Thread Migration and Checkpointing in Java

Technical Report tr-ri-10-315
June 2010

Joachim Gehweiler and Michael Thies

Heinz Nixdorf Institute, University of Paderborn, Germany
{joge,mthies}@uni-paderborn.de

Abstract. We present PadMig, a library for thread migration and checkpointing in Java. Its language specification is based on standard Java 5 annotations such that no additional keywords are used, which eases the developers' work as they can keep using their favorite IDE and can produce both a migratable and a non-migratable version of their program out of the same source code. Java source code with PadMig specific annotations is transformed into migratable Java code by the PadMig source-to-source compiler *migc*.
In this paper we present the PadMig Language Specification and its production quality compiler. We focus especially on the implementation of the unfolding technique employed by the compiler, and give some evaluation concerning code growth and migration overhead.

1 Introduction

In distributed, volunteer-based computing environments, such as our web computing library PUB-Web [1, 12], it is necessary to migrate threads of lengthy calculations at runtime to other hosts in order to meet execution deadlines because the donated computing power continually fluctuates and hosts may even become unavailable. Furthermore, it is desirable to regularly create checkpoints of the execution state, so that a certain state of a thread can be restored in case of a system crash rather than restarting all the calculations from the beginning.

There are three ways to migrate threads in Java: modification of the Java Virtual Machine (VM) [6], bytecode transformations [13, 17], and sourcecode transformations [14, 4]. Modifying the Java VM is out of the question because everybody would have to replace his installation of the original Java VM with one from a third party, just to run a migratable Java program. Approaches of this kind do not only have limited success due to their installation overhead, but also because of trust matters: people would need to trust that a third party VM does not have any security defects. Additionally, from the developers' point of view, this approach would result in a lot of maintenance work to adapt all future releases and updates of Sun's VM. An obvious alternative to modifying Sun's VM is of course to develop an own VM, but this results in even more implementation work. A quite well-known approach of this kind is the Jikes Research VM,

which provides—among other features—thread migration techniques [2]; but although a lot of man-power has been spent into this project, it is not suitable for production use.

The bytecode transformation approach is also less suitable in our case because we would need to re-synthesize high-level constructs such as loops or `try-catch-finally` blocks for our translation approach. Additionally, a bytecode transformer should be able to deal with all possible bytecode constructs, not only those found in well-shaped *javac* output, which means additional effort for the development of such a compiler.

Thus, this paper deals with the sourcecode transformation approach. There are two ways to accomplish this: using code unfolding [14] or an artificial program counter [4]. Using the former approach, nested loops and branches have to be unfolded, whereas additional code fragments have to be inserted for each statement to check whether or not the statement has to be skipped in the latter approach. Obviously, unfolding needs only be done if there are migratory sub-statements; similarly, successive statements to be skipped can be grouped in case there are no migratory statements in between.

During the development of our web computing library PUB-Web, we first employed a very promising prototype implementation [5] of the unfolding technique, called *JavaGo*, which extends the Java programming language with three keywords: migrations are performed using the keyword `go` (passing a filename instead of a hostname as parameter creates a backup copy of the execution state). All methods, inside which a migration may take place, have to be declared `migratory`. The depth, up to which the call stack will be migrated, can be bounded using the `undock` statement.

The translation of this extended language into Java sourcecode is done using the JavaGo compiler *jgoc*. Migratable programs have a special `main` method and are launched via a wrapper class. In order to continue the execution of a migratable program, an instance of a migration server has to run on the destination host.

But, unfortunately, this prototype implementation was not fully compatible with the Java RMI standard and could also not be extended to support Java 5 due to design issues. Thus and because of the following two more reasons, we decided to start from scratch with our own implementation, the *Paderborn Thread Migration and Checkpointing (PadMig)* library [8]:

- Using annotations instead of additional keywords, we can stick to the Java standard instead of deriving a new programming language. As a side effect, this allows developers to keep using their favorite IDEs without any drawbacks. Furthermore, we are able to design PadMig such that developers do not need to maintain two versions if they like to have a non-migratable and a migratable version of their code — they can simply skip the intermediate compilation step with our *migc* compiler to obtain a non-migratable version of their code.
- The prototype implementation of the JavaGo compiler was more or less a quick-hack written in Objective Caml [3], which was terribly slow, did not

produce useful error messages, was hard to debug and not available for all important platforms. For portability reasons we based our new implementation solely on tools in Java; in particular, we use the Java transformation framework Spoon [15, 10, 7], which provides a complete model of the abstract syntax tree where any element can be accessed both for reading and modification.

The PadMig API consists of special functions to initiate a migration to another machine or to save a checkpoint into a file. All methods, inside which a migration can occur or a checkpoint is created, need to be annotated. The PadMig compiler then transforms this code into migratable code. As our implementation does not modify the Java language, the original, non-migratable code is fully functional Java code, which just produces a warning rather than actually migrating.

The remainder of this paper is organized as follows: in Section 2 we describe the language specification of PadMig; in the following sections we provide the technical background and give insight into the translation concepts; finally, we evaluate our technique in Section 5.

2 The PadMig Language Specification

In order to migrate the calling thread or to create a checkpoint, migration points have to be inserted into a program. Only there — at statements consisting of a call to the `migrate()` or `checkpoint()` method of the library class `padmig.PadMig` — the calling thread is migrated to a remote host or a checkpoint of the calling thread is created, respectively. In particular, the two methods have the following signatures:

```
public static void migrate(
    java.net.URL migrationServer)
    throws padmig.MigrationException;
public static void checkpoint(
    java.io.File backupFile)
    throws padmig.MigrationException;
```

When a checkpoint is generated, the execution continues locally. In case the migration or checkpointing fails, a `MigrationException` will be thrown.

In order to allow migrations or checkpoints inside a method, either directly by calling the `migrate()` or `checkpoint()` method, or transitively by calling some other migratable method, the particular method has to be annotated with `padmig.Migratory`. In some cases, it is desirable to migrate only a part of the call stack; in PUB-Web, for example, only the user program, but not the PUB-Web VM should be migated. The method, which forms the bottom element of the call stack to be migrated has to be annotated with `padmig.Undock` instead of `Migratory` (cf. Fig. 1). According integrity checks are performed at compile time: *migc* stops with an error if a migratable method is called from an ordinary method, i.e., a method which is neither migratable nor undockable. *migc*

also ensures that migratory annotations are consistent when inheriting from a (possibly abstract) class or when implementing or inheriting interfaces.

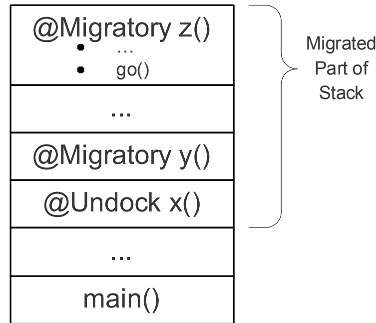

Fig. 1. Illustration of the migratory part of a call stack.

We distinguish two kinds of migrations: if a method annotated with `Undock` has a return value, we call migrations on this call stack *synchronous* as the local execution has to wait for the undocked method to return; otherwise, if the `Undock` method has no return value, we call migrations on that call stack *asynchronous* because the local execution can already continue with the next statement after the call to the `Undock` method directly after the `Undock` method has migrated.

As all local variables are migrated by default, it is necessary to mark the locals which cannot be migrated (because they are not serializable) or should not be migrated (as they generate an unnecessary overhead). This can be done using the `padmig.DontMigrate` Annotation. The PadMig compiler does not explicitly check for all possible kinds of side effects; however, the most prominent issues, such as open files or sockets, are often implicitly detected, e.g., because file or socket handles are not serializable. Note: In order to support, e.g., open socket connections in a transparent way with respect to inheritance etc., it is necessary to adapt the runtime environment appropriately by rewriting the according classes using proxies, which is out of the scope of this work. User programs inside PUB-Web are only allowed to open files for read access or to write output back to the user's peer via the PUB-Web API. Such open files or socket connections are not migrated but reside in the PUB-Web VM; after a migration, the user program needs to re-open files for read access or can continue to write output to the user's peer via the PUB-Web API.

The object whose method is to be migrated has to be serializable of course, i.e., it must implement the `java.io.Serializable` interface. The main class of an application is also required to implement a special main method defined in the `padmig.Migratable` interface:

```
public java.io.Serializable migratableMain(
    java.io.Serializable[] args);
```

For interoperability reasons (see next chapter), the parameters and return value of the main method can be any serializable object. Listing 1.1 shows an example for a migratable program, which demonstrates all the languages features.

Listing 1.1. Example for a migratable program.

```
1  import java.io.*;
2  import java.net.*;
3  import padmig.*;
4
5  public class Example implements Migratable, Serializable
     {
6
7    public boolean migrationNecessary() {
8      // evaluate situation here ...
9      return true;
10   }
11
12   public URL getMigrationTarget() {
13     try {
14       return new URL("pp://some.host:1234/
           migration_server_name");
15     } catch (MalformedURLException mue) { /* ... */ }
16   }
17
18   @Migratory
19   public int someMethod(int n) throws MigrationException
       {
20     for (int i=n; i>=0; i--) {
21       // some complicated calculation here
22       @DontMigrate
23       File someFileHandle;
24       // complicated calculation continued
25       if (i % 10 == 0) {
26         PadMig.checkpoint(new File("/path/checkpoint-"+n
             +".bak"));
27       }
28       if (migrationNecessary()) {
29         PadMig.migrate(getMigrationTarget());
30       }
31     }
32   }
33
34   @Undock
35   public int syncUndockMethod(int n) throws
       MigrationException {
36     return 2 * someMethod(n);
37   }
38
```

```
39    @Undock
40    public void asyncUndockMethod(int n) throws
        MigrationException {
41      // remote result output
42      System.out.println("the result is " + (2 *
          someMethod(n)));
43    }
44
45    public Serializable migratableMain(Serializable[] args
        ) {
46      try {
47        asyncUndockMethod(21);
48        System.out.println("thread has undocked");
49        // local result output
50        System.out.println("the result is " +
            syncUndockMethod(23));
51      } catch (MigrationException e) {
52        System.out.println("migration has failed");
53        e.printStackTrace();
54      }
55      return null;
56    }
57  }
```

3 Technical Background

In this section we give technical insight in how the PadMig compiler transforms the annotated Java code into migratable standard Java code.

When migrating the calling thread, we need to save its current call stack and an abstract representation of its program counter, transfer both to the remote side, reconstruct the stack, and jump back into the code to the equivalent position. To provide the functionality of a program counter, the PadMig compiler surrounds the original body of a migratory method with a switch statement, whose cases are used as entry points to reenter the code after a migration; thus, before every migratory invocation, the code generated by PadMig increases program counter and the migratory invocation is put into a new case statement (cf. Listing 1.2). If migratory expressions occur in loops or conditionals, an unfolding technique needs to be applied, as described in detail in Section 4.

For each migratory method a method-specific subclass of `padmig.lib.StackFrame` is generated, which stores all migratory locals and a symbolic program counter. When the current call stack is to be saved, it is represented as a list of instances of these stack frame classes. This list is created on demand by throwing a special `java.lang.Throwable`, namely `padmig.lib.SaveStack`, which holds the growing saved stack and has to be caught by every method and passed on after adding its own stack frame. To restore the call stack on the remote side, the corresponding stack frame is passed to each method on the stack

via a non-`null` additional parameter called `__parentState` (this parameter is `null` during ordinary method invocations). The locals as well as the program counter are then restored by generated code inserted at the beginning of each method. A simplified example is provided in Listing 1.2.

Listing 1.2. Simplified example of a generated method.

```
public void foo(String param1, StackFrame __parentState)
    throws SaveStack, MigrationException {
  FooStackFrame __state = null;
  int __entryPoint = 0;
  // locals declarations here...
  if (__parentState != null) {
    __state = ((FooStackFrame)(__parentState.child));
    __entryPoint = __state.entryPoint;
    // restore locals here...
  }
  try {
    switch (__entryPoint) {
    case 0 :
      // original method body before migration here...
      __entryPoint = 1;
      throw new DoMigration(getMigrationDst());
    case 1 :
      // original method body after migration here...
      // ...
      // original method body before checkpointing here
        ...
      __entryPoint = 2;
      throw new DoCheckpoint(getBackupFile());
    case 2 :
      // original method body after checkpointing here
        ...
    }
  } catch (SaveStack __stack) {
    __state = new FooStackFrame();
    __state.self = this;
    __state.entryPoint = __entryPoint;
    // save locals here...
    __state.child = __stack.bottomOfStack;
    __stack.bottomOfStack = __state;
    throw __stack;
  }
}
```

Once you have compiled both your migratable program with the PadMig compiler *migc* and and the resulting code with the Java compiler *javac*, you are ready to run your migratable program — either standalone or as part of another application.

In order to execute a migratable program stand-alone, you need to run a migration server `padmig.standalone.Server` on every possible migration target and start the migratable program via `padmig.standalone.Client`, which is a wrapper around the migratable main method. As the internal communication of PadMig is performed via the *Paderborn Remote Method Invocation (PadRMI)* library [9], the library `padrmi.jar` has to be included in the classpath. Java properties specify the IP address, port, etc. of the local machine and the codebase: either on a web server or file server as a `http:` or `file:` protocol URL or a `pp:` protocol URL pointing to one of the migration servers. Note that multiple cooperating migration servers must all point to the same codebase. Handlers for the PadRMI protocol (denoted by `pp:` in URLs) have to be installed into all participating JVMs as well via the Java Protocol Handler mechanism.

Instead of running your migratable application standalone, you can also integrate it into other Java applications — like we integrated it into PUB-Web — using the PadMig interoperability interface. In order to enable your application to accept incoming thread migrations, you need to start the PadRMI daemon and register a `padmig.iop.Service`, for example `padmig.iop.DefaultServiceImpl`:

```
padrmi.Server.startDefaultServer();
padrmi.Server.getDefaultServer().addObject(
    padmig.lib.PadMigLib.PADRMI_SERVICE_NAME,
    new padmig.iop.DefaultServiceImpl(),
    padmig.iop.Service.class, null, null);
```

Furthermore, you will probably want to implement the `padmig.iop.MigrationListener` interface and register it with the `DefaultServiceImpl.addMigrationListener()` method in order to obtain references to incoming migratable objects and to be notified about migration failures; the code snippet in Listing 1.3 illustrates this.

Listing 1.3. Example for a migration listener implementation.

```
1 public void migratableObjectArrived(MigrationEvent e) {
2   System.out.println("incoming migration associated with
       object " + e.getObject());
3 }
4
5 public void migratableObjectContinuationFailed(
    MigrationEvent e) {
6   System.out.println("migration associated with object "
       + e.getObject() + " failed:");
7   e.getError().printStackTrace();
8 }
```

Supposed you have correctly set the PadRMI related properties like in the standalone case and your migratable program `HelloWorld` is located at the path specified in the `padrmi.path` property, you can start your program from an enclosing Java application like this:

```
Object returnValue = padmig.Launcher.launch(new URL(
    padrmi.Server.getDefaultServer().getURL() + "/"),
    HelloWorld.class.getName(),
    new Serializable[] { "hi", "there!" });
```

4 Translation Concepts

Before translating migratory methods, the PadMig compiler first checks if the provided code is syntactically correct Java code and obtains its abstract syntax tree using the Java transformation framework Spoon [15, 10, 7]. Then it performs some integrity checks on the code, in particular if every class (or interface) containing migratory methods implements (or extends, respectively) the `java.io.Serializable` interface, and if none of the reserved variable names `__state`, `__tmpState`, `__parentState`, `__entryPoint`, `__stack`, `__t`, `__gen`, `__tryNestingDepth`, `__cFlowBreakLevel` is used inside migratory methods. When overriding methods, either all or none of them can be migratory due to the additional stack frame parameter; thus the PadMig compiler verifies that a `@Migratory` or `@Undock` annotation of a method is compatible with all possibly existing overridden methods. Finally, it ensures that there are no migratory methods inside anonymous classes because stack frame containers can only be created for named classes.

Then every migratory method is translated (other code passes through unchanged). First, the signature of the method is changed in order to allow the special throwable `padmig.lib.SaveStack` to be thrown on a migration, and to get the transmitted call stack passed in as a parameter on the remote side. In particular, a parameter `__parentState` of type `padmig.lib.StackFrame` is added to the parameter list and `padmig.lib.SaveStack` to the throws clause.

Before explaining the actual translation of a migratory method body in detail, we have to regard two special cases: First, if the method to be translated is an `@Undock` method, a helper method with the original signature is additionally required, which invokes the translated migratory method and handles the `padmig.lib.SaveStack` throwable (see listings 1.4 and 1.5 for an example). Second, if the method to be translated is abstract or declared inside an interface, or if the body of an ordinary `@Migratory` method contains no migratory invocation, no further processing of the method body is required.

Next, an inner class extending `padmig.lib.StackFrame` is created for each migratory method, which contains a field for each parameter of the method and for each local variable that is not excluded from migration (see Listing 1.5).

Each translated method is structured as follows (see Listing 1.5): at the beginning some PadMig specific variables and locals of the original method are declared. In case the `__parentState` parameter is not `null`, the method call is a continuation of the execution after a migration, which means that the values of the locals have to be restored from the saved call stack, and that the entry point, from where on to resume the execution, has to be set. The original method body is enclosed in a `try` statement, whose catcher creates a new stack frame and saves

all the locals in case of a migration. The purpose of the endless `while(true)` loop around the original method body will be explained later together with the unfolding technique. Finally, the `switch` statement is used to jump back into the original code at the correct entry point.

Listing 1.4. A simple example (Java code with PadMig annotations)

```
1  import java.io.*;
2  import padmig.*;
3
4  public class Example implements Serializable {
5    @Undock
6    public int foo(Object bar) throws MigrationException {
7      double myDoubleLocal;
8      @DontMigrate
9      long myLongLocal;
10     // method body
11     return 42;
12   }
13 }
```

Listing 1.5. Output of the PadMig compiler for the example in Listing 1.4

```
1  import java.io.*;
2  import padmig.*;
3  import padmig.lib.*;
4
5  public class Example implements Serializable {
6    public int foo(Object bar) throws MigrationException {
7      try {
8        return foo(bar, null);
9      } catch (SaveStack stack) {
10       return ((Integer)(padmig.lib.PadMigLib.
           syncTransmit(stack)));
11     }
12   }
13
14   public class FooStackFrame extends StackFrame {
15     public Object bar;
16     public double myDoubleLocal;
17
18     public Object continueExecution() throws Exception,
         SaveStack {
19       StackFrame frame = new EmptyStackFrame();
20       frame.child = this;
21       return ((Example)(self)).foo(null ,frame);
22     }
23   }
24
25   public int foo(Object bar, StackFrame __parentState)
       throws SaveStack, MigrationException {
```

```
26      FooStackFrame __state = null;
27      FooStackFrame __tmpState;
28      int __entryPoint = 0;
29      int __cFlowBreakLevel;
30      double myDoubleLocal = 0;
31      long myLongLocal = 0;
32      if (__parentState != null) {
33        __state = ((FooStackFrame)(__parentState.child));
34        __entryPoint = __state.entryPoint;
35        bar = __state.bar;
36        myDoubleLocal = __state.myDoubleLocal;
37      }
38      try {
39        while (true) {
40          __cFlowBreakLevel = -1;
41          switch (__entryPoint) {
42            case 0:
43              // method body
44              return 42;
45          }
46        }
47      } catch (SaveStack __stack) {
48        __state = new FooStackFrame();
49        __state.self = this;
50        __state.entryPoint = __entryPoint;
51        __state.bar = bar;
52        __state.myDoubleLocal = myDoubleLocal;
53        __state.child = __stack.bottomOfStack;
54        __stack.bottomOfStack = __state;
55        throw __stack;
56      }
57    }
```

Now we are ready to have a look at the actual translation of a migratable method. It is organized in two traversals of the syntax tree. During the first pass, the following tasks are done:

- If migratable code occurs in places where it is not allowed or not supported, the compilation is aborted. In particular, no migratable code is allowed inside `assert` statements, as the left-hand side of an assignment, in looping expressions, in `synchronized` sections, in catchers, and in finalization blocks.
- Local variable declarations are moved to the beginning of the method, i.e., their scope is widened to the whole method. This is necessary because we need to access them when saving their values in a stack frame upon a migration and when restoring their values from the stack frame on the remote side (see Listing 1.5). Java ensures disjoint life ranges for local variables with the same name of the same type; so such variables can be unified. However, variables of different types must be disambiguated just to ensure static type safety; thus if two or more variables with the same name but different types

exist, we will distinguish these variables by different appendices to their name, uniquely identifying their types as well as their type arguments and array dimensions if applicable.
The only variable declarations not moved are those declared as parameters in catchers. On the one hand, it is not possible to move them due to the Java language specification; on the other hand, it is also not necessary to do so because catchers are not allowed to contain migratable code.

– Ordinary `for` loops are converted into `while` loops by moving the initial assignment(s) before the loop and the increment operation(s) to the end of the loop.
  Enhanced `for` loops (also known as *foreach* loops) are handled similarly: If the looping expression is a subtype of `java.lang.Iterable`, a compiler generated variable of type `java.util.Iterator` parameterized with the appropriate type element is initialized before the loop; the `hasNext()` operation is used as the new looping expression, and the value obtained via the `next()` operation is assigned to the respective local variable in a new first statement of the loop's body.
  Else, if the looping expressing has an array type, a compiler generated variable to hold a reference to the array and a second variable of type `int` to iterate through the array are initialized before the loop; this iteration variable is compared to the length of the array in the new looping expression; as a new first statement of the loop's body the particular array element is assigned to the respective local variable, and as a new last statement the iteration variable is increased.
  All three cases are illustrated in Listings 1.6 and 1.7.
– Loops, `if`-, `switch`-, and `try`-statements containing migratory invocations are marked for unfolding during the second traversal of the syntax tree.
– Statements containing one or more migratory invocations are expanded. This is illustrated in Listings 1.8 and 1.9. If such statements would not be expanded, this could lead to redundant execution of already executed code. In the example, the call to `foo()` would be executed a second time at the migration destination, if a migration occurs and the statement were not expanded using temporary variables.

Listing 1.6. A simple example of `for` loops to convert into `while` loops

```
1  for (i = 0, j = 42; i < 3; i++, j-= 2) {
2    // loop body
3  }
4
5  Set<String> set;
6  // ...
7  for (String s : set) {
8    // loop body
9  }
10
11 String[] array;
```

```
12 // ...
13 for (String t : array) {
14   // loop body
15 }
```

Listing 1.7. Converted loops from Listing 1.6

```
1  i = 0;
2  j = 42;
3  while (i < 3) {
4    // loop body
5    i++;
6    j -= 2;
7  }
8
9  Set<String> set;
10 // ...
11 String s;
12 Iterator<String>
     __gen_java_util_Iterator_java_lang_String = set.
     iterator();
13 while (__gen_java_util_Iterator_java_lang_String.hasNext
     ()) {
14   s = __gen_java_util_Iterator_java_lang_String.next();
15   // loop body
16 }
17
18 String[] array;
19 // ...
20 String t;
21 String[] __gen_java_lang_String_array = array;
22 int __gen_int = 0;
23 while (__gen_int < __gen_java_lang_String_array.length)
     {
24   t = __gen_java_lang_String_array[__gen_int];
25   // loop body
26   __gen_int++;
27 }
```

Listing 1.8. A simple example for expansion of a statement

```
1 result = foo() + bar(myMigratoryMethod());
```

Listing 1.9. Expanded statement from Listing 1.8

```
1 tmp1 = foo();
2 tmp2 = myMigratoryMethod();
3 result = tmp1 + bar(tmp2);
```

During the second traversal of the syntax tree, marked control structures are unfolded. The idea behind unfolding is to reduce the control flow to basic blocks with migratory invocations and then convert it into a finite automaton implemented as a Java `switch` statement surrounded by an endless loop; the state of the automaton, implemented as an `int`, serves as a symbolic program counter during migration.

The loops, `if`-, `switch`-, and `try`-statements identified to contain migratory invocations during the first syntax tree traversal are now rewritten as follows:

In order not to duplicate all code fragments of a loop body before, between, and after migratory invocations or nested statements to be unfolded, the loop is moved to the top level, and a `switch` statement is used to jump to the correct entry point. At the end of a `do` loop, the looping condition is checked using an `if` statement, and if the looping condition is still fulfilled, we jump back to the top of the loop's body by setting the correct entry point and using the `continue` statement. In case loops subject to unfolding are nested, the hierarchy is flattened this way, i.e., we only have one outer `while(true)` loop and `switch` statement.

When unfolding a `while` loop, the negated looping condition is additionally checked in the beginning to skip over the loop body in case the looping condition is already initially false. Unfolding of both loop types is illustrated in Listings 1.10 and 1.11.

Listing 1.10. A simple example for loop unfolding

```
1  while (i < 3) {
2    // while loop body
3  }
4
5  do {
6    // do loop body
7  } while (j < 5);
```

Listing 1.11. Unfolded code from Listing 1.10

```
1  __entryPoint = 0;
2  while (true) {
3    switch (__entryPoint) {
4    case 0:
5      if (!(i < 3)) {
6        __entryPoint = 2;
7        continue;
8      }
9    case 1:
10     // while loop body
11     if (i < 3) {
12       __entryPoint = 1;
13       continue;
14     }
15   case 2:
```

```
    // code between while and do loop
  case 3:
    // do loop body
    if (j < 5) {
      __entryPoint = 3;
      continue;
    }
  case 4:
    return;
  }
}
```

`if` statements are handled by inverting the conditional expression to jump into the `else` part (if present) in case the condition is not fulfilled. An Example is provided in Listings 1.12 and 1.13.

Listing 1.12. A simple example for unfolding an `if` statement

```
if (i == 0) {
  // if body
} else {
  // else body
}
```

Listing 1.13. Unfolded code from Listing 1.12

```
__entryPoint = 0;
while (true) {
  switch (__entryPoint) {
  case 0:
    if (!(i == 0)) {
      __entryPoint = 2;
      continue;
    }
  case 1:
    // if body
    __entryPoint = 3;
    continue;
  case 2:
    // else body
  case 3:
    return;
  }
}
```

When processing a `switch` statement, a separate entry point is generated for each `case` label (see Listings 1.14 and 1.15).

Listing 1.14. A simple example for unfolding a `switch` statement

```
switch (i) {
```

```
2  case 0:
3    // case 0 body
4    break;
5  case 1:
6    // case 1 body
7    // fall-trough into case 2
8  case 2:
9    // case 2 body
10   break;
11 default:
12   // default case
13 }
```

Listing 1.15. Unfolded code from Listing 1.14

```
1  __entryPoint = 0;
2  while (true) {
3    switch (__entryPoint) {
4    case 0:
5      switch (i) {
6      case 0:
7        __entryPoint = 1;
8        continue;
9      case 1:
10       __entryPoint = 2;
11       continue;
12     case 2:
13       __entryPoint = 3;
14       continue;
15     default:
16       __entryPoint = 4;
17       continue;
18     }
19   case 1:
20     // case 0 body
21     __entryPoint = 5;
22     continue;
23   case 2:
24     // case 1 body
25   case 3:
26     // case 2 body
27     __entryPoint = 5;
28     continue;
29   case 4:
30     // default case body
31   case 5:
32     return;
33   }
34 }
```

If there are migratory invocations inside a `try` block, the block has to be split before each migratory invocation. Each such code fragment is surrounded by its own copy of the original `try` block. The exception handling code can simply be copied for each of the new `try` statements, but finalization blocks and labels require special treatment (cf. Listings 1.16 and 1.17). The finalization code is only to be executed when the whole `try` block is executed to completion without errors, when an exception occurs, or when the control flow is diverted using `return`, `break`, or `continue`. For this purpose, the nesting-depth of each `try` statement is determined, and the special local variables `__tryNestingDepth` and `__cFlowBreakLevel` are introduced to indicate whether or not to run the finalization code when leaving a copy of a split `try` statement. `__tryNestingDepth` is increased at the beginning of a copy of a split `try` statement and decreased at its end (except for the last copy) or when a `SaveStack` throwable is caught as a result of a migration. `__cFlowBreakLevel` is set to -1 by default and to the number of the outer-most finalization block to run if the control flow is diverted. The finalization code is eventually surrounded by an `if` statement to ensure that it is only executed if the `__tryNestingDepth` has not been decreased or the `__cFlowBreakLevel` has been set accordingly.

Each copy (except the last one) of a split `try` statement is succeeded by code to skip over the remaining copies in case of an abnormal termination, i.e., when the `__tryNestingDepth` has not been decreased.

Finally, we need to remove unused catchers as not all the catchers might be necessary in every copy of the `try` statement.

Listing 1.16. A simple example for unfolding a `try` statement

```
1  myLabel: try {
2    foo();
3    if (x == 1) {
4      break myLabel;
5    }
6    myMigratoryMethod();
7  } catch (MyException e) {
8    // exception handling code
9  } finally {
10   // finalization code
11 }
```

Listing 1.17. Unfolded code from Listing 1.16

```
1  __entryPoint = 0;
2  while (true) {
3    __cFlowBreakLevel = -1;
4    __tryNestingDepth = -1;
5    switch (__entryPoint) {
6    case 0:
7      try {
8        __tryNestingDepth = 1;
```

```
 9          foo();
10          if (x == 1) {
11            __cFlowBreakLevel = 0;
12            __entryPoint = 2;
13            continue;
14          }
15          __entryPoint = 1;
16          __tryNestingDepth = 0;
17        } catch (MyException e) {
18          // exception handling code
19        } finally {
20          if ((__cFlowBreakLevel == -1 && __tryNestingDepth
              >= 1) || (__cFlowBreakLevel >= 0 &&
              __cFlowBreakLevel < 1)) {
21            // finalization code
22          }
23        }
24        if (__tryNestingDepth > 0) {
25          __entryPoint = 2;
26          continue;
27        }
28      case 1:
29        try {
30          __tryNestingDepth = 1;
31          myMigratoryMethod(__state);
32          __state = null;
33          // __tryNestingDepth NOT decreased here because
              this is the last part of a split try block
34        } catch (MyException e) {
35          // exception handling code
36        } catch (SaveStack __t) {
37          __tryNestingDepth = 0;
38          throw __t;
39        } finally {
40          if ((__cFlowBreakLevel == -1 && __tryNestingDepth
              >= 1) || (__cFlowBreakLevel >= 0 &&
              __cFlowBreakLevel < 1)) {
41            // finalization code
42          }
43        }
44      case 2:
45        return;
46    }
47  }
```

After the second traversal of the syntax tree we finally have to determine and link the correct entry points for the structured non-local jumps caused by `break` and `continue` statements. This completes the translation of a migratory method.

5 Evaluation

In this section, we will discuss the impact of our translation approach in terms of code growth and increased running time. The basic structure of an unfolded method causes a small constant overhead by

- an additional method parameter,
- a few additional local variables (`int`s and references), including their initial assignment,
- an `if` statement, whose body is only executed in case of a migration,
- a `try` statement, whose catcher will only be triggered in case of a migration,
- a `while` loop, which will be iterated more than once only if the translated method contains at least one unfolded element or migration, and
- a `switch` statement, which will be evaluated once per iteration of the main loop and which contains more than one `case` label only if the translated method contains at least one unfolded element or migration.

The methods subject to unfolding are usually the big, central ones in a software library; however, there are typically only a few of them, so that only a small part of the total amount of code is concerned.

Only in case of a migration

- an exception is thrown, caught, and passed on,
- a stack frame object is instantiated, consisting of a few basic member variables (`int`s and references) and additional members corresponding to all locals on the stack that are not excluded from migration,
- all members of the stack frame object are initialized using flat copies of the locals on the stack, and
- the stack frame object is inserted into a linked list.

When the execution continues after a migration, all members of the stack frame object are copied back into the local variables using flat copies.

Altogether, this overhead is negligible for an ordinary execution of a method and minimal for a migration. In the following we will discuss the additional code growth and running time increase caused by unfolding.

When `if` or `switch` statements are unfolded, no overhead is generated for the `if` block or the first `case` block, respectively, and little constant overhead (increase of symbolic program counter, one iteration of the main loop, and one evaluation of the main `switch` statement) for the `else` block or all subsequent `case` blocks, respectively.

In unfolded loops the loop condition is checked using an `if` statement at the end of the loop body, so we have the same little constant overhead as for an `if` statement per loop iteration (except the last iteration, where we fall through into the code below the loop without overhead).

Only when unfolding `try` statements, we experience a notable overhead because the catchers and finalization code are copied (and transformed causing a small constant overhead) once for every migratory invocation in the `try` block.

However, this affects typically only quite short portions of error handling code; in particular, only about 3% of the code is usually enclosed by `try` statements according to [16]. Furthermore, the running time does not increase because at most one of the copied catchers is actually executed.

Nesting of unfolded elements does not cause any additional overhead. Altogether, the overhead caused by unfolding is acceptable in terms of code growth and negligible in terms of the running time. For Example, the total size of the bytecode of PUB-Web grows from 413 KB to 423 KB, which is an increase of 2.4%.

The data transferred during a migration consists of the instance of the class, whose method is subject to migration, all serialization dependencies, and the contents of the stack from the topmost element up to the `@Undock` boundary. Recall, that the developer can exclude stack elements from migration using `@DontMigrate` and thus reduce the volume of the transferred stack contents to the minimum. The absolute data volume transferred depends on the efficiency of serialization. Beside manual fine tuning using the `Externalizable` interface, Java's default serialization mechanism can be substituted by more efficient drop-in replacements. The "UKA-serialization" [11], for example, reduces the serialization overhead for objects, which are similar to our stack frame objects, notably by 81% to 97% compared to JDK.

6 Conclusion

In this paper we have presented the PadMig Language Specification, which uses annotations instead of additional keywords for the migration primitives, so that developers can keep using their favorite IDE and only need to run the PadMig compiler before the Java compiler to produce migratable code. Furthermore, if they additionally need a non-migratable version of their program, they can simply compile their code a second time leaving away the intermediate PadMig compilation step, i.e., they do not need to maintain two versions of their code.

The PadMig compiler supports the Java 6 standard, is stable, efficient, and produces user-friendly, meaningful error / warning messages on compilation problems. In this paper, the translation concepts, especially the implementation of the unfolding technique, have been presented in detail.

References

1. Bonorden, O., Gehweiler, J., Meyer auf der Heide, F.: A web computing environment for parallel algorithms in Java. In: Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM). pp. 801–808 (2005)
2. Cabri, G., Leonardi, L., Quitadamo, R.: Enabling java mobile computing on the ibm jikes research virtual machine. In: PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java. pp. 62–71 (2006)
3. The Caml language. Website: http://caml.inria.fr/

4. Fünfrocken, S.: Transparent migration of Java-based mobile agents. In: Mobile Agents. pp. 26–37 (1998)
5. The first JavaGo prototype. Website: http://homepage.mac.com/t.sekiguchi/javago/
6. Ma, M.J.M., Wang, C.L., Lau, F.C.M.: Delta execution: A preemptive Java thread migration mechanism. Cluster Computing 3(2), 83–94 (2000)
7. Noguera, C., Pawlak, R., Petitprez, N.: Spoon: Program analysis and transformation in java. Tech. Rep. N° 5901, Institut National de Recherche en Informatique et en Automatique (INRIA) (2006)
8. The Paderborn Thread Migration and Checkpointing (PadMig) library. Website: http://padmig.cs.uni-paderborn.de/
9. The Paderborn Remote Method Invocation (PadRMI) library. Website: http://padrmi.cs.uni-paderborn.de/
10. Pawlak, R.: Spoon: annotation-driven program transformation—the AOP case. In: AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development (2005)
11. Philippsen, M., Haumacher, B.: More efficient object serialization. In: Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. pp. 718–732 (1999)
12. The Paderborn University BSP-based Web Computing (PUB-Web) library. Website: http://pubweb.cs.uni-paderborn.de/
13. Sakamoto, T., Sekiguchi, T., Yonezawa, A.: Bytecode transformation for portable thread migration in Java. In: ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents. pp. 16–28 (2000)
14. Sekiguchi, T., Masuhara, H., Yonezawa, A.: A simple extension of Java language for controllable transparent migration and its portable implementation. In: Coordination Models and Languages. pp. 211–226 (1999)
15. The Spoon framework. Website: http://spoon.gforge.inria.fr/
16. Thies, M.: Combining static analysis of Java libraries with dynamic optimization. Shaker Verlag (2001)
17. Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., Verbaeten, P.: Portable support for transparent thread migration in Java. In: ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents. pp. 29–43 (2000)