Ahead-of-time Compilation of FastR Functions Using Static Analysis*

Jeremie Miserez
jeremie.miserez@oracle.com

Oracle Labs

August 2016

Abstract. The FastR project delivers high peak-performance through the use of JIT-compilation, but cannot currently provide this performance for methods on first call. This especially affects startup-performance and performance of applications that only call functions once, possibly with large inputs (i.e. data processing). This project presents an approach and the necessary patterns for implementing an AOT-compilation facility within FastR, enabling compilation of call targets just before being first called. The AOT-compilation produces code that has profiling and specialization information tailored to the expected function argument values for the first call, without needing to execute the function in full. The performance results show a clear and unambiguous performance gain for first-call performance of AOT-compiled functions (up to 4x faster, excluding compilation time). Due to constant compilation time there is the potential for overall startup performance improvement for long-running functions even when compilation time is included. While the static analysis itself imposes almost no overhead, compilation times are up to 1.4x higher than with regularly compiled code, due to the inherent imprecision of the current analysis. Although peak performance is reduced, AOT-compilation can be the solution where faster first-call performance, the possibility of offloading/remote execution, and more performance predictability are important.

1 Introduction

1.1 FastR compilation strategy

FastR is an implementation of the R language, based the Truffle framework. It supports just-in-time (JIT) compilation of frequently used methods through the use of the Graal compiler. In order to achieve high peak performance, FastR heavily relies on profiling information and specializations to enable the partial evaluator to produce efficient code during the compilation. As such, FastR has to rely on deoptimization when any of the assumptions no longer hold true

*Formerly "GR-142: Static analysis for FastR"

during execution of a JIT-compiled method, e.g. when a branch is taken that was never taken during the initial profiling runs. If the method is hot enough, it may then later be recompiled with the new profiling information.
Call targets (R functions, or loop bodies in the case of on-stack replacement (OSR)) may only be candidates for compilation after they have been called a certain minimal number of times, with the default being 3 calls. This minimum guarantees that i) the method is hot, ii) that there is sufficient profiling information, and iii) that any specializations and accompanying AST node rewrites have already been done. Attempting to compile methods earlier (e.g. before the first call) would result in immediate deoptimizations on the subsequent call due to missing profiling information or AST node rewrites, invalidating the initially compiled code in the process.

1.2 Compilation on first call

With the current implementation the only possibility of encouraging FastR to JIT-compile a specific method is to call it repeatedly. At some later point, the method may then be compiled. However, for certain functions this is not possible, and it would be advantageous if compilation could be triggered before the first call of the function. For the subsequent first call to succeed, a guarantee would be needed that the compiled version will be used and that no deoptimizations will happen on the next call. This could significantly improve the first-call performance of certain functions, as well as improve predictability with regard to execution time. It also has the potential to improve the startup performance of FastR as a whole.
Specific functions that could immediately benefit from such an option would be long-running functions that are only called once and are not candidates for OSR, as well as functions that process large amounts of input data. Compilation brings with it a whole host of optimization opportunities, and optimizations such as cross-loop optimizations could significantly affect performance even for a single execution.
Alternative scenarios that could make use of a mechanism for ahead-of-time (AOT) compilation are e.g. offloading scenarios, where the goal is either to avoid compiling N times on N remote nodes, or where the remote nodes do not have the capability to run the full JVM/FastR and can only execute straightforward machine code.

1.2.1 Feasibility of AOT compilation

As the main goal is to ensure successful use of the compiled code on first-call, the proposed mechanism would need to set up all profiles and specializations exactly as needed for the first call, i.e. without any deoptimizations. While simply compiling in every possible branch is technically possible, this would quickly result in the explosion of the size of the compiled code and would be impractical. For specializations and AST node rewrites, compiling a completely generic version would not only make many optimizations impossible, but also require drastic changes in FastR and Truffle. Thus, the most practicable approach is to make the AOT compilation dependent on the expected input for the next call(s). This means that any ahead-of-time (AOT) compilation scheme within FastR will be aware of the actual first-call input arguments.

Warming up using sample data
The simplest way to get compiled code satisfying these criteria is for the programmer to simply call the function 3 times with the full inputs. While this will result in the desired compiled code, it does not in any way achieve the goal of improved performance. A better approach is to derive smaller inputs from the full inputs, e.g. by subsampling input vectors to make them smaller. The method of generating this sample data is application-specific and up to the programmer; given the right sample data FastR will generate compiled code that will not deoptimize when it is later called with the full inputs for the first time. With small enough sample data and large enough full inputs, this method can significantly improve startup performance and works to achieve peak-performance quickly in practice.

Caveats
While the aforementioned method can be a useful manual tool for any FastR programmer, it is not bullet-proof. Programmers must take care to generate sample data that exercises all the same profiles as the full data, and this may change quickly with the data and/or between FastR versions (as new profiles are added to FastR). Even assuming that perfect sample data sets can be found, there is still no visibility into what goes on internally (e.g. profiles touched, side-effects, caches, etc), large vector allocations may still happen, and there is no real guarantee that deoptimizations will not happen for some unforeseen reasons. In addition, this method will not work at all for functions that do not take inputs but nevertheless do large amounts of work within the function body.

2 Static analysis

A more generic and robust approach is to analyze the function in advance without executing it, in order to determine exactly how the profiles and specializations need to be set up. In our case however, in addition to this we also directly transform the Truffle AST and any internal FastR structures that have directly to do with profiling, as these may affect what code the compiler and partial evaluator will later generate. We cannot gather the required information by running any of the available static analysis tools on the R code or the Truffle AST, thus our approach will be a combination of static analysis and execution. In the current implementation the focus is on functions taking RDoubleVectors as arguments, a common case for data processing functions.

2.1 Simulation types

We define a simulation type as a data type that implements the interface of a real data type, but compared to the real type the implementations only update the relevant parts of the analysis instead of performing any work. The idea is that to run an analysis within FastR we should be able to simply execute the function in question, providing simulation type replacements for all input arguments and variables. Then, when the function is compiled all mentions of simulation types should be rewritten to point to the equivalent real type so that the compiled code can run with the real types only.
In our implementation, the main type supported for the analysis is the RDoubleVector, which represents an R vector containing numeric values within FastR, and which implements the RAbstractDoubleVector interface. Each RDoubleVector contains a fixed length Java double array containing it's values.
The corresponding simulation type is named RSimulationDoubleVector, and implements the same interface as the real type, however it does not contain a Java array of doubles. Instead, it simply contains a abstract representation of the state of the vector[1]. The basic idea is to use RSimulationDoubleVectors instead of RDoubleVectors and generating the simulation type equivalents where needed, e.g. in the very beginning from any arguments of type RDoubleVector. Then, any internal FastR methods call the corresponding interface methods which update the abstract state of the RSimulationDoubleVector, and thus the analysis. Note that for this approach, complete knowledge of the program state and specifically the function arguments immediately before the analysis/call is necessary.

2.2 Abstract interpretation

We would like the simulation types to be significantly faster and use less memory than the real types: they should be significantly faster than actually doing the computation with the real types (i.e. the real, large vectors). In order to achieve this we turn to abstract interpretation, which should give us a sound approximation of the values the real types would have.

[1] The concrete length, and an abstract representation of the possible element values.

2.2.1 Abstract domains

A key part of this is defining the appropriate abstract domains for the relevant parts of the analysis. All abstract domains below implement the *meet* and *join* operations, and have at least the $\top$ and $\bot$ states. Domains that represent numbers also implement abstract numerical operations (e.g. $+, -, *, /,$) while the Java boolean domain implements logical operations such as and, or.

Java boolean

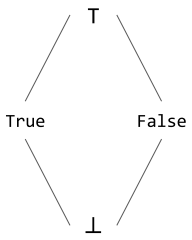


Figure 1: The abstract domain for the Java boolean data type.

The abstract domain shown in Figure 1 represents the set of possible values for Java boolean variables. In this case there are four different values: $\bot = \{\}$, $\top = \{true, false\}$, $True = \{true\}$, $False = \{false\}$. This allows determining e.g. if the value is always/never true, always/never false, possibly both true and false, and so on. Note that this is exactly the sort of information necessary to determine if e.g. branches could be taken or not.

Java int

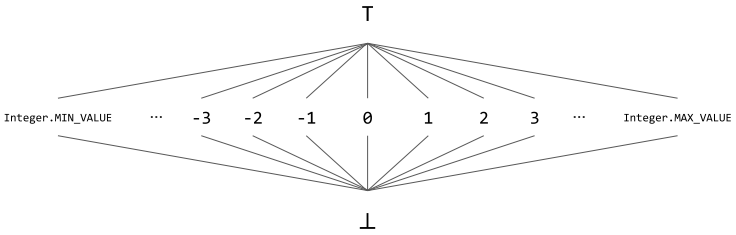


Figure 2: The abstract domain for the Java int data type.

Figure 2 shows a simple abstract domain for Java int variables, where either a concrete value is known, any value is possible ($\top$), or no value is possible (unreachable state). Other abstract domains exist, e.g. Graal has the IntegerStamp.

RDouble, RDoubleVector

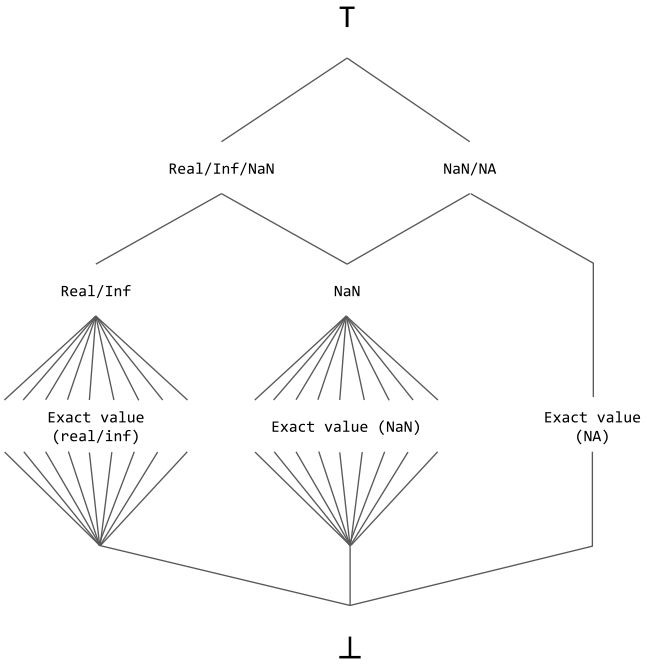


Figure 3: The abstract domain for the RDouble and RDoubleVector data types.

This domain is intended to represent the possible values within an RDouble or RDoubleVector. Note that we can represent both using the same domain, as we can just use a single abstract RDouble to represent the possible values contained within the elements of the RDoubleVector vector (the length of the vector is then represented using the Java int abstract domain). This domain implements all the common unary and binary mathematical operations and allows to check whether NA values or NaNs are part of the possible set of values. Note that the exact value labels represent that there is one element for each possible value, analogous to the abstract domain for Java int.

```
@Override
public AiRDouble floor() {
    if (this.state == State.EXACT_REAL_VALUE) {
        return new AiRDouble(Math.floor(this.value));
    } else {
        return this.copy();
    }
}
```

Listing 1: Implementation of the *floor* operation for the abstract RDouble domain (in AiRDouble.java)

```java
@Override
public AiRDouble sqrt() {
    if (this.state == State.EXACT_REAL_VALUE) {
        return new AiRDouble(Math.sqrt(this.value));
    } else {
        if (this.maybeNonNANonNaNReal()) {
            return this.join(new AiRDouble(State.NAN)); // negative values
        } else {
            return this.copy();
        }
    }
}
```

Listing 2: Implementation of the *sqrt* operation for the abstract RDouble domain (in AiRDouble.java)

```java
@Override
public AiRDouble div(AiAbstractDomain other) {
    AiRDouble otherCast = castSameType(other);
    if (this.isBot() || otherCast.isBot()) {
        return BOT.copy();
    } else if (this.alwaysNA() || otherCast.alwaysNA()) {
        return new AiRDouble(RRuntime.DOUBLE_NA);
    } else if (this.isExactValue() && otherCast.isExactValue()) {
        return new AiRDouble(this.value / otherCast.value);
    } else {
        boolean maybeNA = (this.maybeNA() || otherCast.maybeNA());
        boolean maybeNaN = (this.maybeNaN() || otherCast.maybeNaN());
        boolean alwaysNaN = (this.alwaysNaN() || otherCast.alwaysNaN());
        boolean maybeThisZero = this.maybeNonNANonNaNReal();
        boolean maybeOtherZero = otherCast.maybeNonNANonNaNReal();
        if (maybeThisZero && maybeOtherZero) {
            maybeNaN = true; // 0/0 = NaN!
        }
        return joinWithNaNan(otherCast, maybeNA, maybeNaN, alwaysNaN);
    }
}

private AiRDouble joinWithNaNan(AiRDouble otherCast) {
    boolean maybeNA = (this.maybeNA() || otherCast.maybeNA());
    boolean maybeNaN = (this.maybeNaN() || otherCast.maybeNaN());
    boolean alwaysNaN = (this.alwaysNaN() || otherCast.alwaysNaN());
    if (alwaysNaN) {
        if (maybeNA) {
            return new AiRDouble(State.NAN_NA);
        } else {
            return new AiRDouble(State.NAN);
        }
    }
    if (maybeNaN) {
        if (maybeNA) {
            return join(otherCast).join(new AiRDouble(State.NAN_NA));
        } else {
            return join(otherCast).join(new AiRDouble(State.NAN));
        }
    }
    if (maybeNA) {
        return join(otherCast).join(new AiRDouble(State.EXACT_NA_VALUE));
    } else {
        return join(otherCast);
    }
}
```

Listing 3: Implementation of the *div* operation for the abstract RDouble domain (in AiRDouble.java)

While some of the operations are simple to implement (e.g. *floor*, Listing 1), others may lead to a loss in precision as we need to approximate: E.g. *sqrt* (Listing 2) may result in a NaN result if the argument is < 0, and division (Listing 3) may result in NaN when dividing 0/0.

3 Implementation

The analysis and AOT compilation is currently implemented for a subset of FastR, and thus a subset of the R language. Currently both unary and binary arithmetic as well as the the accompanying reduce operations are supported for the RDouble and RDoubleVector data types. This means that all R programs which only use double vectors, elementary math operations as well as *sum* or *product* can be AOT-compiled.

3.1 Method signatures for simulation types

Ideally, in order to use simulation types as simple replacements for their real counterparts, all methods within FastR should i) only rely on the interface methods defined in RAbstractDoubleVector, ii) always use RAbstractDoubleVectors for method arguments and return types, rather than RDoubleVectors or Java doubles. This would enable just passing a simulation type where a real type is expected, and it would allow the information gathered during the static analysis to flow back up to the caller (which is hard if one can only return a Java double). Unfortunately, this is not currently the case in FastR, the use of the interface and it's methods is not strictly enforced yet.
Thus, several method signatures needed to be changed in order to support simulation types: For return types, we slightly modify the behaviour and return RDouble (and RSimulationDouble respectively) in place of the Java double. Ideally, after partial evaluation this should result in the same code being generated, but it may negatively affect performance in the interpreter.

```
@Specialization
protected double doDoubleVector(RDoubleVector operand, boolean naRm) {
```

(a) Unmodified code

```
protected boolean transformLocalRDoubleVector(Class<?> c) {
    return RSimulationTypesHelper.CacheHelper.toRealType(c) == RDoubleVector.class;
}

@Specialization(guards = "transformLocalRDoubleVector(operand.getClass())")
protected RAbstractScalarDoubleVector doDoubleVector(RAbstractDoubleVector operand, boolean naRm) {
if (CompilerDirectives.inInterpreter() && TruffleSimulationTypes.isEnabledAndIsSimulationTypeObject(operand)) {
   // simulation type (analysis code) here
} else {
   assert operand instanceof RDoubleVector;
   RDoubleVector operandCast = (RDoubleVector) operand;
   // real type (existing code) here
}
```

(b) Code modified to support simulation types

Listing 4: Modifications to method signature (return type and argument type) (in UnaryArithmeticReduceNode.java)

```
public static class CacheHelper {
    public static Class<?> toRealType(Class<?> c) {
        if (CompilerDirectives.inInterpreter() && TruffleSimulationTypes.isEnabledAndIsSimulationType(c)) {
            return TruffleSimulationTypes.getRealTypeForSimulationType(c);
        } else {
            return c;
        }
    }
}
```

Listing 5: The CacheHelper helper class (in RSimulationTypesHelper.java)

For method arguments, extra guards must be added (Listing 4, Listing 5) when changing from a more specific type (RDoubleVector) to a more general type (RAbstractDoubleVector) to ensure that the compiled code does not need extra type checks or virtual method calls. Without the guard changing the argument type to the interface type may inadvertently introduce virtual method calls as the concrete type is now not known at compile time, and many optimizations are not possible anymore. Adding the guard effectively ensures that only RSimulationDoubleVector and RDoubleVector types are allowed. As simulation types are only used in the interpreter, the compiled code does not need to handle them and can effectively emit the same code as before.

3.2 Profiles

Profiling is used in several different ways within FastR, and the following examples are not exhaustive. However, often the code is very similar, and the same pattern can be used (e.g. for R call argument and return type class profiles).

3.2.1 @Cached

```
@Specialization(limit = "3", guards = "vector.getClass() == cachedClass")
protected static Object doCached(Object vector, @Cached("vector.getClass()") Class<?> cachedClass) {
        return cachedClass.cast(vector);
}
```

(a) Unmodified code

```
 protected Class<?> transformLocal(Class<?> c) {
    return RSimulationTypesHelper.CacheHelper.toRealType(c);
}

@Specialization(limit = "3", guards = "transformLocal(vector.getClass()) == cachedClass")
protected static Object doCached(Object vector, @Cached("transformLocal(vector.getClass())") Class<?> cachedClass)
        {
    if (CompilerDirectives.inInterpreter() && TruffleSimulationTypes.isSimulationTypeObject(vector)) {
        return vector;
    } else {
        return cachedClass.cast(vector);
    }
}
```

(b) Code modified to support simulation types

Listing 6: Modifications to method signature when @Cached annotation is present (in BoxPrimitiveNode.java)

The caching of parameter types (Class<?> objects) can present a problem when the analysis uses simulation types: The compiled code will contain the simulation types as the cached types, leading to deoptimization when the function is subsequently compiled and called with the real types. To mitigate this issue we lookup the corresponding real type for each simulation type during analysis, and substitute the cached type with the real type. Listing 6 shows an example of this, with Listing 5 containing the code for the helper function which looks up the real type for a given simulation type[2].

[2]Each simulation type has an annotation with a reference to the real type, which is then looked up.

3.2.2 ValueProfile.ExactClass

```java
@SuppressWarnings("unchecked")
@Override
public <T> T profile(T value) {
    // Field needs to be cached in local variable for thread safety and startup speed.
    Class<?> clazz = cachedClass;
    if (clazz != Object.class) {
        if (clazz != null && value != null && clazz == value.getClass()) {
            /*
             * The cast is really only for the compiler relevant. It does not perform any
             * useful action in the interpreter and only takes time.
             */
            if (CompilerDirectives.inInterpreter()) {
                return value;
            } else {
                return (T) clazz.cast(value);
            }
        } else {
            CompilerDirectives.transferToInterpreterAndInvalidate();
            if (clazz == null && value != null) {
                cachedClass = value.getClass();
            } else {
                cachedClass = Object.class;
            }
        }
    }
    return value;
}
```

(a) Unmodified code

```java
@SuppressWarnings("unchecked")
@Override
public <T> T profile(T value) {
    // Field needs to be cached in local variable for thread safety and startup speed.
    Class<?> clazz = cachedClass;
    if (clazz != Object.class) {
        if (clazz != null && value != null && clazz == value.getClass()) {
            /*
             * The cast is really only for the compiler relevant. It does not perform any
             * useful action in the interpreter and only takes time.
             */
            if (CompilerDirectives.inInterpreter()) {
                return value;
            } else {
                return (T) clazz.cast(value);
            }
        } else {
            if (CompilerDirectives.inInterpreter() &&
                            value != null && clazz != null &&
                            TruffleSimulationTypes.isEnabledAndIsSimulationTypeObject(value)) {
                // in 2nd simulation, class does not match, override
                if (!TruffleSimulationTypes.getRealTypeForSimulationTypeObject(value).equals(clazz)) {
                    // cached class is different from real type
                    cachedClass = Object.class;
                }
                return value; // return simulation although real type is stored
            }
            CompilerDirectives.transferToInterpreterAndInvalidate();
            if (clazz == null && value != null) {
                cachedClass = value.getClass();
                if (CompilerDirectives.inInterpreter() && TruffleSimulationTypes.isEnabledAndIsSimulationType(
                        cachedClass)) {
                    // in 1st simulation, override
                    cachedClass = TruffleSimulationTypes.getRealTypeForSimulationType(value.getClass());
                }
            } else {
                cachedClass = Object.class;
            }
        }
    }
    return value;
}
```

(b) Code modified to support simulation types

Listing 7: Modifications to ValueProfile.ExactClass (in ValueProfile.java)

When a ValueProfile.ExactClass is used, the principle is the same as for @Cached annotations, with a slightly different implementation due to the number of states that need to be manually handled. If more than one class is seen, the ValueProfile will cache the generic class Object.class.

3.2.3 ValueProfile.Identity

```java
@Override
@SuppressWarnings("unchecked")
public <T> T profile(T newValue) {
    // Field needs to be cached in local variable for thread safety and startup speed.
    Object cached = this.cachedValue;
    if (cached != GENERIC) {
        if (cached == newValue) {
            return (T) cached;
        } else {
            CompilerDirectives.transferToInterpreterAndInvalidate();
            if (cachedValue == UNINITIALIZED) {
                cachedValue = newValue;
            } else {
                cachedValue = GENERIC;
            }
        }
    }
    return newValue;
}
```

(a) Unmodified code

```java
@Override
@SuppressWarnings("unchecked")
public <T> T profile(T newValue) {
    // Field needs to be cached in local variable for thread safety and startup speed.
    Object cached = this.cachedValue;
    if (cached != GENERIC) {
        if (cached == newValue) {
            return (T) cached;
        } else {
            if (CompilerDirectives.inInterpreter() && TruffleSimulationTypes.isEnabledAndIsSimulationTypeObject(
                    newValue)) {
                // in 1st simulation, override
                cachedValue = GENERIC;
            } else {
                CompilerDirectives.transferToInterpreterAndInvalidate();
                if (cachedValue == UNINITIALIZED) {
                    cachedValue = newValue;
                } else {
                    cachedValue = GENERIC;
                }
            }
        }
    }
    return newValue;
}
```

(b) Code modified to support simulation types

Listing 8: Modifications to ValueProfile.Identity (in ValueProfile.java)

Not all profiles can be set up with simulation types exactly as they would be when running with the actual inputs: The ValueProfile.Identity profile caches a complete object, and has the two additional states `UNINITIALIZED` and `GENERIC`. As we cannot create the correct real type object during simulation, the only remaining option that prevents deoptimizations is to set the profile to the `GENERIC` state. Listing 8 shows this change.

3.2.4 ConditionProfile

```java
public static class ConditionProfileHelper {

    public static void profile(ConditionProfile profile) {
        CompilerAsserts.neverPartOfCompilation();
        profile.profile(true);
        profile.profile(false);
    }

    public static void profile(ConditionProfile profile, AiJavaPrimitiveBoolean value) {
        CompilerAsserts.neverPartOfCompilation();
        if (value.isConcrete()) {
            profile.profile(value.toConcrete());
        } else if (value.isTop()) {
            profile.profile(true);
            profile.profile(false);
        } else {
            assert value.isBot();
            // dont do anything
        }
    }
}
```

Listing 9: Enabling of ConditionProfiles (in RSimulationTypesHelper.java)

With a ConditionProfile, only branches that were taken at least once be present in the compiled code. In order to prevent deoptimization when a previously untaken branch is taken, we can either enable both branches or only the one that will be taken in the next execution. In Listing 9 the first method simply enables both branches, while the second function takes an abstract Java boolean (AiJavaPrimitiveBoolean) and enables exactly the profiles that could possibly be reached according to the set of possible values.

```java
public static AiJavaPrimitiveBoolean determineIsNA(AiRDouble a) {
    CompilerAsserts.neverPartOfCompilation();
    if (a.alwaysNA()) {
        return new AiJavaPrimitiveBoolean(true);
    } else if (a.isBot()) {
        return new AiJavaPrimitiveBoolean().getBot();
    } else if (!a.maybeNA()) {
        return new AiJavaPrimitiveBoolean(false);
    } else {
        assert a.maybeNA();
        return new AiJavaPrimitiveBoolean().getTop();
    }
}
```

(a) Determining if an abstract RDouble is/could be NA

```java
/**
 * return !isNAorNaN(d) && !Double.isInfinite(d);
 */
public static AiJavaPrimitiveBoolean determineIsFinite(AiRDouble a) {
    CompilerAsserts.neverPartOfCompilation();
    if (a.alwaysFinite()) {
        // always finite
        return new AiJavaPrimitiveBoolean(true);
    } else if (a.isBot()) {
        return new AiJavaPrimitiveBoolean().getBot();
    } else if (!a.maybeFinite()) {
        // never finite
        return new AiJavaPrimitiveBoolean(false);
    } else {
        assert a.maybeFinite() && !a.alwaysFinite();
        return new AiJavaPrimitiveBoolean().getTop();
    }
}
```

(b) Determining if an abstract RDouble is/could be finite

```java
public static AiJavaPrimitiveBoolean testEquals(AiRDouble a, AiRDouble b) {
    CompilerAsserts.neverPartOfCompilation();
    if (a.isBot() || b.isBot()) {
        // cannot say anything
        return new AiJavaPrimitiveBoolean().getBot();
    }
    if (a.isConcrete() && b.isConcrete()) {
        return new AiJavaPrimitiveBoolean(a.toConcrete() == b.toConcrete());
    } else if (a.alwaysNAorNaN() || b.alwaysNAorNaN()) {
        // comparisons with NaN are always false
        return new AiJavaPrimitiveBoolean(false);
    } else if (a.contains(b) || b.contains(a)) {
        // there may exist a pair which are equal
        return new AiJavaPrimitiveBoolean().getTop();
    } else {
        // no two pairs are equal
        return new AiJavaPrimitiveBoolean(false);
    }
}
```

(c) Determining if two abstract RDoubles are/could be equal to each other

Listing 10: Helper functions to determine conditions as abstract booleans (in RSimulation-TypesHelper.java)

Such abstract booleans can be generated using helper functions, a selection of several such functions is shown in Listing 10.

3.2.5 NACheck

```
public static class NACheckHelper {

    public static void naCheckEnable(NACheck na, boolean enable) {
        CompilerAsserts.neverPartOfCompilation();
        if (enable) {
            na.enable(true);
            na.check(RRuntime.DOUBLE_NA);
        }
    }

    public static void naCheckEnable(NACheck na, AiJavaPrimitiveBoolean value) {
        CompilerAsserts.neverPartOfCompilation();
        if (value.isConcrete()) {
            naCheckEnable(na, value.toConcrete());
        } else if (value.isTop()) {
            naCheckEnable(na, true);
        } else {
            assert value.isBot();
            // dont do anything
        }
    }

    public static void naCheckEnable(NACheck na, AiRDouble value) {
        CompilerAsserts.neverPartOfCompilation();
        naCheckEnable(na, determineIsNA(value));
    }

}
```

Listing 11: Enabling of NAChecks (in RSimulationTypesHelper.java)

If NAs are expected to be present in the input, NACheck profiles can be set to always check for NAs and not deoptimize when an NA is encountered. However, this usually comes at the expense of performance and reduces the number of opportunities for optimizations. However, we can again use the helper function $determineIsNA$ method from Listing 10 to determine if abstract RDouble contains NA values within the it's set of possible values. We then enable only the NAChecks that are necessary. In Listing 11, the first function always enables NAChecks, while the second and third functions use information from the abstract RDouble domain.

3.2.6 RPromise evaluation

In order to run the analysis and convert the function arguments to simulation types, knowledge of the values of the function arguments is necessary (§2.1). In FastR programs, function arguments are often supplied as RPromise objects that are then evaluated. To prevent this from interfering with the analysis, the results of all RPromise evaluations are converted on-the-fly to the corresponding simulation types. Currently, RPromise evaluation is only available for function arguments, as the analysis is not run for functions executed during evaluation of a promise.

3.3 Translation of existing code

Aside from profiles, other parts of the FastR codebase need modifications in order to work correctly with simulation types. This is partly because the majority of the internal FastR methods do not use the RAbstractDoubleVector interface exclusively, use explicit casts or call methods that cannot be accurately simulated (e.g. materializing a vector). Several FastR classes and methods have been modified to work with both the real types and the simulation types, exchanging concrete operations with the versions implemented in the abstract domains, i.e. replacing addition of two doubles with the equivalent *plus* operation defined in

the RDouble abstract domain. In addition to this, certain slow and unnecessary or impossible operations are skipped completely, such as vector materialization or vector allocation, while retaining all of the effects on specializations and profiles. Repetitive changes have been collected into helper classes as much as possible, thus making it easier to translate and reason about existing code.

3.3.1 Operations

```
@Override
public double op(double left, double right) {
    return left + right;
}
```

(a) Initial code

```
@Override
public AiRDouble op(AiRDouble left, AiRDouble right) {
    return left.add(right);
}
```

(b) Additional method for the RDouble abstract domain type

Listing 12: Code additions to Add (in BinaryArithmetic.java)

```
@Override
public final double applyDouble(double operand) {
    if (operandNACheck.check(operand)) {
        return RRuntime.DOUBLE_NA;
    }
    return arithmetic.op(operand);
}
```

(a) Initial code

```
@Override
public AiRDouble simTypeApplyDouble(AiRDouble operand) {
    AiJavaPrimitiveBoolean condition = RSimulationTypesHelper.determineIsNA(operand);
    RSimulationTypesHelper.NACheckHelper.naCheckEnable(operandNACheck, condition);
    if (condition.isAlwaysTrue()) {
        return new AiRDouble(RRuntime.DOUBLE_NA);
    }
    return arithmetic.op(operand);
}
```

(b) Additional method for the RDouble abstract domain type

Listing 13: Code additions to ScalarUnaryArithmeticNode.java

```
@Override
public double applyDouble(double left, double right) {
    if (leftNACheck.check(left)) {
        if (this.arithmetic instanceof BinaryArithmetic.Pow && right == 0) {
            // CORNER: Make sure NA^0 == 1
            return 1;
        } else if (this.arithmetic instanceof BinaryArithmetic.Mod && right == 0) {
            // CORNER: Make sure NA%%0 == NaN
            return Double.NaN;
        }
        return RRuntime.DOUBLE_NA;
    }
    if (rightNACheck.check(right)) {
        if (this.arithmetic instanceof BinaryArithmetic.Pow && left == 1) {
            // CORNER: Make sure 1^NA == 1
            return 1;
        }
        if (leftNACheck.checkNAorNaN(left)) {
            // CORNER: Make sure NaN op NA == NaN
            return left;
        }
        return RRuntime.DOUBLE_NA;
    }
    double value = arithmetic.op(left, right);
    resultNACheck.check(value);
    return value;
}
```

(a) Initial code

```
@Override
public AiRDouble simTypeApplyDouble(AiRDouble left, AiRDouble right) {
    AiJavaPrimitiveBoolean condition;
    AiJavaPrimitiveBoolean subCondition;
    condition = RSimulationTypesHelper.determineIsNA(left);
    RSimulationTypesHelper.NACheckHelper.naCheckEnable(leftNACheck, condition);
    if (condition.isAlwaysTrue() && right.isConcrete()) {
        if (this.arithmetic instanceof BinaryArithmetic.Pow && right.toConcrete() == 0) {
            // CORNER: Make sure NA^0 == 1
            return new AiRDouble(1);
        } else if (this.arithmetic instanceof BinaryArithmetic.Mod && right.toConcrete() == 0) {
            // CORNER: Make sure NA%%0 == NaN
            return new AiRDouble(Double.NaN);
        }
        return new AiRDouble(RRuntime.DOUBLE_NA);
    }
    condition = RSimulationTypesHelper.determineIsNA(right);
    RSimulationTypesHelper.NACheckHelper.naCheckEnable(rightNACheck, condition);
    if (condition.maybeTrue()) {
        subCondition = RSimulationTypesHelper.determineIsNAorNaN(left);
        RSimulationTypesHelper.NACheckHelper.naCheckEnable(leftNACheck, subCondition);
    }
    if (condition.isAlwaysTrue() && left.isConcrete()) {
        if (this.arithmetic instanceof BinaryArithmetic.Pow && left.toConcrete() == 1) {
            // CORNER: Make sure 1^NA == 1
            return new AiRDouble(1);
        }
        if (left.alwaysNAorNaN()) {
            return left;
        }
        return new AiRDouble(RRuntime.DOUBLE_NA);
    }
    AiRDouble value = arithmetic.op(left, right);
    RSimulationTypesHelper.NACheckHelper.naCheckEnable(resultNACheck, RSimulationTypesHelper.determineIsNA(value))
        ;
    return value;
}
```

(b) Additional method for the RDouble abstract domain type

Listing 14: Code additions to BinaryMapArithmeticFunctionNode.java

Certain operations are relatively easy to add, e.g. adding versions of mathematical operations for the abstract RDouble domain as seen in Listing 12 or Listing 13. Other operations need more work translating, especially when there are many profiles or nested if-else constructs with profiles that may all need to be exercised.

3.3.2 Java loops

The Java loops present in classes such as UnaryArithmeticReduceNode (that compute the *sum* and *product* functions) present a problem, as they can potentially have a large or unknown number of loop iterations. Simulation types such

as RSimulationDoubleVectors currently store the concrete length and present themselves as having that length (§2.1). Thus for large inputs, the analysis may spend a large amount of time within such loops.

```
assert operand instanceof RDoubleVector;
RDoubleVector operandCast = (RDoubleVector) operand;
RNode.reportWork(this, operandCast.getLength());
boolean profiledNaRm = naRmProfile.profile(naRm);
double result = semantics.getDoubleStart();
na.enable(operandCast);
int opCount = 0;

double data[] = operandCast.getDataWithoutCopying();
for (int i = 0; i < operandCast.getLength(); i++) {
    double d = data[i];
    if (na.check(d)) {
        if (profiledNaRm) {
            continue;
        } else {
            return RDouble.createNA();
        }
    } else {
        result = arithmetic.op(result, d);
    }
    opCount++;
}
if (opCount == 0) {
    emptyWarning();
}
return RDouble.valueOf(result);
```

(a) Loop using real types

```
RSimulationDoubleVector operandCast = (RSimulationDoubleVector) operand;
RNode.reportWork(this, operand.getLength());
boolean profiledNaRm = naRmProfile.profile(naRm);
AiRDouble result = new AiRDouble(semantics.getDoubleStart());
RSimulationTypesHelper.NACheckHelper.naCheckEnable(na, operandCast.getSimTypeValues());

for (int i = 0; i < Math.min(operand.getLength(), RSimulationTypesHelper.AI_MAX_ITER); i++) {
    AiRDouble prev = result;
    Object prevDirection = null;
    AiRDouble d = operandCast.getSimTypeValues();
    if (profiledNaRm) {
        d = d.withoutNA(); // NA is never used, so we remove it from d.
    }
    result = arithmetic.op(result, d);
    Object direction = result.direction(prev);
    if (!result.isMoving(direction)) {
        break; // result is stable, exit.
    } else {
        if (direction == prevDirection && i > RSimulationTypesHelper.AI_WIDENING_ITER) {
            assert prevDirection != null;
            result = result.widen(direction);
            prevDirection = null;
        }
    }
    prevDirection = direction;
}
// opCount is also 0 when all values are NA and profiledNaRm is true
if (operand.getLength() == 0 || (profiledNaRm && na.isEnabled())) {
    TruffleSimulationTypes.enableWithoutObjects(true);
    emptyWarning();
    TruffleSimulationTypes.enableWithoutObjects(false);
}
return new RSimulationDouble(operandCast.getSimTypeIsNull(), result);
```

(b) Loop using abstract interpretation

Listing 15: Code additions to UnaryArithmeticReduceNode.java

All that is needed for our purposes is a sound approximation of the simulation type state after the loop. Thus, we execute loop iterations only until the abstract states of the simulation types are not modified any further and a fixpoint is reached. As all operations are executed within the abstract RDouble domain. As an additional measure to ensure quick fixpoint convergence widening is applied after a fixed number of iterations, if necessary.

4 Evaluation

4.1 Benchmarks

```
work <- function(x) {
  result <- (sum(x))
}
```

(a) Function A

```
work <- function(x) {
  N <- as.double(length(x))
  mu <- (1/N)*(sum(x))
  variance <- (1/N) * sum((x-mu)^2)
  stddev <- sqrt(variance)
  result <- mu+variance+stddev
}
```

(b) Function B

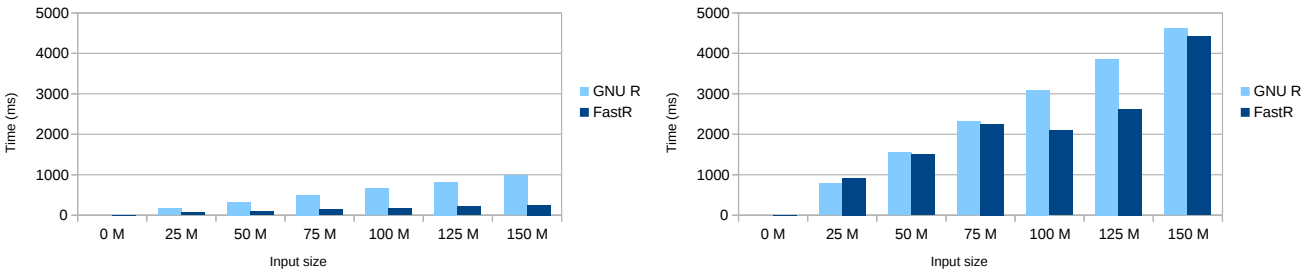
Listing 16: Functions used for evaluation

The evaluation uses two functions (Listing 16) that make use of the currently supported functionality: Function A is a minimal example that calculates the sum of a vector, while Function B calculates the standard deviation of a vector. The functions were run with varying inputs from 0 elements up to $150 Mio.$ elements, once as the baseline, once with a set of subsample data inputs with 1000 elements, and finally once using the AOT-compilation approach described in §2. Each benchmark was run 4 times (1 warmup, 3 runs), with each function being called 10 times per run. All reported numbers are the arithmetic mean over all 3 runs (excluding the warmup run). For comparison an additional benchmark was performed with GNU R 3.2.4. The experiments were performed on a physical machine running Ubuntu 14.04.3, jdk1.8.0_101, a 4x 2.2GHz Intel i7 CPU and 16GB of RAM. Frequency scaling and other power management features were disabled, Hyper-threading was enabled. The project was branched from revisions on June 6, 2016, these revisions[3][4][5][6] were used as a baseline.

[3] fastr Rev. 416bbbc31928817c5a80b2d18b7425b4cfd83ea8
[4] truffle Rev. b2ec743323a1f5b67d9178576a740a0aed1852e1
[5] graal-enterprise Rev. 7850730a50afa16397a4e4b4e98c1741143149c7
[6] graal-core Rev. 505dc834ea584ba3667a0542add4d03b7469bbcb

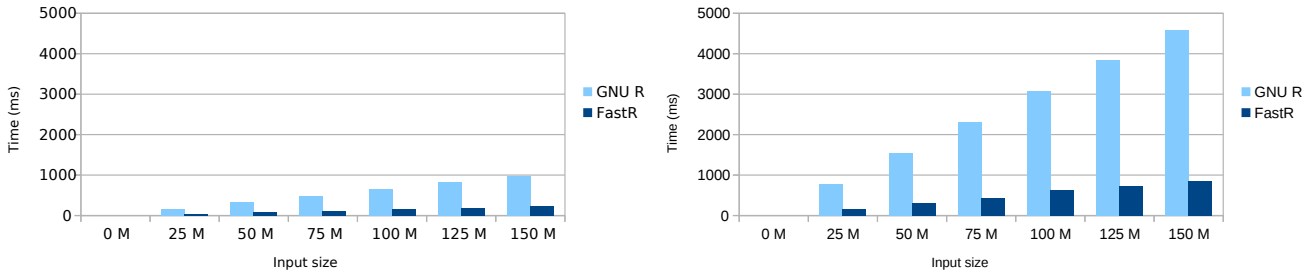
4.2 Performance

4.2.1 FastR baseline performance vs GNU R



(a) Function A (b) Function B

Figure 4: First call performance of GNU R and FastR baseline (both interpreted)

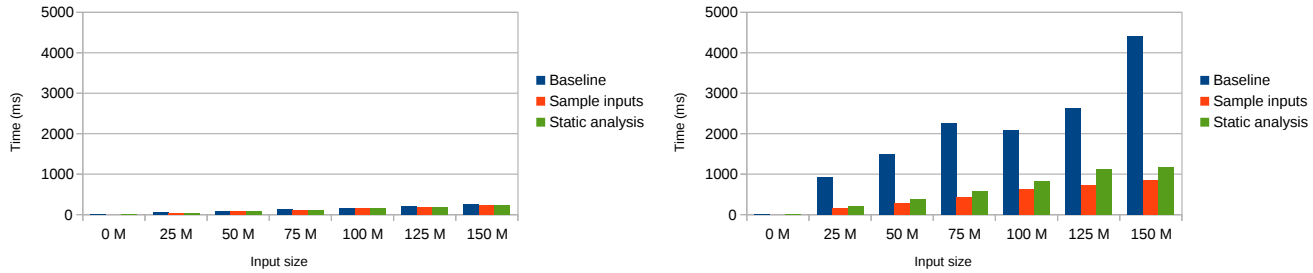


(a) Function A (b) Function B

Figure 5: Peak performance of GNU R (interpreted) and FastR baseline (JIT-compiled)

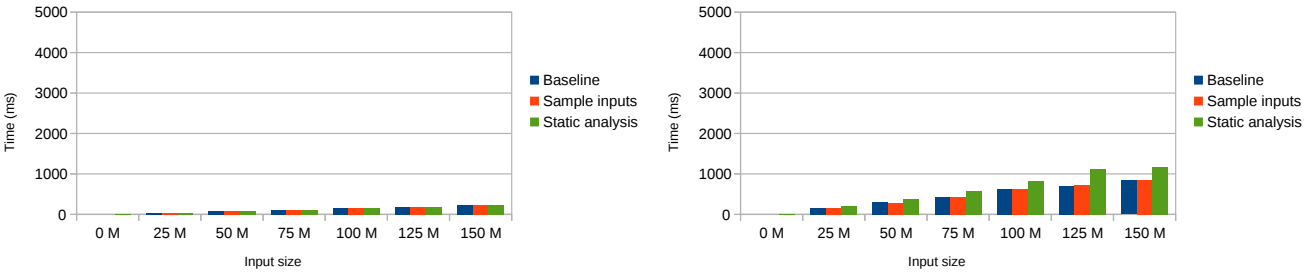
Figure 5 shows that for these two chosen functions, peak performance after warmup and compilation is markedly better with FastR baseline. However, when comparing the performance of the first call of each function in Figure 4 it can be seen that the gap narrows. There is an opportunity for FastR to improve startup/first-call performance.

4.2.2 First call performance comparison between FastR baseline, JIT-compiled, and AOT-compiled code



(a) Function A (b) Function B

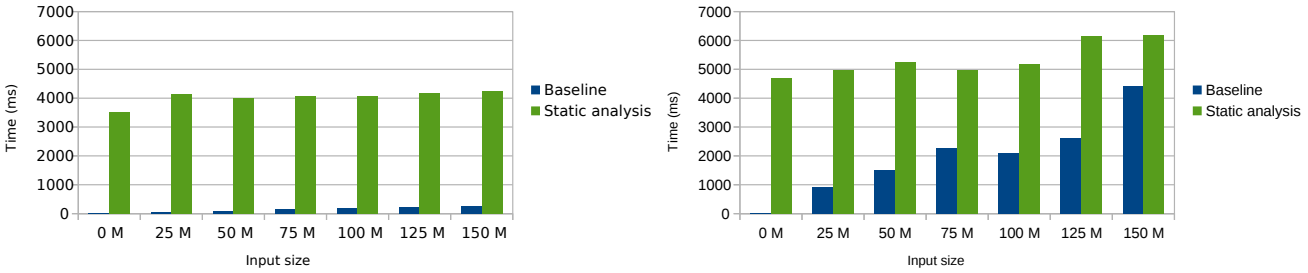
Figure 6: First call with full data, excluding compilation time.
FastR baseline interpreted, Sample inputs JIT-compiled, Static analysis AOT-compiled



(a) Function A (b) Function B

Figure 7: Peak performance with full data, excluding compilation time.
FastR baseline JIT-compiled, Sample inputs JIT-compiled, Static analysis AOT-compiled

From Figure 6 it can be seen that performance of the compiled functions compared to baseline is greatly improved (up to 4x faster) for the first call. This holds true for the static analysis (AOT-compiled, §2.1) approach as well as the sample inputs (warmup using a subset of input date then JIT-compiled, §1.2.1) approach. Peak performance (Figure 7) of the AOT-compiled code is worse (1.3x to 1.6x slower) when compared to the regularly JIT-compiled code.



(a) Function A

(b) Function B

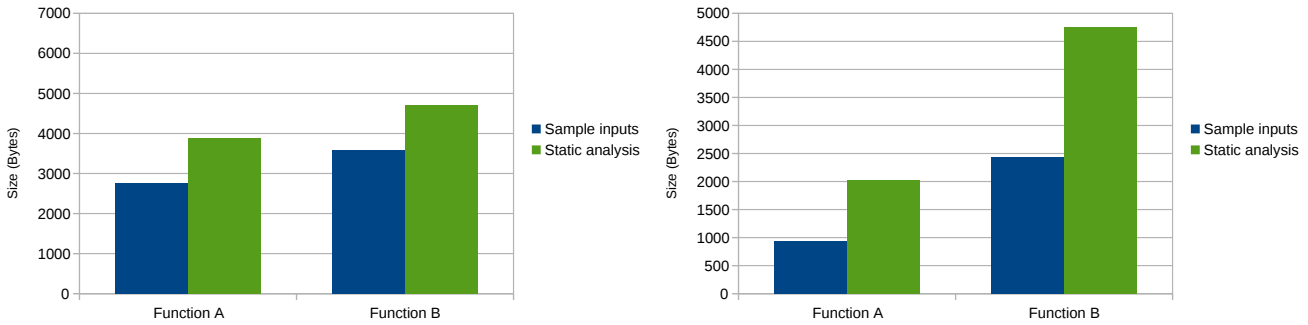
Figure 8: First call with full data, including compilation time.
FastR baseline JIT-compiled, Sample inputs JIT-compiled, Static analysis AOT-compiled

Figure 8 shows that even when including compilation time there is the potential for an overall performance improvement for very large input sizes, as the time for the baseline (interpreted) grows much faster than the time for AOT-compiled execution. As the compilation time is constant and not dependent on the input size, a break-even point for the input size exists. Above this point it will always be advantageous to AOT-compile a function before executing it, assuming the slight reduction in peak performance is not an issue.

4.3 Time spent in the analysis

The time spent on analysis for either function is completely input independent, with an average of 17ms for Function A and 26ms for Function B. The analysis is run 3 times in a row[7] to ensure that all specializations have completed their AST node rewrites, however most of the elapsed time is spent in the first run, with < 2ms for each subsequent analysis run.

4.4 Generated code



(a) Compilation time

(b) Size of compiled code

Figure 9: Compilation time and size of compiled code

[7] in many cases 1 or 2 or passes suffice

While the reasons for the slower peak performance are not exhaustively known, the difference in code size indicates that different compiler graphs are generated when using AOT-compilation, leading to larger compiled code and slower execution of the executed code. This is likely due to impreciseness introduced during the analysis, and profiles that are not set up in exactly the same way. A visual representation of the compiler graphs in §A.1, and Figure 9 summarizes the difference in code size and the effect this has on the overall compilation time. It is possible that implementing a more precise analysis could reduce the resulting code size and thus restore some of the lost peak performance.

4.5 Threats to validity, future work

While some of the restrictions and limitations have been touched upon in previous chapters, not all of them have been mentioned. One main concern is that the current implementation suffers from relatively invasive changes that touch many parts of FastR. Due to the large number of changes needed, the current implementation is not complete: Only a small subset of the R language can be used, and the implementation is a proof of concept rather than a feature ready to be integrated into FastR. The current implementation is hard to maintain, as changes to FastR cannot be automatically applied to the simulation type counterparts. For a full-fledged implementation, large parts of the code translation and generation would need to be automatically generated. In addition, the static analysis is incomplete in that it currently does not allow abstract interpretation of nested loops. In order to support nested loops, a more general abstract interpretation implementation is needed, with access to not just method arguments and class fields, but variables in general that are defined outside the inner loop. Another concern is that the current implementation assumes that no undesirable side effects will be triggered during the static analysis / execution using simulation types, i.e. there are no checks for side effects. This assumption is true for the current small and limited examples, but not for general R code. Also unsolved is the problem of determining which functions to target for AOT-compilation and when. It is possible to build a working heuristic for this, however currently the names of the functions that should be AOT-compiled are passed on the command-line as FastR options. Finally, a comparison of the currently attained performance with a more aggressive OSR implementation would be interesting: interpreted FastR statements may execute long-running Java loops, and these loops could be more efficiently optimized if there was FastR support for OSR within Java loops[8]. While this would not enable cross-loop optimizations between R statements, it would enable faster execution of e.g. the *sum* function without needing the current static analysis approach.

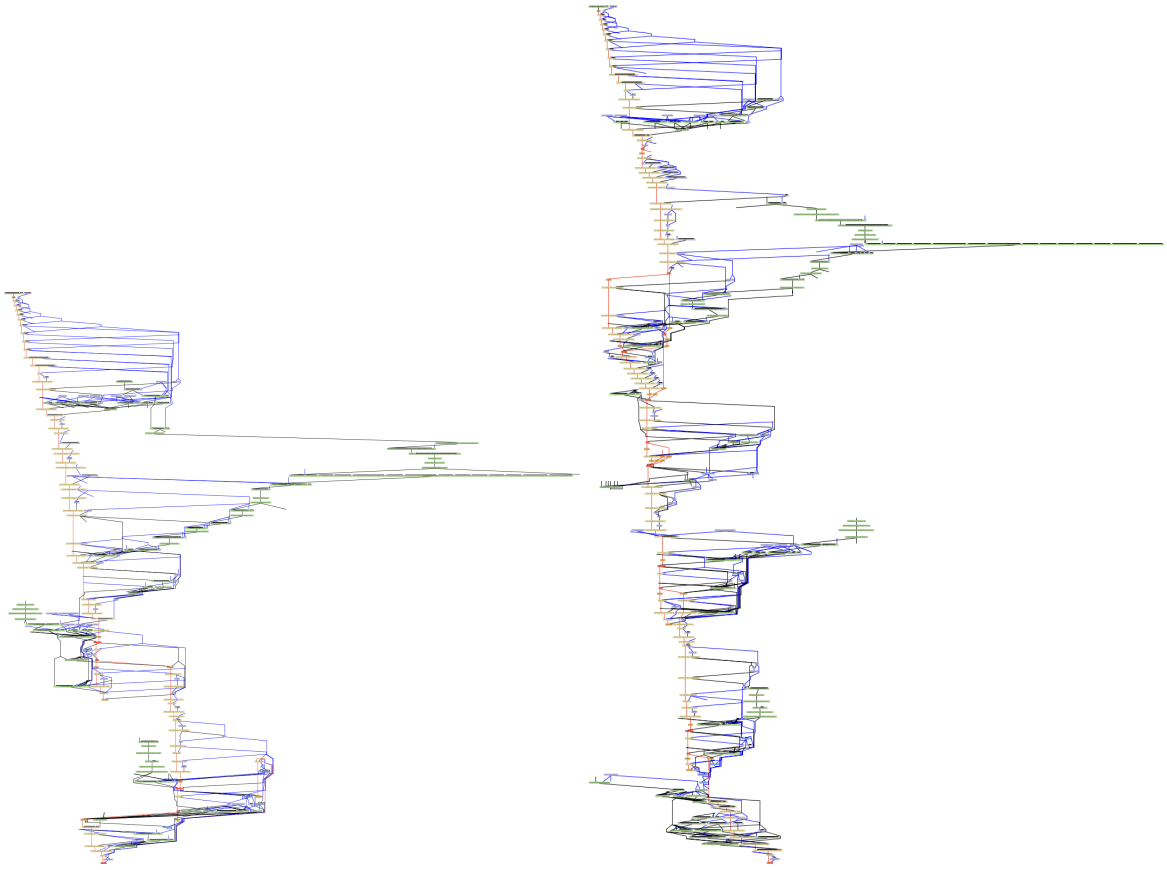
[8] as the JVM may not have enough information to effectively optimize the loop.

5 Conclusion

This project shows that AOT-compilation of functions within FastR is feasible, even though FastR relies heavily on profiling information and specializations that are generally not amenable to AOT-compilation. It shows that there is room for improvement on startup times, and that first-call performance of functions can be significantly improved. The analysis in its current form is precise enough for the functions and inputs used for the benchmarks, and results in compiled code that is up to 4x faster than regularly interpreted code when first calling a function.
This technique may be of interest when first-call performance, remote execution, and more performance predictability trump the need for peak performance. However, significant effort would be necessary to integrate this approach fully with FastR. Many of the modifications that were done manually for this project would need to be generated and applied automatically, possibly through the creation of DSL that for some of the patterns described in §3.

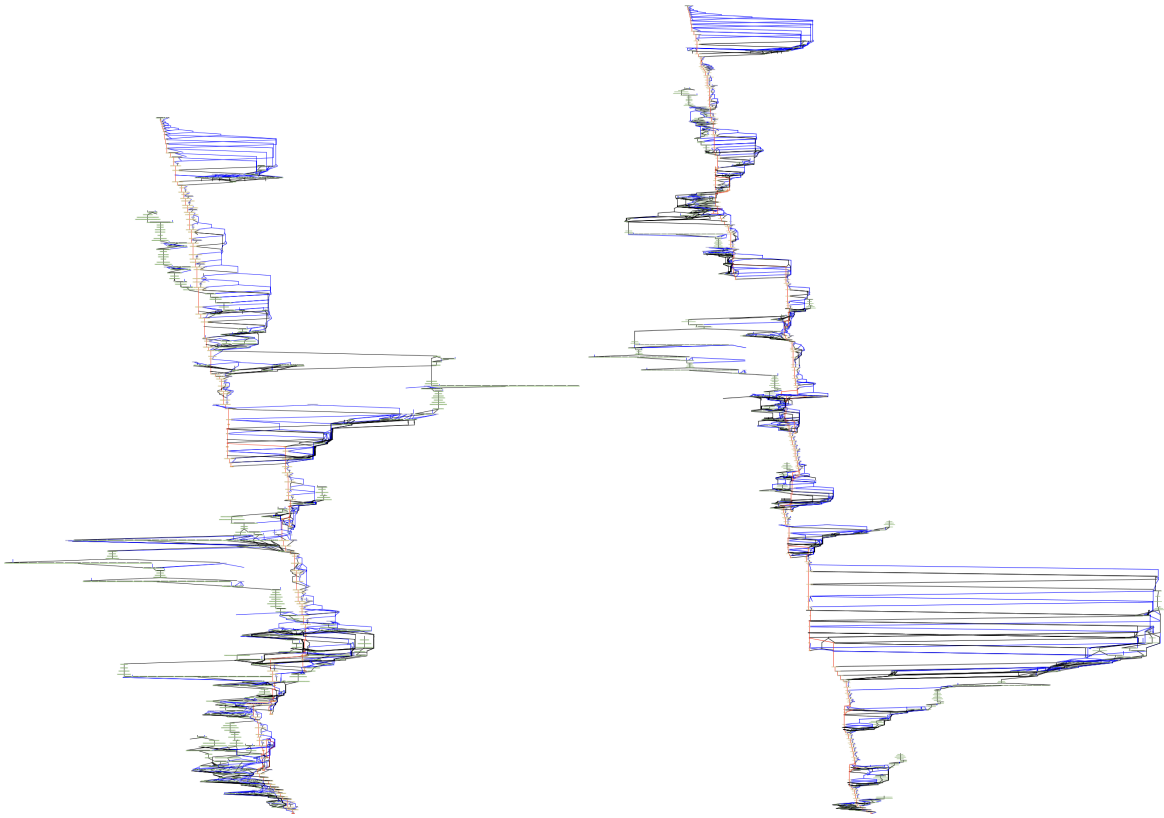
A Appendix

A.1 Comparison of compiler graphs



(a) Sample inputs (b) Static analysis

Figure 10: Comparison of compiler graphs for function A



(a) Sample inputs (b) Static analysis

Figure 11: Comparison of compiler graphs for function B

Figure 10 and Figure 11 show the different compiler graphs generated by Graal for the two functions A and B, depending on whether regular JIT-compilation or AOT-compilation using static analysis is used. From the visual shape it can be seen that the graphs are generally similar, with the graph generated using AOT-compilation being larger overall, resulting in a larger compiled code size and slower peak performance.