presents coroutines as instances of the `Coroutine` and `AsymCoroutine` classes. These classes contain all methods needed to perform the basic operations upon coroutines.

3.1.1 Thread Affinity

Operations on coroutines implicitly always operate on the current thread. In fact, it is not possible to perform any operations on the coroutines of another thread, other than thread-to-thread migration and read-only operations like querying a coroutine's status.

This allows the system to assume that only a minimal amount of synchronization is necessary, which is important in order to achieve high performance.

3.1.2 Stacks allocated directly from the Operating System

Most garbage collection algorithms used in JVMs perform a compaction step that moves objects in memory. Stack memory cannot be moved to another address because of locking and native stack frames, so it is not possible to allocate it from the ordinary Java heap.

Additionally, stack memory needs to be aligned on page boundaries and guarded by protected memory pages, in order to detect stack overflows. These guarantees and properties can only be acquired directly from the operating system.

The fact that the memory needs to be allocated outside of the normal Java heap makes controlling the resources of an application harder, but on the other hand this is no different than the allocation of threads - most of the memory used by a thread also lies outside of the Java memory subsystem.

3.1.3 Scheduling of Symmetric Coroutines

For symmetric coroutines there needs to be a scheduling strategy. In the system presented in this thesis symmetric coroutines are scheduled on a first come, first served

basis. Each time a coroutine is executed it is moved to the end of the queue of coroutines that will be executed, and it is thus only executed again after all other coroutines have been scheduled.

While the specification of coroutines should be careful about requiring a particular behavior, this scheduling strategy seems to be a reasonable choice that can be implemented very efficiently using a doubly-linked ring of coroutines. Advancing to the next coroutines is achieved by simply moving a pointer to the next coroutine. The current coroutine will then automatically be at the end of the queue.

3.1.4 Input and Output Values of Asymmetric Coroutines

Asymmetric coroutines do not need to be scheduled, because they will only execute when they are called explicitly. However, it is very common for asymmetric coroutines to receive an input value to work with and/or return a result value to the caller each time they are called.

In fact, this behavior is common enough to warrant an explicit implementation, although it might also be implemented by the user on top of a simpler coroutine system. In order to make asymmetric coroutines more convenient and type-safe we specify the types of the input and output values as generic type parameters of the asymmetric coroutine class.

3.1.5 Thread-to-Thread Migration

Due to the thread affinity of coroutines, it is not possible to resume a coroutine on some other thread. In fact, trying to do so will lead to an exception. This is the expected behavior in the normal case and leads to good performance characteristics, but in some cases it is necessary to transfer a coroutine to some other thread.

In an application that implements user interactions as coroutines, with many user interactions running in one thread, a coroutine might be blocked due to a computationally expensive operation at one point in the user interaction. It is not acceptable for this coroutine to block all other coroutines that happen to be on the same thread, thus there needs to be a way to move coroutines to other threads.

3.1.5.1 Coroutine Stealing

When moving coroutines from one thread to the other there are four possible semantics depending on who takes the active role in the process:

Source and Target Thread Both the source and the target thread need to reach a synchronization point where the coroutine is handed over from one thread to the other.

Source Thread The source thread "injects" the coroutine into the target thread.

Target Thread The target thread "steals" the coroutine from the source thread.

Third Party Thread A third party thread takes the coroutine from the source thread and gives it to the target thread.

In the case of a blocked coroutine it is impractical to require the source thread to take part in the process, because it is busy and will likely not reach a synchronization point quickly. The target thread, however, is idling anyway (otherwise it would not try to acquire new coroutines), so it makes sense for it to take the active role in the process. This is called *coroutine stealing*, analogous to *work stealing*, which is an established concept in concurrent programming.

3.1.5.2 Migration Process

Coroutines that are moved from one thread to the other need of course be aware of the changes they will see after migration: `Thread.current()` will return the new thread, and the value of thread locals might also have changed.

Also, a coroutine cannot be transferred to another thread if there are native method frames on the stack. The VM does not have sufficient information to relocate native frames to another stack, which is required during the transfer process.

Stack frames of methods which hold locks (via the **`synchronized`** keyword) also cannot be transferred to another thread, because locks are bound to threads. Additionally, the implementation would be much more complex if locks needed to be transferred.

3.1.6 Serialization/Deserialization

When a coroutine has been suspended for a long time, but may be needed in the future, it might be best to move the coroutine into permanent storage (disk, DB, ...) in order to free the memory associated with it. This is especially important in systems that sustain a large number of concurrent sessions, e.g. web servers that support many concurrent user interactions.

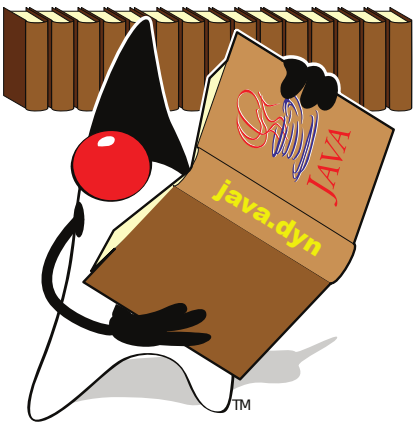
Java serialization allows a program to serialize all of its data, but it is not possible to serialize a thread. The coroutine system, on the other hand, provides a way to serialize a coroutine, which essentially turns a coroutine into an array of Java objects. Each

of these objects represents one Java activation frame, and contains the frame's current method, bytecode index, local variables and expression stack. The application that uses the coroutine serialization mechanism can then decide how it wants to handle the actual serialization of these values. In a sense, the coroutine serialization mechanism does not actually perform the serialization, it only makes the contents of the coroutine's stack frames accessible to serialization.

This also means that for coroutine serialization the user has to deal with the same problems as for other uses of the Java serialization mechanism: Where to make the cut between the serialized and non-serialized parts of the program state and what to do with non-serializable objects. This is application-dependent and cannot be decided by the coroutine implementation. An application can use the object-replacement functionality provided by the `ObjectInputStream` and `ObjectOutputStream` classes to communicate its requirements to the Java serialization mechanism.

The restrictions for coroutine serialization are similar to the restrictions for thread-to-thread migration: The coroutine cannot contain native frames or frames with locked methods.

3.2 Coroutine 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. The Java API presented here tries to be consistent with the majority of programming languages and to fit 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. The size of a new coroutine's stack can be specified via the `stackSize` parameter at construction. Like the `stackSize` parameter of the `Thread` constructors this is only a hint which may or may not be obeyed by the runtime system. The system also enforces a platform-specific minimum value for this parameter.

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

The user can also provide the runtime with a hint that two specific coroutines should share a stack. To do so the constructor of the second coroutine needs to be called with the `sharedStack` parameter set to the first coroutine. Again, this is only a hint that may or may not be obeyed by the runtime system.

Both `Coroutine` and `AsymCoroutine` extend the class `CoroutineBase`, which contains implementation-specific code that is common to both symmetric and asymmetric coroutines.

3.2.1 Symmetric Coroutines

The interface for symmetric coroutines is shown in Figure 3.1. It is similar to `Thread` in that it can either be subclassed (overriding the `run()` method) or supplied with a `Runnable` at construction[1].

Each thread maintains the coroutines that are associated with it in a data structure that provides fair scheduling. The details of the scheduling algorithm are explained in Section 3.1.3. The coroutine system inserts each newly created `Coroutine` object after the current one, which means that it is the next coroutine to be scheduled. This also means that there is no need to register a coroutine with the runtime system, and that coroutines can only be added to the current thread. While this might seem to be an arbitrary restriction it is vital for the performance of the system, as explained in Section 3.1.1.

There are two ways to switch to another coroutine:

- **`yield`**`()` transfers control to the next coroutine, as determined by the scheduling algorithm. Under normal circumstances this means that the current coroutine will only be activated again after all other coroutines.

[1] `Coroutine`, however, does not implement `Runnable`. In hindsight it was a poor decision for `Thread` to do so.

```
1 package java.dyn;
2
3 public class Coroutine extends CoroutineBase {
4
5   public Coroutine();
6   public Coroutine(Runnable r);
7   public Coroutine(long stackSize);
8   public Coroutine(Runnable r, long stackSize);
9   public Coroutine(Runnable r, CoroutineBase sharedStack);
10
11  public static void yield();
12  public static void yieldTo(Coroutine target);
13
14  public boolean isFinished();
15  protected void run();
16 }
```

Figure 3.1: Coroutine Java API: `Coroutine` class

- **`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. The user of this method cannot make assumption as to how it influences scheduling, other than that the future calls to **`yield`**`()` will again be scheduled in a fair way.

Both these methods are static because they will often be used in a context where the current coroutine object is not readily available. They also always act on the current thread's list of coroutines, following the general paradigm that coroutines are strongly coupled to the thread they are running on. Note that the main program, i.e., the thread's initial execution context, is a coroutine as well.

The instance method `isFinished()` returns true if the `Coroutine` in question has ended by either reaching its end, by executing a **`return`**-statement or by throwing or propagating an exception.

Figure 3.2 shows an example of a simple coroutine that produces the following output:

```
1 start
2 Coroutine running 1
3 middle
4 Coroutine running 2
5 end
```

In practice it would be better to specify the program using the `Runnable` interface, which leads to the same behavior. This is shown in Figure 3.3.

```
1  public class ExampleCoroutine extends Coroutine {
2
3    public void run() {
4      System.out.println("Coroutine running 1");
5      yield();
6      System.out.println("Coroutine running 2");
7    }
8
9    public static void main(String[] args) {
10     new ExampleCoroutine();
11     System.out.println("start");
12     yield();
13     System.out.println("middle");
14     yield();
15     System.out.println("end");
16   }
17 }
```

Figure 3.2: Example for the usage of `Coroutine`

```
1  public class ExampleCoroutine implements Runnable {
2
3    public void run() {
4      System.out.println("Coroutine running 1");
5      Coroutine.yield();
6      System.out.println("Coroutine running 2");
7    }
8
9    public static void main(String[] args) {
10     new Coroutine(new ExampleCoroutine());
11     System.out.println("start");
12     Coroutine.yield();
13     System.out.println("middle");
14     Coroutine.yield();
15     System.out.println("end");
16   }
17 }
```

Figure 3.3: Example for the usage of `Coroutine` with the `Runnable` interface

3.2.2 Asymmetric Coroutines

For reasons of generality, we also provide an implementation of asymmetric coroutines, called `AsymCoroutine`, shown in Figure 3.4. Instances of this class can be created by either subclassing `AsymCoroutine` (overriding the `run` method) or providing an instance of `AsymRunnable` (shown in Figure 3.5).

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

`AsymCoroutines` 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

```
1  package java.dyn;
2
3  public class AsymCoroutine<I, O> extends CoroutineBase
4                                   implements Iterable<O> {
5
6    public AsymCoroutine();
7    public AsymCoroutine(AsymRunnable<? super I, ? extends O> target);
8    public AsymCoroutine(long stackSize);
9    public AsymCoroutine(AsymRunnable<? super I, ? extends O> target,
10                        long stackSize);
11   public AsymCoroutine(AsymRunnable<? super I, ? extends O> target,
12                        CoroutineBase sharedStack);
13
14   public O call(I input);
15   public I ret(O output);
16
17   public boolean isFinished();
18   protected O run(I input);
19
20   public Iterator<O> iterator();
21 }
```

Figure 3.4: Coroutine Java API: `AsymCoroutine` class

```
1  package java.dyn;
2
3  public interface AsymRunnable<I, O> {
4
5    public O run(AsymCoroutine<? extends I, ? super O> coroutine,
6                 I value);
7  }
```

Figure 3.5: Coroutine Java API: `AsymRunnable` interface

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.

`AsymCoroutine` and `AsymRunnable` use co- and contravariant generic type parameters in order to allow for a maximum of compatibility. An `AsymCoroutine<I, O>` can be initialized with any `AsymRunnable` that can work with any input parameter that can be cast to `I` and that will return an output parameter that can be cast to `O`. The `run` method of `AsymRunnable` has an additional parameter `coroutine`, which provides the asymmetric coroutine object needed for calls to `ret`, again with a co- and contravariant type. For example: An `AsymCoroutine<Integer, Object>` can be initialized with an `AsymRunnable<Number, String>`, because the runnable takes an input that is less specific and returns an output that is more specific.

The fact that `AsymCoroutine` implements `Iterable` allows such coroutines to be used in enhanced for loops (see Figure 3.7). 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:

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

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

- **`boolean`** `isFinished()` returns true if the `AsymCoroutine` in question has reached its end. Invoking `call` on a `AsymCoroutine` that is not alive leads to an exception.

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

Figure 3.6 shows an example `AsymCoroutine` that inverts a SAX parser [22] to return one XML element at a time. Figure 3.7 contains an alternative version of the main method that uses an enhanced for loop.

3.2.3 Thread-to-Thread Migration

The API for thread-to-thread migration of coroutines is very simple and consists of one method that is available in both the `Coroutine` and `AsymCoroutine` classes:

```
public boolean steal();
```

This method will migrate the coroutine it is called upon to the current thread. If the transfer was successful it will return **`true`**, otherwise it will return **`false`** or throw an exception.

The operation can fail for a number of reasons:

```
1  public class CoSAXParser
2              extends AsymCoroutine<Void, String> {
3
4    public String run(Void input) {
5      SAXParser parser = ...
6      parser.parse(new File("content.xml"),
7        new DefaultHandler() {
8          public void startElement(String name) {
9            ret(name);
10         }
11       });
12     return null;
13   }
14
15   public static void main(String[] args) {
16     CoSAXParser parser = new CoSAXParser();
17
18     while (!parser.isFinished()) {
19       String element = parser.call(null);
20       System.out.println(element);
21     }
22   }
23 }
```

Figure 3.6: SAX parser inversion

```
1    ...
2    public static void main(String[] args) {
3      CoSAXParser parser = new CoSAXParser();
4
5      for (String element: parser) {
6        System.out.println(element);
7      }
8    }
9  }
```

Figure 3.7: SAX parser inversion using enhanced for loops

- It will return `false` if the coroutine is busy. This will happen if the coroutine is currently running or the coroutine's current thread performs an operation on it, e.g., serialization. Any use of the migration mechanism needs to be prepared for this to happen, because the migration is performed without busy waiting and on a best-effort basis.

- It will throw an `IllegalArgumentException` if:

  - The coroutine already belongs to the current thread.
  - The coroutine is the initial coroutine of its thread. The initial coroutine can never be migrated to some other thread, because this coroutine is needed in order to correctly terminate the thread.
  - The coroutine contains a stack frame of a method that holds a lock.
  - The coroutine contains a native stack frame.

– The coroutine is an asymmetric coroutine that has not yet returned to its calling coroutine.

The distinction between a **false** return value and an exception is intended to differentiate between failures that can happen for any use of `steal` (because of the fact that migration is performed on a best-effort basis) and failures that will not occur for correct uses of the migration mechanism.

When a symmetric coroutine is migrated it is removed from its thread's scheduling sequence and added to the current thread's sequence as the next coroutine to be executed.

Symmetric coroutines that are migrated need to be aware that `Coroutine.yield` will now schedule different coroutines, and that calls to `Coroutine.yieldTo` that were valid before will now likely be invalid.

Symmetric and asymmetric coroutines need to be aware that calls to an asymmetric coroutine (via `AsymCoroutine.call`) that were valid before will now be invalid, unless the asymmetric coroutine was migrated as well.

3.2.4 Serialization / Deserialization

```
1  package java.dyn;
2
3  import java.io.Serializable;
4  import java.lang.reflect.Method;
5
6  public class CoroutineFrame implements Serializable {
7
8    public Method method;
9    public int bci;
10
11   public int localCount;
12   public int expressionCount;
13
14   public long[] scalarValues;
15   public Object[] objectValues;
16
17   public CoroutineFrame(Method method, int bci, int localCount,
18    int expressionCount, long[] scalarValues, Object[] objectValues);
19 }
```

Figure 3.8: Coroutine Java API: `CoroutineFrame` class

Serialization and deserialization is also available for both symmetric and asymmetric coroutines. The main task of these two mechanisms is to transform the stack of a coroutine into a data structure that is accessible from Java, and vice-versa. The class

`CoroutineFrame`, which is used to represent stack frames, is shown in Figure 3.8 and contains the following fields:

method This field contains the method that this stack frame belongs to. It is important to note that `java.lang.reflect.Method` is not serializable. An application that serializes `CoroutineFrame` objects needs to be aware of this.

bci The bytecode index of the next instruction to execute in the method.

localCount The number of local variables in this stack frame.

expressionCount The number of stack slots on the expression stack that were used for evaluating an expression at the time the coroutine was suspended. A stack frame will only contain temporary expressions if the coroutine was suspended while calculating an expression, i.e., during a function call.

scalarValues This array of length `localCount + expressionCount` contains the scalar values of all local variables and the expression stack. Entries that contain an object reference have no scalar value, and the `scalarValues` array contains a zero value in that case.

objectValues This array is of the same length as `scalarValues` and contains the object references of all local variables and the expression stack. Entries that contain a scalar value have no object reference, and the `objectValues` array contains a null value in that case.

The serialization mechanism can of course not only be used for storing coroutines to disk. In fact, the ability to modify the stack of a running application can be useful for dynamic languages that compile to bytecode on the fly (Ruby, JavaScript, etc.): If the language runtime detects that a piece of code that is executed by an interpreter loop is taking too long, it can suspend the execution, replace the interpreter loop with compiled (to bytecode) methods and resume the execution. This allows languages that run on top of a JVM to perform on-stack replacement at the Java level, similar to the JVM itself, which performs on-stack replacement at the native code level.

Arbitrarily modifying stack frames of course requires an application to have extensive knowledge about the structure of the compiled methods, which is not provided by the coroutine implementation.

An example that uses the serialization mechanism to perform session persistence is shown in Section 5.2.

3.2.4.1 Serialization

The serialization of coroutines is supported via the following method that is available in both symmetric and asymmetric coroutines:

```
public CoroutineFrame[] serialize();
```

If the coroutine contains stack frames of native methods or stack frames that currently hold locks, which cannot be transformed into `CoroutineFrame` objects, it will throw an `IllegalArgumentException`. It will also throw an exception if the coroutine is the initial coroutine of a thread, which cannot be serialized because it will always contain native frames.

If successful, this method will return an array of `CoroutineFrame` objects. If the coroutine's stack contained references to Java objects, then the resulting `CoroutineFrame` objects will contain references to the same objects, as shown in Figure 3.9.

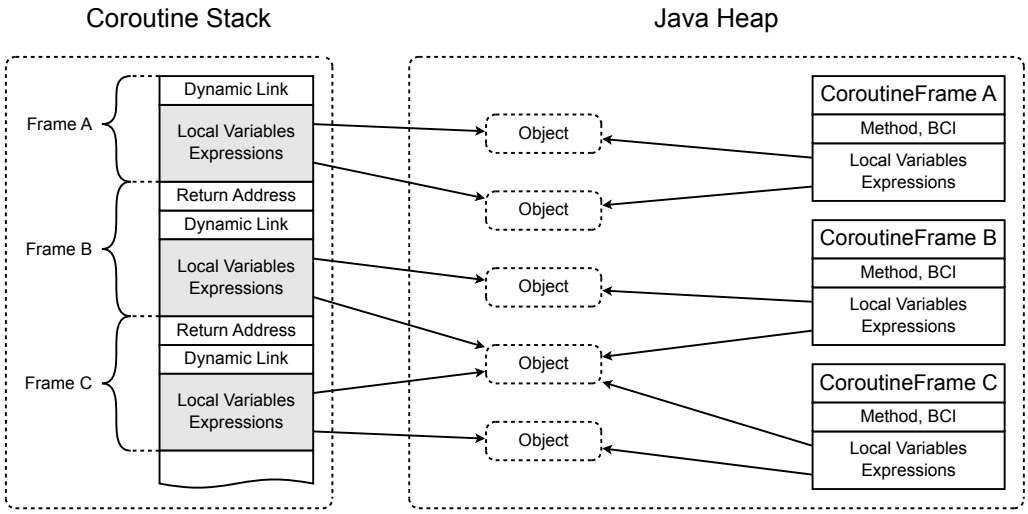


Figure 3.9: A coroutine stack (left) and the equivalent `CoroutineFrame` objects (right).

In the simplest case an application can serialize the array of `CoroutineFrame` objects using an `ObjectOutputStream`. It only needs to take care of non-serializable classes, as shown in Section 5.2.

In a more complex application it might be necessary to perform additional substitutions, e.g., replace objects that represent database entities with a serializable form or take care of other objects that are not serializable. Applications that keep coroutines in a serialized state for only a limited amount of time might associate IDs with non-serializable objects, and store these objects in a hash table or a similar data structure.

3.2.4.2 Deserialization

Similar to serialization the deserialization is also performed by a method that is available in both symmetric and asymmetric coroutines:

```
public void deserialize(CoroutineFrame[] frames);
```

This method is the opposite of `serialize`: it takes an array of `CoroutineFrame` objects and replaces the target coroutine's stack with the given frames. It will fail if the coroutine is the thread's initial coroutine, in which case it will throw an `IllegalArgumentException`.

Again, if the `CoroutineFrame` objects contain references to Java objects, then the coroutine's stack frames will also contain references to these objects, as shown in Figure 3.9.

Chapter 4

Implementation

This chapter is the main chapter of this thesis. It describes the implementation details of the presented algorithm, which combines the advantages of the two most common implementation techniques. It provides a detailed description of the actual coroutine management implementation. After showing the implementation of the thread-to-thread migration this chapter ends with an overview of the serialization/deserialization code.

4.1 Introduction

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.

This thesis presents 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 numbers of coroutines to be allocated.

This system incurs only a moderate amount of changes to the JVM. Apart from the additional stack management code (which ties into the thread 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 are required.

Our implementation of coroutines for the Java HotSpot™ VM on the Windows and Linux platforms shows that it is feasible for JVMs and managed runtimes in general.

4.2 Coroutine Management

4.2.1 Additional Stacks

In order to provide optimal performance, the 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 (frame pointer, stack pointer, program counter, callee saved registers) need to be saved and restored.

The required memory is requested from the operating system's low-level memory management system. Then the operating system is instructed to provide (non-executable) memory for the usable part of this stack area, because the operating system initially only reserves address space, without actually providing any real memory.

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.

4.2.2 Stored Coroutines

Separate stacks take up significant amounts of memory and address space. A stack occupies at least 32 kB (Linux), 64 kB (Windows) or 100 kB (Solaris) of memory, due to guard pages and the operating system's allocation granularity (see Section 6.1). On

32-bit systems the address space is exhausted after the allocation of roughly 25,000 stacks. 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.

Applications that use coroutines typically have a deeply nested stack while a coroutine is running, and a shallow stack while it is suspended. Thus, only a small amount of the stack is actually used by a suspended coroutine, typically 1 to 2 kB. The coroutine implementation can exploit this by letting multiple coroutines share a single stack, and storing the active contents of the stacks into temporary objects.

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. For example, native frames might depend on the addresses of stack-allocated objects. Some library functions, like reflection, are implemented using native code. Thus, if the system would relocate coroutines to other stacks, it would not be able to switch to and from coroutines that have methods called via `Method.invoke` on the stack.

Allowing coroutines to be restored on another stack would thus lead to performance degradation and the inability to use native frames within coroutines.

Note that relocation of frames is only a problem if the relocated frames are executed at their new positions; 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 via a simple hash function. The programmer can also exploit knowledge about the behavior of the application by selecting a coroutine's stack explicitly, it is possible to do so by specifying some other coroutine that will share its stack with the newly created one.

4.2.3 Coroutine State

As coroutines are copied to and from their stacks they need to keep track of their state. Figure 4.1 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 on the heap. Every new coroutine is created in this state, with a small stub data

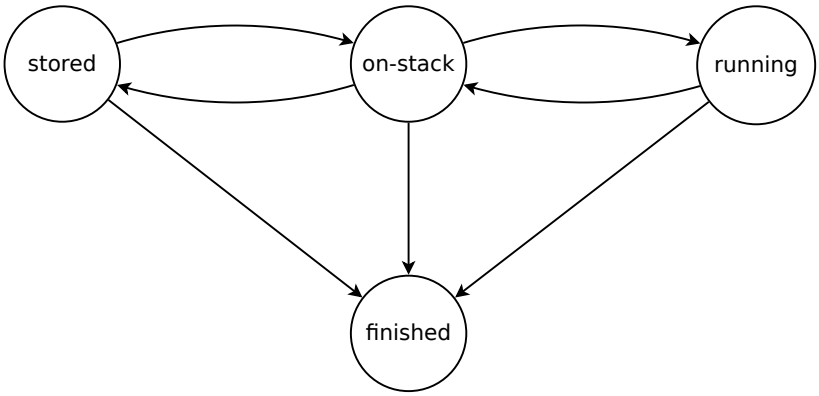


Figure 4.1: Coroutine states and transitions

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. Note 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.

4.2.4 Data Structures

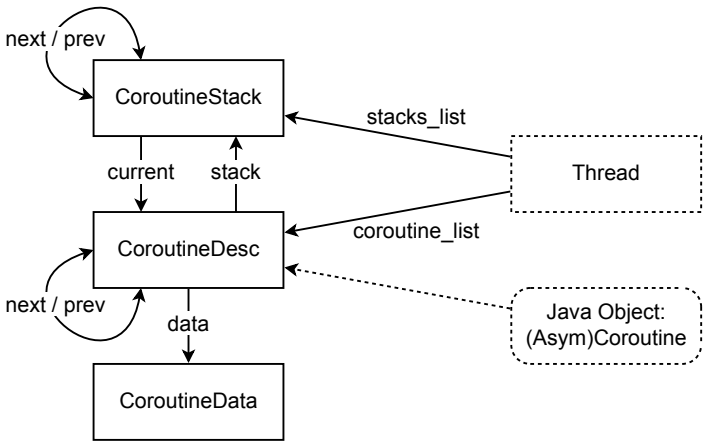


Figure 4.2: Native coroutine data structures

The VM uses a set of data structures to keep track of all existing coroutines (see Figure 4.2):

Thread is the preexisting data structure for threads. It is enhanced with a list of coroutines (`coroutine_list`) and a list of coroutine stacks (`stack_list`). Both are implemented as circular doubly-linked lists. Note that the thread's initial execution context 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. The doubly-linked list, combined with the fact that the initial elements will never be removed, leads to very simple add and remove operations with no special cases.

CoroutineDesc holds the stored CPU context while the coroutine is not running. It contains VM-internal information that is coroutine-local (e.g., for resource allocation and pointers from native code to Java objects). 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 object in the form of a `java.lang.Thread` instance, there is an instance of `java.dyn.Coroutine` or `java.dyn.AsymCoroutine` for each coroutine (see Section 3.2). 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.

4.2.5 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 selects coroutines that share stacks using a simple hash function.

- If 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. Thus, a newly created coroutine is in the state "stored".

- Finally the `CoroutineDesc` structure is created and associated with the `Coroutine` or `AsymCoroutine` Java object.

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

4.2.6 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 4.3. In this case there are two coroutine stacks, one of which is shared by two coroutines. The first stack contains the coroutines α and β, while the second stack only contains γ.

Initially α is running, which implies that the stack frames of β are stored in the `CoroutineData` structure. γ is the only coroutine associated with the second stack, so its stack frames are on the stack even though it is not running.

The second part of the example shows the situation after the switch from α to β: In order to make way for the stack frames of β the stack frames of α were transferred into its `CoroutineData` structure. This switch of course did not affect γ and the second stack.

The third part shows the situation after the switch from β to γ: There was no need to transfer any stack frames to or from a stack, the only change is that β is now "on-stack" and γ changed to "running".

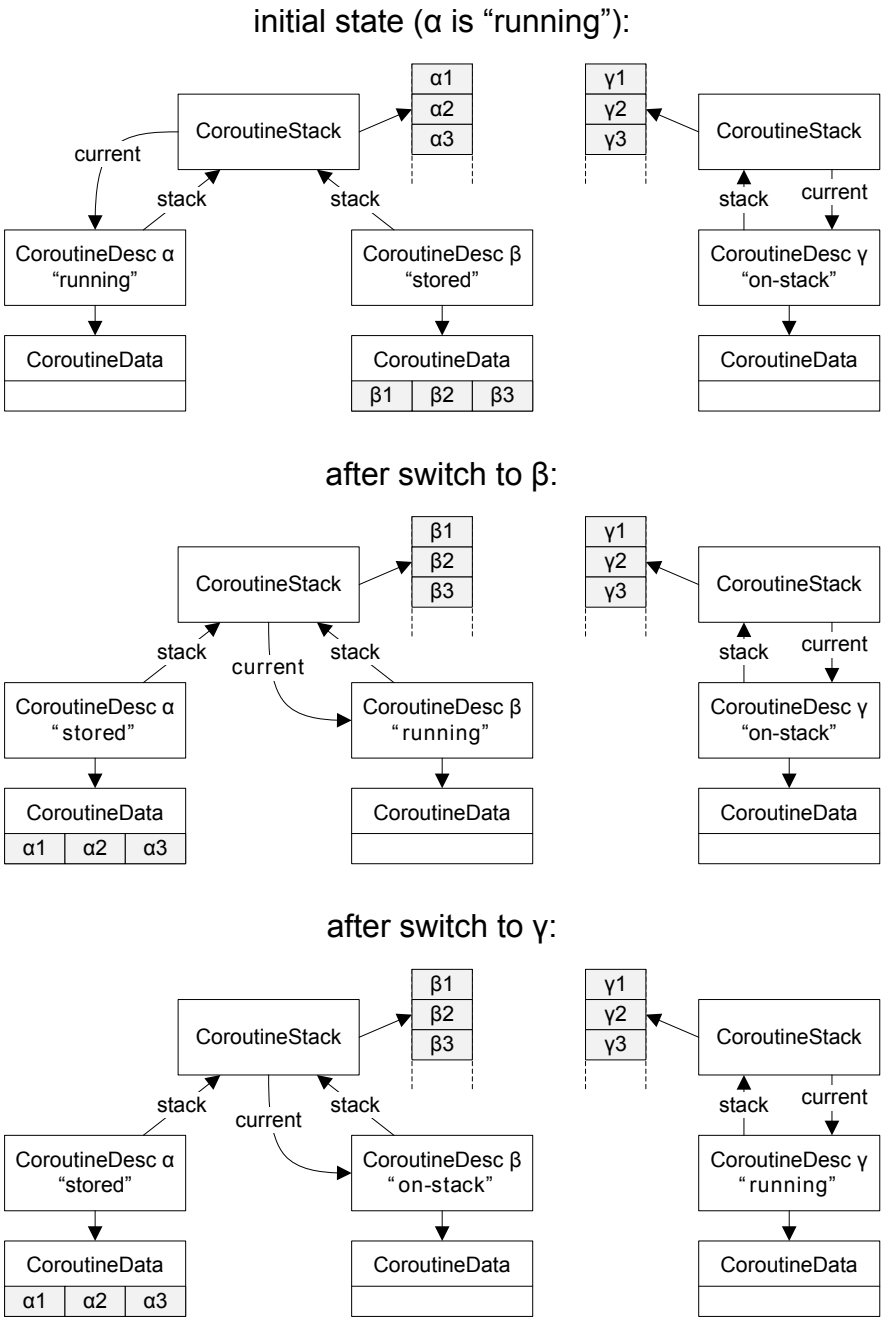


Figure 4.3: Data structures before and after context switch

The core functionality is implemented 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 assembly instructions). Depending on the operating system, the CPU, and the VM configuration, our system is comprised of approximately 100-120 assembly instructions[1], 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 lines in Figure 4.4, which are explained in the next paragraph).

Figure 4.4 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 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.

The system first checks whether the target coroutine belongs to the current thread. It is not possible to switch to some other 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 occupied by some other coroutine, the system checks if the source 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 a Java 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.

[1] The amount of assembly instructions created can vary. The code is created at VM startup and is tailored towards the actual configuration: garbage collection mechanism, heap layout, compiler, etc.

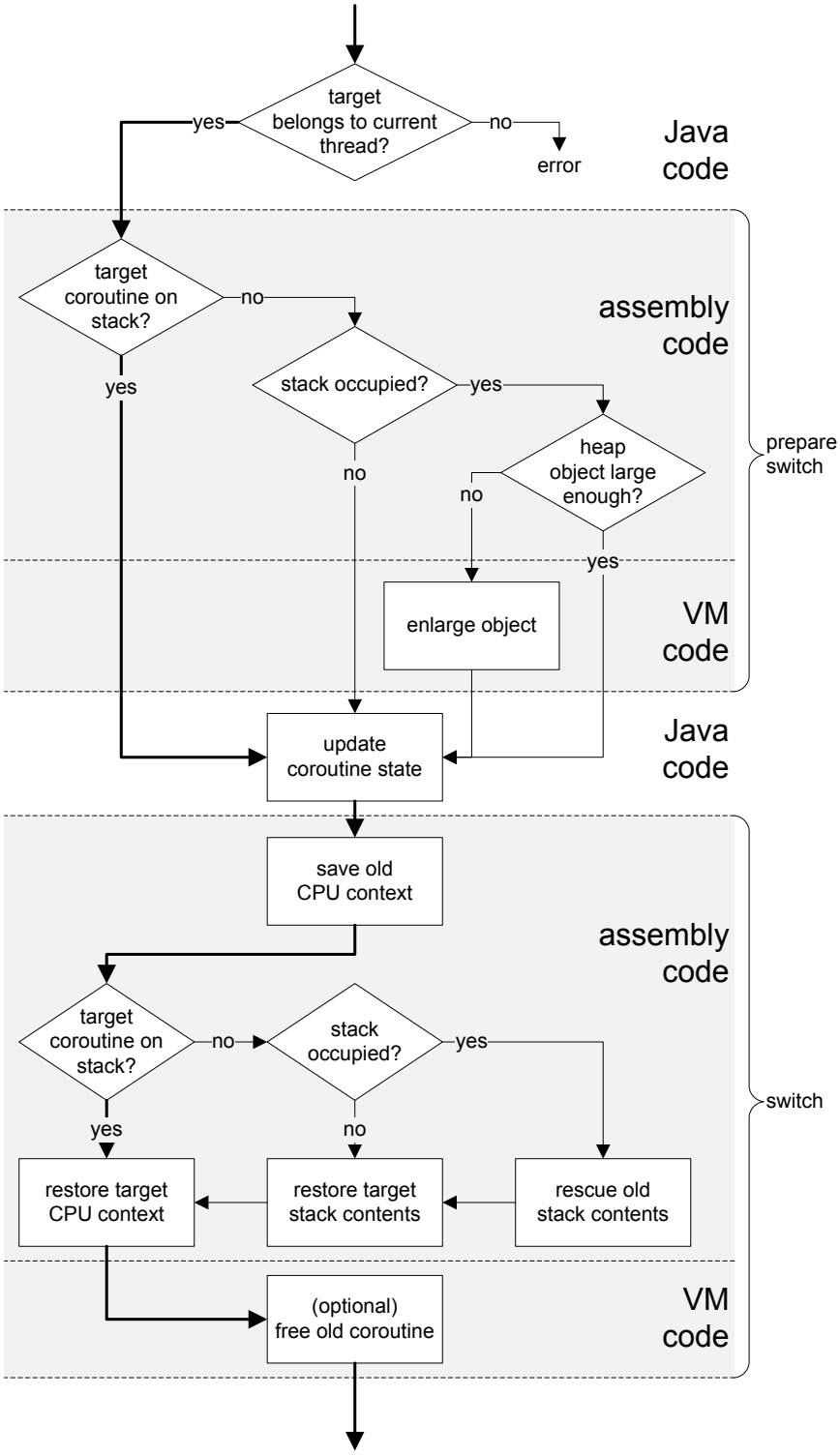


Figure 4.4: Coroutine context switch operation

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.

4.2.7 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* [18].

- From time to time the JVM tries to collect unused compiled methods. To do so it also needs to be able to walk over all method activations in the system.

- 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 return addresses of compiled frames and the method pointers of interpreted frames serve as pointers to the methods that the stack frames belong to.

The thread's `coroutine_list` is used to iterate over all live coroutines when needed. The stack of the currently running coroutine is walked automatically, because it is seen as the thread's stack by the garbage collection system. If a coroutine is on-stack then its stack is walked in a way similar to the ordinary thread stack walking. 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.

4.2.8 The CoroutineSupport Class

As much as possible of the coroutine logic is implemented in Java code. The main part of this Java code is contained in the class `CoroutineSupport`, which is a helper class that has exactly one instance for each thread that uses coroutines. It is responsible for managing the Java objects of class `Coroutine` and `AsymCoroutine` (as shown in Figure 4.2) and for forwarding the API calls (as shown in Section 3.2) to the underlying VM.

The `CoroutineSupport` class contains the following features:

Initialization Each thread that uses coroutines needs to be accompanied by a number of data structures, which are set up by initialization code. This also creates the thread's initial coroutine, which represents the thread's initial stack.

Debugging/Tracing Two flags allow for debugging and tracing to be enabled, which leads to verbose output on the console for each coroutine operation. This is useful mainly for debugging internals of the system, but can be used to illustrate coroutine operations.

Stack Management `CoroutineSupport` contains code to manage the stacks of the thread it is associated with and to determine the number of stacks per thread and the hashing parameters. It also keeps a list of all stacks associated with the thread.

Coroutine Management A doubly-linked list of all the Java objects representing symmetric coroutines is kept, which is used for coroutine scheduling. There is also a pointer to the current symmetric or asymmetric coroutine, i.e., `Coroutine` or `AsymCoroutine` object. If the current coroutine is an asymmetric coroutine, then the current symmetric coroutine can be determined by following the caller chain of the asymmetric coroutine.

Symmetric/Asymmetric Distinction The distinction between symmetric and asymmetric coroutines is only visible on the Java side, and `CoroutineSupport` contains code for the scheduling of symmetric coroutines and for the call/return behavior of asymmetric coroutines.

Coroutine Creation A number of data structures need to be set up for each new coroutine, along with the actual VM-internal coroutine object (`CoroutineDesc`, as shown in Figure 4.2). This setup is slightly different for symmetric and asymmetric coroutines.

Yield The `CoroutineSupport` class implements the Java parts of Figure 4.4, which means that it checks if a yield operation is valid and updates the coroutine states.

Thread End When a thread ends the system needs to make sure that all coroutines have finished cleanly. Otherwise the system could violate basic JVM invariants, for example finally blocks could possibly never be executed. In order to avoid this the system drains all coroutines before the thread is allowed to actually end, which means that each coroutine is sent a `CoroutineExitException` until only the thread's initial coroutine is left.

4.2.9 Java / C++ Interface

The `CoroutineSupport` class also contains the definitions for a number of native methods which provide the interface to the underlying native code. These methods return pointers to internal data structures as **`long`** values. While this might seem like a security hole it is important to note that these values are never exposed to application code.

```
native long getThreadCoroutine();

native long createCoroutineStack(long stack_size);
native void freeCoroutineStack(long stack);

native long createCoroutine(CoroutineBase coro, long stack);
native void freeCoroutine(long coroutine);

native void prepareSwitch(CoroutineBase target);
native void switchTo(CoroutineBase current, CoroutineBase target);
native void switchToAndTerminate(CoroutineBase current,
                                 CoroutineBase target);
native void switchToAndExit(CoroutineBase current,
                            CoroutineBase target);
```

`CoroutineBase` is a common base class of symmetric and asymmetric coroutines.

- The `getThreadCoroutine` method retrieves the thread's initial coroutine and, if necessary, sets up the native thread for coroutine usage.
- The `...CoroutineStack` methods are used to create coroutine stacks with a specified size (-1 to use the system default) and to free coroutine stacks.
- The native data structures for coroutines are created and destroyed by the `...Coroutine` methods.
- The coroutine switch operation is performed by the `prepareSwitch` and `switchTo...` methods. `switchTo` just passes control to the target coroutine, while `switchToAndTerminate` terminates the current coroutine after the switch, which is used in case it has reached its end. `switchToAndExit` will raise a `CoroutineExitException` after the switch is finished, which, if the exception is not handled, will cause the target coroutine to terminate.

4.3 Thread-to-Thread Migration

When running compiled or interpreted methods, a JVM can normally assume that one run of a method will always be executing on the same thread. However, when a coroutine is migrated to some other thread this assumption is broken, and the JVM needs to take additional steps to make sure that the code will execute as expected on its new thread.

When a coroutine migrates to a new thread it cannot simply take its stack with it, because this stack might be shared with coroutines of the old thread. Thus the migrated coroutine needs to get a new stack or be associated with a stack of the new thread. The JVM needs to be careful when moving a coroutine to some other stack, to ensure that all references to the old stack are replaced with references to the new stack.

Part of the migration is performed by VM-internal code, which is accessed via the following native method in `CoroutineSupport`:

```
native boolean stealCoroutineData(long coroutineData, long stack,
                    Thread thread, CoroutineSupport coroutineSupport);
```

This native method performs all the checks and modifications that require an iteration over the coroutine's stack frames.

4.3.1 Locking

Coroutine migration is performed by the target thread, which "steals" the coroutine from its current thread. This means that the source thread is running during the migration process.

The system needs to ensure that the source and target threads do not access the coroutine concurrently, which would lead to undefined and, most likely, faulty behavior (e.g., the source thread starts executing the coroutine during migration).

In order to make sure that the coroutine is accessed exclusively, the Java coroutine classes contain a field that is used for locking. This field is modified via a compare-and-swap instruction, which is the most efficient locking scheme available on x86 hardware.

Compare-and-swap is not a Java primitive, nor is it available in the public class library. It is available as a method of the `sun.misc.Unsafe` class, which, although proprietary in nature, is available on all major JVMs. In the case of the HotSpot™ JVM this method is recognized as a so-called *intrinsic method* which compiles into a single assembly instruction.

4.3.2 Restrictions on Thread-to-Thread Migration

As soon as exclusive access is established the system can check if the migration of this coroutine is a valid operation.

First it checks that the coroutine does not already belong to the current thread, that the coroutine is not currently running and that the coroutine is not the initial coroutine of its thread. In case of an asymmetric coroutine it also makes sure that the coroutine is not currently calling some other asymmetric coroutine.

Then the system iterates over all the stack frames of the coroutine, to determine if there are stack frames of native methods or stack frames of methods with locks. This *stack walking* is implemented very efficiently for both compiled and interpreted frames in most JVMs, because it is needed for performance-critical operations, e.g., during garbage collection or when expanding locks in a biased locking scheme [27].

4.3.3 Migration Process

As soon as the validity is established the actual migration will start.

If the coroutine is currently on-stack, the stack contents will be moved into the `CoroutineData` structure (similar to the "rescue old stack contents" step in Figure 4.4).

Then the coroutine is assigned a new stack that belongs to the target thread. The difference (delta) between the address of the old stack and the address of the newly chosen stack is calculated. During another stack walk this delta is used to correct all pointers in stack frames that point to locations on the old stack, while also replacing all references to internal data structures of the old thread with references to the new thread.

The location of thread references is not normally needed, thus the data structure that stores debugging information was extended to include these locations.

After the stack walk the coroutine is ready to be executed in the new thread, and it is added to the new thread as if it were a new coroutine (see Section 4.2.5). Finally, the system releases the lock on the coroutine object, after which the migration is finished.

4.4 Serialization / Deserialization

Coroutine serialization undoes all the optimizations the JVM performs while compiling and linking methods, thus extracting the state of the hypothetical stack-based Java machine. This extraction is also needed when debugging Java applications.

The HotSpot™ JVM is prepared to start debugging an application at any time, even if parts of it have been compiled to optimized machine code (this is called *full speed debugging*). The VM-internal code that supports this debugging mechanism can extract the Java state of stack frames at any time, and can thus be used for coroutine serialization.

Part of the serialization and deserialization is performed by VM-internal code, which is accessed via the following native methods in `CoroutineSupport`:

```
native CoroutineFrame[] serializeCoroutineData(long data);
native void replaceCoroutineData(long data, CoroutineFrame[] frames);
```

The first method is used for serialization, and the second method is used for deserialization.

4.4.1 Restrictions on Serialization

In order to make sure that the serialization operation is valid, the system checks that the coroutine belongs to the current thread, that the coroutine is not currently running and that the coroutine is not the initial coroutine of the current thread. In case of an asymmetric coroutine it makes sure that the coroutine is not currently calling some other asymmetric coroutine.

Then the system needs to iterate over all the stack frames of the coroutine, to determine if there are stack frames of native methods or stack frames of methods with locks. This check is performed during the serialization itself and does not require a separate stack walk.

4.4.2 Serialization

The serialization itself is performed by iterating over all stack frames of the coroutine and using the debug information to extract the Java state of the method activations. Note that while one interpreted frame always contains one method activation, compiled frames might contain more that one method activation due to inlining.

During the stack walk the system creates a `CoroutineFrame` object for each method activation. The `CoroutineFrame` objects contain an activation's method, bytecode index, local variables and used stack slots. The frame objects are collected into an array of `CoroutineFrame` objects and returned to the caller.

4.4.3 Deserialization

Deserialization replaces a new coroutine's stack contents with stack frames stored in an array of `CoroutineFrame` objects. The system does not check whether the new stack frames are valid, e.g., if the types of local variables correspond to the expected types at the given bytecode index.

A new `CoroutineData` structure is created for the new stack contents, and this structure is filled with one interpreter stack frame for each element of the `CoroutineFrame` array. Interpreter frames have a fixed layout that can be created efficiently.

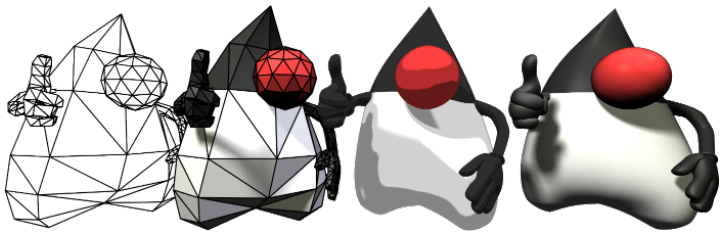
It is important to note that it is not possible to directly create compiled stack frames. The JIT compiler does not provide enough information to do so. For some compilations it will even be theoretically impossible, since the compiled stack frames might contain intermediate calculation results that cannot be recreated.

An interpreted frame that was created by coroutine deserialization and that is executing a long-running loop will be detected and replaced with a compiled method by the JVM's *on-stack replacement* (OSR) facility.

The new `CoroutineData` structure is associated with the coroutine, and the next time a switch to the coroutine occurs the deserialized stack frames will be copied to the coroutine's stack and executed.

Chapter 5

Case Studies and Examples



This chapter presents a number of examples that both motivate and explain the usage of coroutines for real world applications. Although the examples are only small in size, they are sufficient to show the advantages of coroutines for numerous tasks. They also motivate the more advanced concepts of thread-to-thread migration and serialization.

5.1 Machine Control

Machine control programs typically have a fixed cycle time, for example 1 ms. In each cycle all control programs for the different components are executed exactly once. Some of them are simple and perform almost the same actions in every cycle, while others are more complex and need to perform a series of actions over a longer period of time. In general every component only executes a few instructions before switching to the next one, so that the total execution time is influenced heavily by the time it takes to perform a switch.

The following example is influenced by the author's experience with the Monaco programming language [25], which is a domain-specific language for machine control applications.

The example program controls an injection molding machine, which is used to produce parts from thermoplastic materials by injecting molten plastic into a mold cavity, letting it cool and ejecting the finished part.

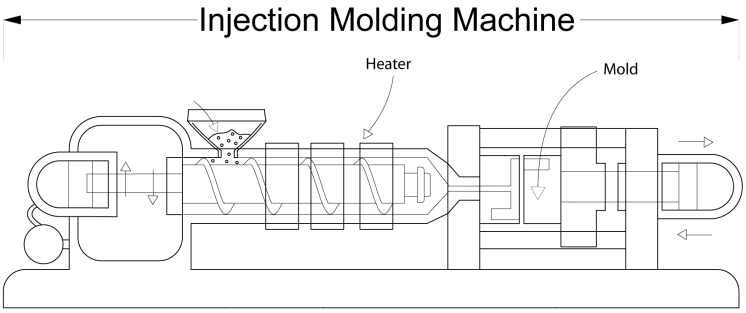


Figure 5.1: Schematic of an injection molding machine
Created by Brendan Rockey, University of Alberta Industrial Design. License: CC BY 3.0

The control program in this simplified example needs to control the following components of the molding machine (see Figure 5.1):

Heater The heater controls the temperature of the molten material. In order to produce faultless parts the temperature needs to be within a specified range.

Mold and Injection Unit The injection unit starts and stops the injection of material. It also controls the position of the mold and the ejection actuator.

5.1.1 Implementation

The components are accessed via the following two interfaces `Heater` and `Injection`: (the actual implementation, which is machine-specific, is not presented here)

Heater.java - Interface for the heater component

```
1 public interface Heater {
2
3   public double getTemperature();
4
5   public void enable();
6   public void disable();
7 }
```

This interface can be used to determine the current temperature of the heater, and to enable and disable the heating component.

HeaterControl.java - Control program for the heater component

```
1  public class HeaterControl extends Coroutine {
2    private final Heater heater;
3
4    public HeaterControl(Heater heater) {
5      this.heater = heater;
6    }
7
8    protected void run() {
9      while (true) {
10       // wait until temperature falls below lower threshold
11       while (heater.getTemperature() >= 75)
12         yield();
13
14       // enable heater
15       heater.enable();
16
17       // wait until temperature is at upper threshold
18       while (heater.getTemperature() < 80)
19         yield();
20
21       // disable heater
22       heater.disable();
23     }
24   }
25 }
```

The coroutine `HeaterControl` is responsible for keeping the heater at the correct temperature. It is implemented as a simple loop that waits for the temperature to drop below 75°C. The heater is then enabled until the temperature reaches 80°C.

Injection.java - Interface for the injection component

```
1  public interface Injection {
2
3    public double getVolume();
4
5    public double getPosition();
6
7    public void setTargetPosition(double pos);
8
9    public void startInjection();
10   public void stopInjection();
11
12   public void ejectPart();
13 }
```

The `Injection` interface is used to determine the total amount (volume) of material that was injected, to determine the position of the movable part of the mold (0 = closed, 100 = open), to set the target position for the mold (the mold will immediately start to move towards the given position), to start and stop the injection of material and to eject the finished part from the machine.

InjectionControl.java - Control program for the injection component

```
1  public class InjectionControl extends Coroutine {
2    private final Injection injection;
3    private final Heater heater;
4
5    public InjectionControl(Injection injection, Heater heater) {
6      this.injection = injection;
7      this.heater = heater;
8    }
9
10   protected void run() {
11     double initialVolume = injection.getVolume();
12
13     // close the mold
14     injection.setTargetPosition(0);
15     while (injection.getPosition() > 0)
16       Coroutine.yield();
17
18     // wait until the heater has reached the minimum temperature
19     while (heater.getTemperature() < 70)
20       Coroutine.yield();
21
22     // start injecting
23     injection.startInjection();
24
25     // wait until 100 units have been injected, then stop
26     while (injection.getVolume() < initialVolume + 100)
27       Coroutine.yield();
28     injection.stopInjection();
29
30     // wait 10 cycles, i.e., 10 ms
31     for (int i = 0; i < 10; i++)
32       Coroutine.yield();
33
34     // open the mold
35     injection.setTargetPosition(100);
36     while(injection.getPosition() < 100)
37       Coroutine.yield();
38
39     // finally: eject the new part
40     injection.ejectPart();
41   }
42 }
```

The coroutine `InjectionControl` performs a more complex series of actions. It first closes the mold, waiting for the movement to be finished. Then it waits for the heater to reach a temperature of at least 70°C, after which it can start the injection process.

As soon as 100 units of material have been injected the injection is stopped. After a cooling time of 10 cycles, i.e., 10 ms, the mold is opened again, and when this is done the finished part is ejected.

5.1.2 Conclusion

This example shows that coroutines can be a very natural and convenient way to write code that performs a complex series of actions. The coroutines in this example also perform non-blocking busy waiting.

Using threads in this context would not be ideal: Real-world systems can have hundreds of components, each of which would need a separate thread. These threads would have to be synchronized in order to pass control from one to the other. This synchronized thread switch is a very expensive operation. Machine control programs normally run at a CPU load of 10-20%, and with hundreds of switches within one 1 ms cycle the switching time will dominate the system performance.

In contrast to threads, the existence of hundreds of coroutines is not a performance problem, because coroutines use less resources than threads and perform switches by an order of magnitude faster than threads. Also, synchronization is unnecessary with coroutines.

As with any coroutine this example could also have been implemented using explicit state machines, but this would have made it much more complex and harder to understand.

This example could also have been implemented even more elegantly if Java had an implementation of lambda expressions [16], which is currently planned for JDK 8 [17]. Instead of:

```
while (heater.getTemperature() >= 75)
  Coroutine.yield();
```

one could simply write:

```
waitUntil(#{heater.getTemperature() < 75});
```

5.2 Session Persistence

Servers that implement user sessions often have to solve the problem of how to deal with a large number of concurrent sessions. For some types of systems there will likely by a large amount of time between subsequent interactions with the user. It may be impractical to keep all sessions in memory until they expire, so there needs to be a way to store these sessions, e.g. to disk or to a database.

The following example shows a simple user interaction that is implemented using coroutines, which can be suspended to disk. Although it is a simplified example, it still demonstrates how the serialization of sessions can be achieved.

A typical interaction with the main program, called `Questions`, looks like this:

```
> java Questions
Please enter your name:

> java Questions "John Doe"
Please enter your country:

> java Questions Genovia
Please enter your age:

> java Questions twenty-eight
Please enter your age:
Please enter a valid number!

> java Questions 28
Please enter your email address:

> java Questions nospam@nospam.at
Name: John Doe, Country: Genovia, Age: 28, Email: nospam@nospam.at
```

While the interaction in this example takes place via command-line parameters, the solution can also be applied to other types of interaction, like web servers.

5.2.1 Implementation

The implementation is divided into three parts: The class that models the user interaction, the main program that drives the serialization and deserialization and helper classes that are needed to serialize the coroutine.

Interaction.java - Implementation of the user interaction

```
public class Interaction implements AsymRunnable<String, String>,
                                    Serializable {

  private AsymCoroutine<? extends String, ? super String> coroutine;

  //helper method used to acquire user input
  private String readString(String message) {
    return coroutine.ret(message);
  }

  //helper method used to acquire numerical user input
  private int readNumber(String message) {
    int tries = 0;
    while (true) {
      try {
        return Integer.valueOf(readString(message));
      } catch (NumberFormatException e) {
        if (tries++ == 0)
          message += "\nPlease enter a valid number!";
      }
    }
  }

  @Override
  public String run(
```

```
26            AsymCoroutine<? extends String, ? super String> coroutine,
27            String value) {
28
29      this.coroutine = coroutine;
30
31      String name = readString("Please␣enter␣your␣name:␣");
32      String country = readString("Please␣enter␣your␣country:");
33      int age = readNumber("Please␣enter␣your␣age:");
34
35      if (age < 70) {
36        String email;
37        email = readString("Please␣enter␣your␣email␣address:");
38        return "Name:␣" + name + ",␣Country:␣" + country + ",␣Age:␣" +
39               age + ",␣Email:␣" + email;
40      } else {
41        String telephone = readString("Please␣enter␣your␣telephone␣number:");
42        return "Name:␣" + name + ",␣Country:␣" + country + ",␣Age:␣" +
43               age + ",␣Telephone:␣" + telephone;
44      }
45    }
46  }
```

This is the only class a programmer would have to implement within a framework that supports the implementation of sessions via coroutines. It is programmed as if the user input was available immediately any time the user is presented with a question.

The user input is forwarded to the asymmetric coroutine via its input value, and the console output is passed to the main program via the coroutine's output value.

The `readString` method would, in a more sophisticated version, employ some form of non-blocking i/o. The coroutine would then be restored whenever user input is available.

It is important to note that local variables are available again when the coroutine is restored and that coroutines can be suspended from within subsequently called methods (like `readNumber` and `readString`).

Questions.java - Main program containing the serialization / deserialization logic

```
1  public class Questions {
2
3    public static void main(String[] args) {
4
5      AsymCoroutine<String, String> coro;
6
7      // check if there is a suspended coroutine on disk
8      if (new File("data.bin").exists()) {
9        FileInputStream fis = new FileInputStream("data.bin");
10       CoroutineInputStream<String, String> cis;
11       cis = new CoroutineInputStream<String, String>(fis);
12       coro = cis.readAsymCoroutine();
13       cis.close();
14     } else {
15       coro = new AsymCoroutine<String, String>(new Interaction());
16     }
17
```

```
18      // transfer control, this will execute the next step
19      String message = coro.call(args.length > 0 ? args[0] : null);
20      System.out.println(message);
21
22      // if the coroutine isn't finished: store to disk
23      if (coro.isFinished()) {
24        new File("data.bin").delete();
25      } else {
26        FileOutputStream fos = new FileOutputStream("data.bin");
27        CoroutineOutputStream<String, String> cos;
28        cos = new CoroutineOutputStream<String, String>(fos);
29        cos.writeAsymCoroutine(coro);
30        cos.close();
31      }
32    }
33  }
```

This main program contains the main serialization / deserialization logic.

It checks whether there is a suspended coroutine on disk (stored in a file called `data.bin`), and if there is one it restores it using the coroutine deserialization mechanism. If there is no suspended coroutine then a new one will be created.

The main program then transfers control to the asymmetric coroutine, which will perform one step in the user interaction.

Finally, if the coroutine (and thus, the user interaction) is not finished yet it is serialized to disk. Otherwise, the data file is deleted.

CoroutineOutputStream.java - Object output stream with special handling of coroutines

```
1  public class CoroutineOutputStream<I, O> extends ObjectOutputStream {
2
3    private AsymCoroutine<I, O> coro;
4
5    public CoroutineOutputStream(OutputStream out) {
6      super(out);
7      enableReplaceObject(true);
8    }
9
10   @Override
11   protected Object replaceObject(Object obj) {
12     if (obj instanceof Method) {
13       return new SerializableMethod((Method) obj);
14     } else if (obj == coro) {
15       return new CoroutineDummy();
16     }
17     return super.replaceObject(obj);
18   }
19
20   public void writeAsymCoroutine(AsymCoroutine<I, O> coro) {
21     this.coro = coro;
22     writeObject(coro.serialize());
23   }
24 }
```

The special object output stream `CoroutineOutputStream` is needed for two reasons: It needs to replace instances of `java.lang.reflect.Method`, which are not serializable, with instances of the `SerializableMethod` class, and it needs to replace all instances of the coroutine in question with an instance of the `CoroutineDummy` class (because coroutine objects themselves are also not serializable).

CoroutineDummy.java - Helper class used to represent serialized coroutine objects

```
1 public class CoroutineDummy implements Serializable {
2 }
```

The helper class `CoroutineDummy` is needed because only the `CoroutineFrame` objects are serializable, and not the `Coroutine` and `AsymCoroutine` objects themselves.

SerializableMethod.java - Helper class to replace instances of `java.lang.reflect.Method`

```
1  public class SerializableMethod implements Serializable {
2
3    private final Class<?> clazz;
4    private final String name;
5    private final Class<?>[] parameters;
6
7    public SerializableMethod(Method method) {
8      this.clazz = method.getDeclaringClass();
9      this.name = method.getName();
10     this.parameters = method.getParameterTypes();
11   }
12
13   public Method getMethod() {
14     return clazz.getDeclaredMethod(name, parameters);
15   }
16 }
```

The serializable class `SerializableMethod` is used by the coroutine object output stream to store references to instances of `java.lang.reflect.Method`. This implementation might not be enough for complex use cases, but it is a simple workaround for the fact that the reflective method class is not serializable.

CoroutineInputStream.java - Object input stream with special handling of coroutines

```
1  public class CoroutineInputStream<I, O> extends ObjectInputStream {
2
3    private AsymCoroutine<I, O> coro;
4
5    public CoroutineInputStream(InputStream in) {
6      super(in);
7      enableResolveObject(true);
8    }
9
10   @Override
11   protected Object resolveObject(Object obj) {
12     if (obj instanceof SerializableMethod) {
13       return ((SerializableMethod) obj).getMethod();
14     } else if (obj instanceof CoroutineDummy) {
15       return coro;
```

```
16      }
17      return super.resolveObject(obj);
18    }
19
20    public AsymCoroutine<I, O> readAsymCoroutine() {
21      coro = new AsymCoroutine<I, O>();
22      CoroutineFrame[] frames = (CoroutineFrame[]) readObject();
23      coro.deserialize(frames);
24      return coro;
25    }
26  }
```

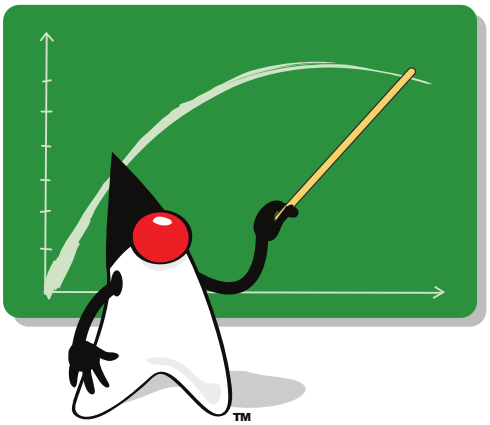
The special object input stream `CoroutineInputStream` undoes the replacements performed by the coroutine object output stream: It transforms instances of `SerializableMethod` back into reflective method objects and replaces `CoroutineDummy` instances with a new coroutine object.

5.2.2 Conclusion

Being able to serialize coroutines is a vital feature for numerous applications such as web frameworks or other systems with long-lasting user sessions. At the same time it will present the implementors of frameworks that are built on top of a coroutine system with a number of challenges: How should non-serializable classes (like `java.lang.Method`) be dealt with, and how deep should the object graph be serialized? The replacement feature of Java object serialization streams is vital in solving these problems.

Chapter 6

Evaluation



This chapter contains a number of evaluations, benchmarks and cost estimations of the different coroutine features, with respect to run time and memory usage. All tests and benchmarks presented in this section were performed on a personal computer with a quad-core Intel i5 750 CPU at 2.67 GHz and 8 GB main memory running Windows 7.

6.1 Memory Consumption

Coroutines that are stored on the heap 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 typically uses no more than 2 kB of memory (roughly 1 kB 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 stored stack frames. 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 kB, depending on the operating system. This leads to a minimum stack size of 32 kB on Linux, 64 kB on Windows and 100 kB on Solaris, which allows the allocation of roughly 60,000, 30,000, or 20,000 stacks on a 32-bit machine.

6.2 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. The time it takes to do a context switch between threads is hard to measure, but this operation is much more expensive and takes at least 1,000 ns [35].

Figure 6.1 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 top graph is a magnification of the bottom 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:

- From 2 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 copying the stack frames between the stacks and the save areas on the heap.

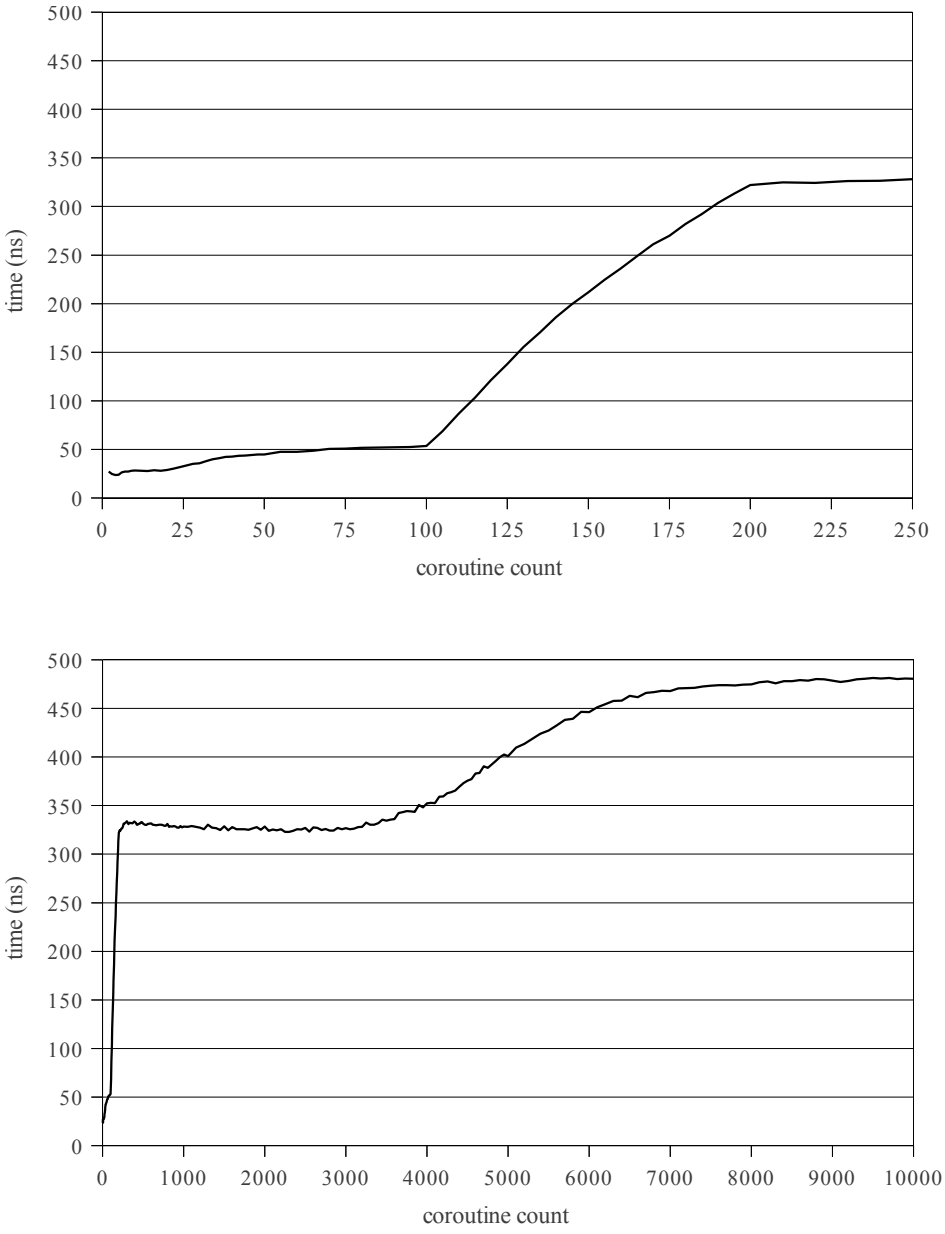


Figure 6.1: Average time for one "switch" operation, relative to the active coroutine count. The top graph is a magnification of the leftmost part of the bottom graph.

- 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

6.3 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 evaluated the performance of different Ruby [11] 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 of Ruby as well as programming frameworks such as Ruby on Rails [33] have made the language very popular, consistently ranking in the top 15 of the TIOBE Programming Community Index [34] 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 [32] 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, where the stack contents are copied to and from the stack at each context switch.

JRuby [31] 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.

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 6.2 and 6.3 show a comparison of the performance of MRI, JRuby without modifications, and JRuby with coroutine support. Figure 6.2 shows the runtime of a benchmark that passes different numbers of messages through a chain of 5 fibers, and Figure 6.3 shows the same benchmark for a chain of 5000 fibers.

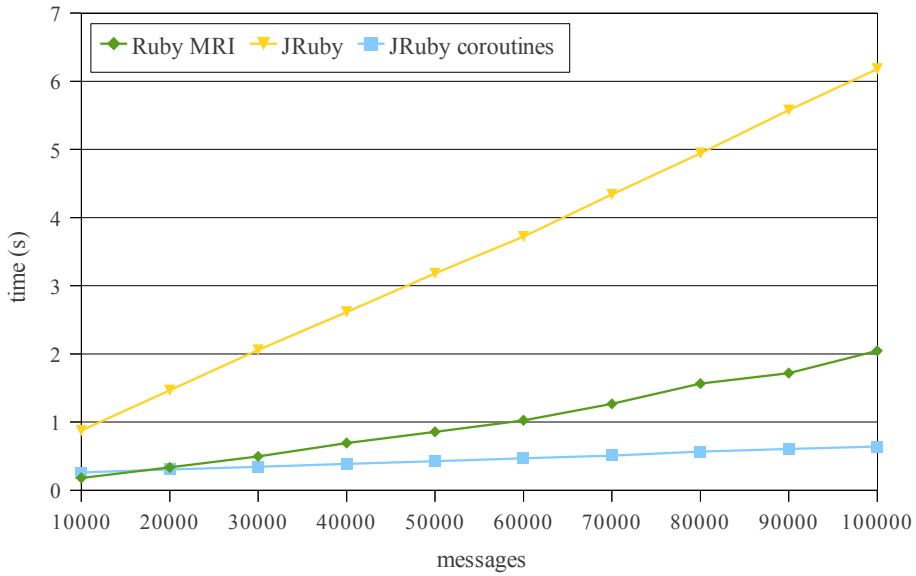


Figure 6.2: Ruby benchmark: 5 fiber chain

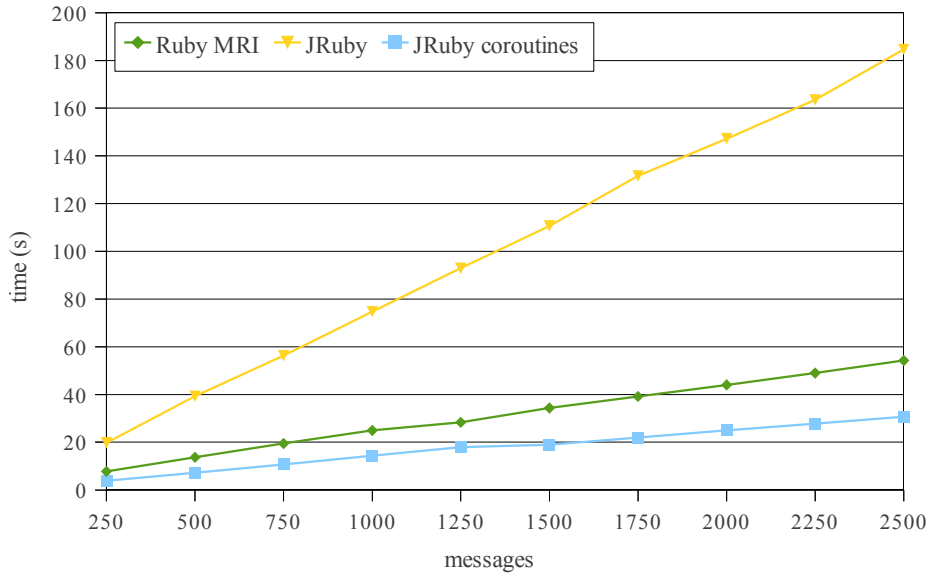


Figure 6.3: Ruby benchmark: 5000 fiber chain

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.

6.4 Thread-to-Thread migration Performance

Thread-to-thread migration is an expensive operation because it requires the system to walk over all the stack frames of a coroutine. This is done in order to make sure that there are no locks and native frames, and also to adjust stack and thread pointers within the stack frames.

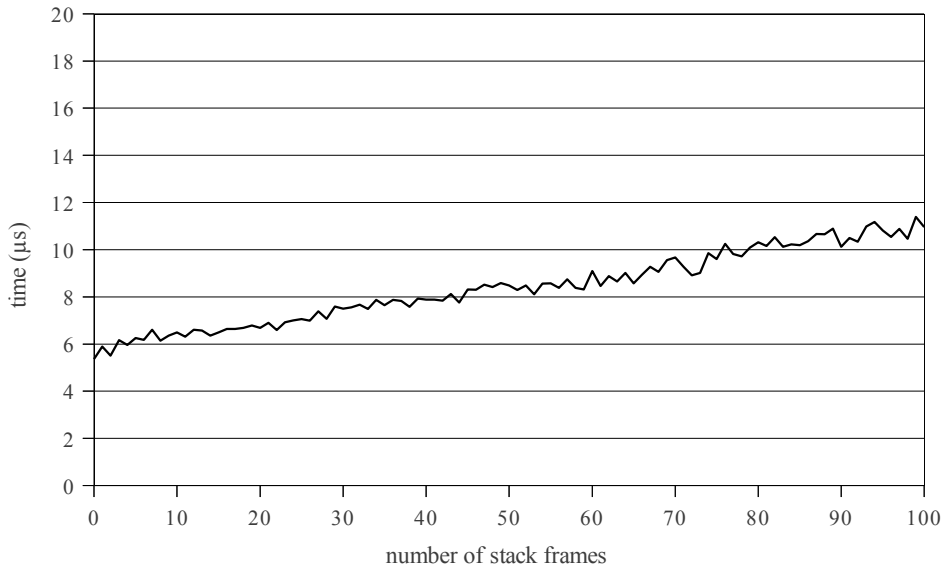


Figure 6.4: Time per migration operation, depending on the number of stack frames.

The time one migration to another thread requires is shown in Figure 6.4. The graph shows the timing depending on the number of stack frames of the coroutine. One migration operation takes approximately 6 + (number of stack frames) $\cdot$ 0.05 μs.

These numbers were acquired by migrating a coroutine back and forth between two threads one hundred times and measuring the total time, which means that they also include a small synchronization overhead (roughly 1%). The two threads are running on different CPU cores, and synchronization is achieved via simple blocking spin locks.

6.5 Serialiation / Deserialization Performance

The run time of serialization and deserialization is less of an issue, since the target where the serialized coroutine is ultimately stored is likely a slow storage medium like a disk or a database.

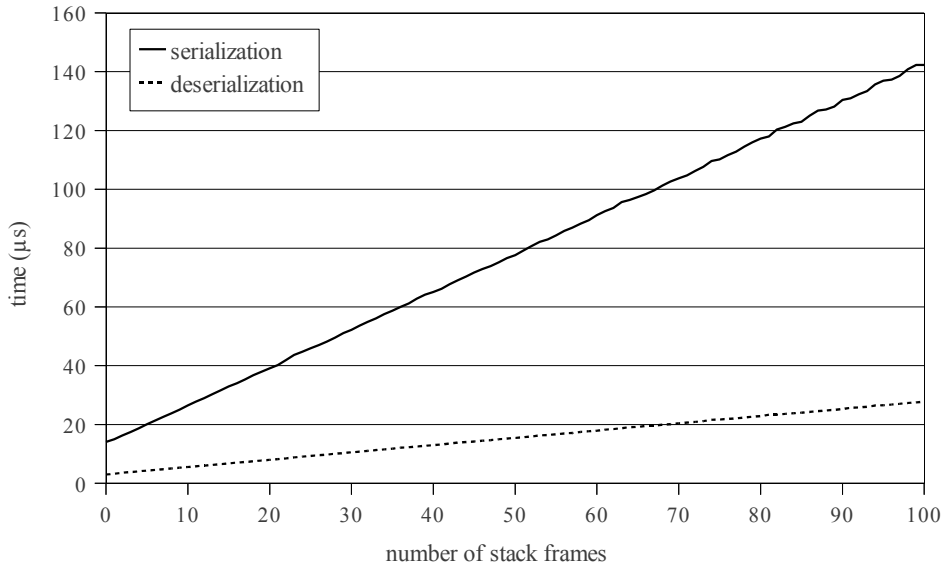


Figure 6.5: Time per serialization and deserialization operation, depending on the number of stack frames.

Figure 6.5 shows the timings of the serialization and deserialization operations. Serialization uses the stack walking code that is normally only used for debugging and deoptimization. This code is not optimized towards fast execution, and thus serialization is very expensive compared to other coroutine operations.

Serialization takes approximately 17 + (number of stack frames) $\cdot$ 1.3 μs, and deserialization takes approximately 3.5 + (number of stack frames) $\cdot$ 0.25 μs.

Chapter 7

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. [36] 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 and implementation in the Java environment.

Ierusalimschy et al. [15] 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 works in a *stackless* way. 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 [5] show an implementation of coroutines that uses a "spaghetti stack" containing stack frames of multiple coroutines combined into a single stack. Every time a subroutine is called the system has to check if there is enough space for a new stack frame, because some other coroutine might occupy the area where the stack frame would normally be placed. While this approach can be efficient, it suffers from a number of drawbacks: It requires extensive modifications within the compiler and the runtime system, performs costly range checks for each method call and violates the assumption made by modern operating systems that there is always a certain amount of available stack space beneath the current stack frame.

Pauli and Soffa [24] 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 [20] 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 trade-off between CPU cache efficiency (small heap) and fewer collections (large heap).

Carroll [6] 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.

Second Life, developed by Linden Lab, makes heavy use of the concept of portable agents. These portable agent scripts [8], which implement the behavior of virtual objects, are essentially coroutines that can be transfered to other VMs whenever the virtual object passes the boundary of a simulation server (called *sim*). Since August 2009 Second Life executes these agent scripts on Mono, an open source implementation of the Microsoft .NET platform. In order to be able to suspend coroutines and move running scripts from server to server, the scripts are instrumented in such a way that they can store and restore their current state, similar to the javaflow library [3]. These transformation-based coroutines are at least an order of magnitude slower than a native coroutine or continuation implementation [30].

Chapter 8

Future Work

This thesis presented a basic design and implementation of symmetric and asymmetric coroutines. It is powerful enough to be used in a large number of application domains such as simulation, web servers, or XML parsing. However, our implementation could be enhanced with additional features that would make coroutines even more powerful, efficient and safe. Suggestions for future work are:

Lazy Copying

When copying to and from the stack it might be beneficial to copy only parts of the stack and use an approach similar to [30] to copy the rest on demand. However, it remains to be seen if this leads to noticeable performance gains as it only applies to systems with very large numbers of coroutines.

Scheduling Interface

The implemented scheduling of coroutines is suitable for a large number of use cases, but an interface that allows the customization of the scheduling behavior might be necessary for advanced uses.

Input/Output Value in Registers

Right now, the input and output values of asymmetric coroutines are stored in fields of the `AsymCoroutine` class. A possible optimization would be to pass these values in processor registers during the coroutine context switch, which would make the switch operation faster. This is possible on 64 bit platforms, but seems impractical on 32 bit platforms due to the smaller number of available registers.

Allow Locks in Migration and Serialization / Deserialization

Although it would likely make the implementation much more difficult, it might be possible to derive semantics for the transfer and serialization of locks.

Specialized Stack Walking Code for Serialization

Serialization at the moment relies heavily on the stack walking code that is pri-

marily intended for debugging and deoptimization. A special version of the stack walking code for coroutine serialization would likely make this operation much faster.

Chapter 9

Conclusions

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

This thesis presented a coroutine implementation approach that allocates stacks up to a certain limit and then starts copying data to and from these stacks. It showed the implementation of this approach for the widely used Java HotSpot™ 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.

The additional thread-to-thread migration and serialization mechanisms make the implementation even more powerful. These features are required for a number of advanced use cases.

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.

Acknowledgements

First and foremost, I would like to thank my advisor Hanspeter Mössenböck for his constant support and the thorough comments on my algorithms, my papers and this thesis.

I would also like to thank my current and former colleagues in the Sun/Oracle project, especially Thomas Würthinger and Christian Wimmer, for countless discussions and for all the feedback on my work.

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

The images of duke, the Java mascot, used in this thesis were provided by Oracle under the BSD license and can be retrieved from `http://duke.kenai.com/`.

List of Figures

Listings

Bibliography

[1] Adams, IV, N. I., 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. `ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/standards/r5rs-html.tar.gz`.

[2] Adya, Atul, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur: *Cooperative task management without manual stack management*. In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, Jun 2002. USENIX Association.

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

[4] Arnold, Ken, James Gosling, and David (David Colin) Holmes: *The Java^{TM} Programming Language*. Addison-Wesley, pub-AW:adr, fourth edition, 2005, ISBN 0-321-34980-6.

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

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

[7] Conway, Melvin E.: *Design of a separable transition-diagram compiler*. Communications of the ACM, 6(7):396–408, 1963.

[8] Cox, Robert and Patricia Crowther: *A review of linden scripting language and its role in second life*. In Purvis, Maryam and Bastin Savarimuthu (editors): *Computer-Mediated Social Networking*, volume 5322 of *Lecture Notes in Computer Science*, pages 35–47. Springer Berlin / Heidelberg, 2009.

[9] Dahl, Ole Johan, Bjorn Myrhaug, and Kristen Nygaard: *SIMULA 67. common base language.* Technical report Publ. No. S-2, Norwegian Computing Center, Oslo, Norway, May 1968. Revised Edition: Publication No. S-22.

[10] Dragoş, Iulian, Antonio Cunei, and Jan 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.

[11] Flanagan, David and Yukihiro Matsumoto: *The Ruby programming language.* O'Reilly, 2008, ISBN 978-0-59651-617-8.

[12] Goldberg, Adele and David Robson: *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[13] Google: *Go programming language*, 2009. `http://golang.org/`.

[14] Griswold, Ralph E. and Madge T. Griswold: *The Icon programming language.* Peer-to-Peer Communications, San Jose, CA, USA, third edition, 1997, ISBN 1-57398-001-3. `http://www.cs.arizona.edu/icon/lb3.htm`.

[15] Ierusalimschy, Roberto, Luiz Henrique de Figueiredo, and Waldemar Celes: *The implementation of Lua 5.0.* Journal of Universal Computer Science, 11(7):1159–1176, 2005.

[16] Java Community Process: *JSR 335: Lambda Expressions for the Java™ Programming Language*, 2010. `http://jcp.org/en/jsr/detail?id=335`.

[17] Java Community Process: *JSR 337: Java™ SE 8 Release Contents*, 2010. `http://jcp.org/en/jsr/detail?id=337`.

[18] Kotzmann, Thomas, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox: *Design of the Java HotSpot™ client compiler for Java 6.* ACM Transactions on Architecture and Code Optimization, 5(1):Article 7, 2008.

[19] Lindholm, Tim and Frank Yellin: *The Java™ Virtual Machine Specification.* Addison-Wesley, 2nd edition, 1999.

[20] Mateu, Luis: *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.

[21] Moura, Ana Lúcia De and Roberto Ierusalimschy: *Revisiting coroutines.* ACM Transactions on Programming Languages and Systems, 31(2):6:1–6:31, February 2009.

[22] Murray-Rust, Peter, David Megginson, Tim Bray, *et al.*: *Simple API for XML*, 2004. `http://www.saxproject.org/`.

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

[24] Pauli, W. and Mary Lou Soffa: *Coroutine behaviour and implementation.* Software—Practice and Experience, 10(3):189–204, March 1980.

[25] Prähofer, Herbert, Dominik Hurnaus, Roland Schatz, and Hanspeter Mössenböck: *The domain-specific language monaco and its visual interactive programming environment.* In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 104–110, 2007.

[26] Python Software Foundation: *Greenlet - Lightweight in-process concurrent programming*, 2010. `http://pypi.python.org/pypi/greenlet`.

[27] Russell, Kenneth and David Detlefs: *Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing.* SIGPLAN Not., 41:263–272, October 2006, ISSN 0362-1340. `http://doi.acm.org/10.1145/1167515.1167496`.

[28] Shankar, Ajai: *Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API*, 2005. `http://msdn.microsoft.com/en-us/magazine/cc164086.aspx`.

[29] Simon, Doug, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White: *Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine.* In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM, ISBN 1-59593-332-8. `http://doi.acm.org/10.1145/1134760.1134773`.

[30] Stadler, Lukas, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John 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.

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

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

[33] Thomas, Dave, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz: *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006, ISBN 0977616630.

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

[35] Tsuna's blog: *How long does it take to make a context switch?*, 2010. `http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html`.

[36] Weatherly, R. M. 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.

[37] Wirth, Nikolaus: *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 4th edition, 1988, ISBN 0-387-50150-9.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 15. Februar 2011

Lukas Stadler, BSc.