Parallel Incremental Whole-Program Optimizations for Scala.js

Sébastien Doeraene Tobias Schlatter
École polytechnique fédérale de Lausanne, Switzerland
sebastien.doeraene@epfl.ch schlatter.tobias@gmail.com

Abstract

Whole-program optimizations are powerful tools that can dramatically improve performance, size and other aspects of programs. Because they depend on global knowledge, they must typically be reapplied to the whole program when small changes are made, which makes them too slow for the development cycle. This is an issue for some environments that require, or benefit a lot from, whole-program optimizations, such as compilation to JavaScript or to the Dalvik VM, because their development cycle is slowed down either by the lack of optimizations, or by the time spent on applying them.

We present a new approach to designing incremental whole-program optimizers for object-oriented and functional languages: when part of a program changes, only the portions affected by the changes are reoptimized. An incremental optimizer using this approach for Scala.js, the Scala to JavaScript compiler, demonstrates speedups from 10x to 100x compared to its batch version. As a result, the optimizer's running time becomes insignificant compared to separate compilation, making it fit for use on every compilation run during the development cycle. We also show how to parallelize the incremental algorithm to take advantage of multicore hardware.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental compilers, Optimization

Keywords incremental, whole-program optimizations

1. Motivation

Whole-program optimizations are powerful, and are used in most compilers nowadays, ahead-of-time and just-in-time alike. However, as their name implies, they require knowledge about the entire program. In other words, they are not modular, since changing one part of the program might require reoptimizing other, seemingly unrelated parts. For example, consider these two simple Java classes:[1]

```
class A {
  int x;
  A(int x) {
    this.x = x;
  }
  int getX() {
    return this.x;
  }
  void print() {
    System.out.println(getX());
  }
}

class B {
  void foo(A a) {
    a.print();
  }
}
```

While optimizing `B.foo`, we might decide to inline the body of `A.print()`, because it is so short. To do this, though, we need two pieces of global knowledge:

- The fact that no subclass of `A` overrides `print()`, otherwise inlining would break dynamic dispatch.
- The actual body of `A.print()`.

If `A` and `B` are stored in separate compilation units (files), it might be that `A` is recompiled, but `B` is not. If the body of `A.print()` changes from one compilation to the next, the optimized version of `B.foo` is outdated. Another, more subtle scenario is the *addition* of a new class `C` extending `A` and overriding `print`. This would also invalidate the optimized version of `B`. In general, other kinds of knowledge may be needed, and all kinds of scenarios might invalidate many sorts of optimizations. That is why whole-program optimizers typically work in batch mode: they start the optimization of an entire program from scratch on every compilation run, even if only a small part is changed.

This is, however, wasteful. An optimizer should ideally only reoptimize methods impacted by the change, instead of the entire program. For other methods, it could reuse the

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2984013

[1] We omit visibility modifiers, since they are irrelevant from an optimizer's point of view. Everything is implicitly public.

optimized version from a previous run. This is similar to compilers with separate compilation: they reuse compiled versions of source files that have not changed to speed up every compilation run. Analogously, such an incremental behavior could dramatically reduce the time spent on optimizing the program during the development cycle.

But why is it important to have a fast optimizer? Of course we want a fast resulting program for production, so we want to run whole-program optimizations at that time. Surely we do not need to do so every time we save-compile-test? We can be content with the compiled, non-optimized program for iterative development, can't we?

Indeed, in most setups, this is enough, and we believe that is why incremental whole-program optimizations have been but scarcely addressed so far. However, in some cases, this is not acceptable. For example, the compilation of Scala to Android *requires* whole-program optimizations to be performed on every run, because of limitations imposed by the Dalvik VM.

In this paper, we focus on the case of Scala.js, which is a cross-breed of two languages whose development cycles are improbably dissimilar. On the one hand, Scala, with its slow compiler, its huge standard library with many indirections and higher-order methods, and a virtual machine with slow startup times, yields a relatively slow cycle where we expect the compiler (or the IDE) to detect as many errors as possible. On the other hand, JavaScript has libraries that are as small as possible, and a community expecting instantaneous save-refresh cycles. The problem is, with a non-optimized huge standard library compiled to JavaScript, the JavaScript virtual machines take noticeable time just compiling it every time the program is run. If optimizations are fast enough and bring enough speed improvements to the interpreter startup time, it is worth doing them on every development cycle. This is precisely the case in Scala.js, with incremental whole-program optimizations.

The approach can also be beneficial to other environments. In general, we believe that any rich language with big libraries compiling to a constrained environment can benefit from incremental whole-program optimizations.

Writing incremental optimizers is not a new idea. For example, Chambers et al. [4] developed a very general framework which provides fully automatic incremental reoptimizations of the program. While their ultimate goal was to reduce the time spent when reoptimizing the whole program, they do not measure nor discuss the actual running time of their compiler. What they measure instead is the selectivity of their incremental framework, i.e., how *accurate* it is, and they show that it reoptimizes very few methods. We think that this is not the most appropriate metric: even a highly accurate incremental framework is useless if deciding what to reoptimize takes longer than reoptimizing the whole program. Consider the extreme case, where the incremental optimizer first optimizes the whole program to determine what has changed with respect to the previous run. Such an incremental optimizer would be optimally accurate, yet slower than a batch optimizer. This absurd situation is of course not what we have in the case of Chambers et al.'s framework. It was implemented in the Vortex optimizing compiler and applied during the development cycle of Cecil programs, including Vortex itself. As mentioned by Dean in [6], the framework was selective enough to be used for day to day development, suggesting that it was indeed faster than a batch optimizer. However, we believe that, in aiming for the wrong evaluation metric, they might have missed opportunities on making their framework even better.

In contrast to those previous efforts, we directly focus on evaluating and reducing the running time of the entire pipeline: the detection part plus the reoptimization part. This overall running time is ultimately what the developers care about. To achieve this, we sacrifice the completely automatic nature of the incremental analysis and instead developed a methodology for designing *modular* incremental optimizers. They are modular in the sense that the change detection algorithm is isolated from the optimizer's heuristics and mechanisms and vice versa. It is therefore easy to reason about the correctness of the incremental analysis on the one hand, and the optimizer's logic on the other hand.

1.1 Contributions

In this paper, we make the following contributions:

- We introduce a new approach to designing modular incremental whole-program optimizers for object-oriented and functional languages. Such optimizers detect what methods of the program need to be reoptimized when some parts of the program change. The approach is based on knowledge queries, which we introduce in Section 2. They automatically create a *modular* interface between changes in the program and the optimizations they invalidate.
- We show how the approach accommodates a variety of type-based whole-program optimizations for object-oriented and functional languages in Section 3: inlining with static and dynamic dispatch, elimination of subtyping checks, scalar replacement, and closure elimination.
- Using this approach, we implemented an incremental whole-program optimizer for Scala.js [7]. Results presented in Section 6.1 show that incremental runs are 10 to 100 times faster than batch runs.
- We show how the general algorithm can be made parallel in Section 5. Section 6.2 shows that the parallel algorithm scales with the number of threads.

2. Knowledge Queries

As hinted in the previous section, the difficult part of an incremental optimizer is to detect which methods need to be reoptimized when certain parts of the compiled (non-optimized)

program change, i.e., tracking dependencies between optimized methods and the program.

We introduce our approach by studying the simple case of inlining static methods in Java. Static methods are always referred to with their full name, and do not need polymorphic dispatch resolution. In Section 3, we will progressively lift those restrictions, and show how other kinds of optimizations map into our approach.

2.1 A Pure and Restricted API

Our solution is based on one key idea: provide a restricted API through which the optimizer can query facts about the whole program. We call the functions in this API *knowledge queries*. They must take immutable arguments and return immutable results, which may depend only on the arguments and the program. Knowledge queries are therefore functions of their arguments and the program.

Our optimizer works on a per-method basis. An instance of the optimizer is created for every method, and is tasked to work solely on this method. Hence, we name it a method optimizer. To gather any information about the program other than the method itself, a method optimizer can only use knowledge queries. Its result (the optimized method) must therefore be a function of two things:

- the non-optimized method, and
- the results of knowledge queries.[2]

For example, consider a variant of the program from the introduction where everything is static, and with an additional method `C.bar`. Although our implementation optimizes Scala.js code, we stick to Java code examples for two reasons. First, for familiarity; and second, because the optimizer's intermediate representation is closer to Java than Scala anyway.

```
class A {
  static int x;
  static int getX() {
    return A.x;
  }
  static void print() {
    System.out.println(A.getX());
  }
}
class B {
  static void foo() {
    A.print();
  }
}
class C {
  static void bar() {
    System.out.println("bar: " + A.getX());
  }
}
```

The method optimizer for `B.foo` initially knows nothing about the program, except the body of that method. As it processes the body, it finds the call `A.print()`, which it may decide to inline. Typically, inlining decisions are based on properties of the *target* method as much as the caller. The optimizer already knows about the caller, but not the target.

Since static methods are linked statically, we know that the target method must be the method `print` in the class `A`. But that is all the optimizer can know about the target (since it does not know the program) without calling a knowledge query. Suppose we base our inlining decisions on the body of the target method. In this case, we need the following query:

```
Tree getMethodBody(MethodID method);
```

where `MethodID` is the fully qualified name of a method in the program (here, `"A.print"`). Once the optimizer knows the body of `A.print`, it can decide whether to inline it or not (e.g., based on its size). Smarter heuristics can be used, such as those developed by Sewe et al. [16].

Note that if (and only if) it decides to inline `A.print`, it will process that method's body for optimizations too. While doing that, it will encounter the call `getX()`, which it will also consider for inlining. It will therefore also invoke the query `getMethodBody("A.getX")`. It will not do so if it decides not to inline `A.print`. The set of knowledge queries performed by an optimizer may therefore depend on its optimization decisions: it is not an inherent property of the program.

2.2 An Automatic Dependency Tracker

Since the result of optimizing a method m must be a function of its non-optimized body and the knowledge queries, the latter are a natural and automatic dependency tracker, which we can use for incremental reoptimization. If the non-optimized version of m and the results of all knowledge queries performed by its optimizer are the same for two successive versions of the program, then the result of the optimizer must be the same too. Hence, we need not reoptimize m for the second run of the incremental optimizer: we may reuse the optimized method from the previous run.[3]

This forms the basis for the incremental optimizer based on knowledge queries:

- While optimizing a method m, we log all the knowledge queries invoked by the optimizer, and register them as dependencies for the optimized m.
- On the next run of the incremental optimizer, we detect which knowledge queries have different results than in the previous run, and we invalidate all the optimized methods that depend on them.

[2] Pragmatically, it could depend on other factors such as randomness or time spent. The point is that it cannot depend on the program other than through knowledge queries.

[3] If we accept randomness and other independent factors, we must reformulate this as: the result for the previous run is *a* possible result for the second run as well, which is also sufficient to allow us to reuse it.

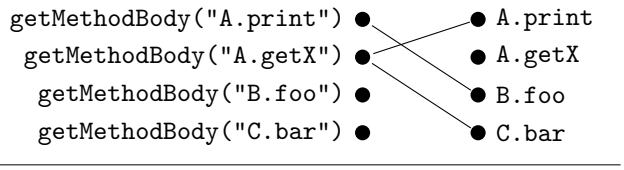

Figure 1. Dependency graph if `A.print` is not inlined

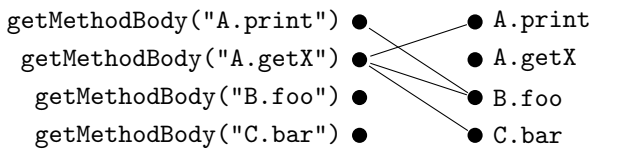

Figure 2. Dependency graph if `A.print` is inlined

The recorded dependencies form a bipartite graph with two kinds of nodes: knowledge queries (including their arguments), and optimized methods.

Figure 1 shows the dependency graph that we get for the program of the previous section if the optimizer for `B.foo` decides not to inline `A.print`. If it does inline `A.print`, then we get the graph of Figure 2, in which there is an additional edge between `B.foo` and `getMethodBody("A.getX")`.

2.3 Program Changes

We still miss one piece of the algorithm: how do we detect which knowledge queries have different results? For this, we compute program changes, and their impact on knowledge queries.

A *program change* is a difference between the program in the current run compared to the previous run. Program changes can be of any granularity, but in practice we only consider class-level and method-level changes. Tracking dependencies at the instruction level is useless for one simple reason: as programs get bigger, they contain arbitrarily more classes and more methods, but the size of each method tends to be bounded, due to established programming best practices. This means that we need not scale with the size of methods, but rather with the number of classes and methods in the program. Section 4 shows how we compute the changes we are interested in.

Intuitively, a program change *impacts* a knowledge query if it changes the result of that query. Hence, for each change, we have to compute which queries it impacts. All the methods depending on those queries (which we get from the dependency graph) must be invalidated.

A little bit more formally, we say that a program change *impacts* a knowledge query if and only if there exists a program P such that the result of that query is different for P than on the program P' obtained by applying the change to P. By this definition, and because of the commutativity of difference, a program change always impacts the same set of queries as its reciprocal.

The changes we need to detect are derived from the queries used by the method optimizer. So far, we have only one query: `getMethodBody`. The only program change we need to cover is straightforward:

- Change the body of a method `C.m`.

which always impacts only `getMethodBody(C.m)`. We will see in Section 3 that not all knowledge queries translate that straightforwardly to program changes.

Unlike the dependency tracking between the optimizer and the knowledge queries, the relation between program changes and their impacts is not automatic: it must be reasoned about manually. However, the big advantage is that the details of the optimizer, which are bound to be complex, are completely abstracted away from that reasoning by the knowledge queries, which are automatic. The knowledge queries therefore create a truly *modular* interface between the optimization logic and the incremental logic.

We conclude this section with the global view of the whole incremental algorithm. It has two phases per run:

- Invalidation phase: diff the program compared to the previous run to isolate *program changes* (see Section 4). For each program change, compute the set of *impacted* knowledge queries (see Section 3). Follow the edges in the dependency graph to invalidate all the methods depending on these queries.
- Optimization phase: run the method optimizer for every invalidated method. First remove all edges in the graph for this method. Then, optimize its body and record all invoked knowledge queries as edges in the graph.

3. Knowledge Queries for OO and Functional Languages

In the previous section, we introduced knowledge queries with the case of inlining static methods only. We now add language features and optimizations with two goals in mind:

- show how knowledge queries guide the design of incremental optimizations, and
- show how a variety of whole-program optimizations for object-oriented and functional languages fit in the knowledge query approach.

First, we add language features, while still studying their impact on inlining only, until we can handle all the features of a JVM-style object system (which is what we get from the Scala.js compiler): single inheritance, interfaces, dynamic dispatch and run-time instance tests. Then, we introduce support for other kinds of whole-program optimizations: elimination of run-time type tests, scalar replacement and closure elimination.

3.1 Inheritance

In Section 2, we have silently omitted to consider *inherited* static methods. In Java, as well as most object-oriented

languages, static methods defined in a superclass `A` can be called on a class `B` when `B` extends `A`. Most languages also allow to *reintroduce* a static method with the same name in a subclass. When calling `B.foo`, the actual target is looked up in the parent chain of `B` on a first match basis. Reintroduced methods therefore shadow methods coming from the superclasses.

For example, consider the following snippet:

```
class A extends Object {
  static int foo() {
    return 3;
  }
}
class B extends A {
}
class C extends B {
  static int foo() {
    return 5;
  }
}
void bar() {
  println(A.foo());
  println(B.foo());
  println(C.foo());
}
```

in which the method `bar` calls the method `A.foo` twice then the method `C.foo` once. The call `B.foo()` resolves to `A.foo`, and it is therefore not possible any more to identify the actual target only looking at the call site. Some global knowledge is required.

There are other cases where static call resolution applies:

- Non-virtual methods in C++.
- `super` calls and constructor calls in Java and Scala.
- In an optimizer, we can use static resolution even if the language prescribes dynamic resolution, provided we know the exact class of the receiver (e.g., because it is allocated with `new` within the scope of the optimizer).

We will now extend the support for incremental inlining from the previous section to handle such resolutions.

3.1.1 Eliciting Knowledge Queries

The first step is to determine what knowledge queries are necessary. To do this, we simply update the method optimizer. When it needs some information about the whole program, we introduce a new knowledge query.

Here, we want to inline calls such as the following:

```
T.m();
```

Because of inheritance, `m` could be declared in a a superclass of `T`. Finding the exact (unique) target depends on whole-program knowledge. We therefore introduce a new knowledge query:

```
MethodID resolveStaticCall(
    ClassID class, String methodName);
```

The optimizer can use this knowledge query to retrieve the actual target of a static call. This query will typically be followed by the existing query `getMethodBody(target)` to decide whether and how to inline that target.

3.1.2 Impact of Program Changes on Queries

We now need to efficiently find queries that are impacted by program changes. Let us start with an example. In one run of the optimizer, the program looks like this:

```
class A extends Object {
  static void foo() {
    println("A");
  }
}
class B extends A {}
class C extends B {}

class Main {
  static void main() {
    C.foo();
  }
}
```

While optimizing `Main.main`, we consider the call `C.foo()` for inlining. The semantics of static calls tell us that the actual target method is `A.foo`. Therefore, `resolveStaticCall(C, "foo")` returns `"A.foo"`.

Now, in the next run, we *add* the method `B.foo`:

```
class B extends A {
  static void foo() {
    println("B");
  }
}
```

Assuming we have a perfectly accurate incremental *compiler*, this change does not impact recompilation of `Main`, since the public API of `C` has not changed.[4] However, it must prompt reoptimization of `Main.main`, since the actual target method changes: now it is `B.foo`. In other words, the result of the knowledge query `resolveStaticCall(C, "foo")` has changed from `"A.foo"` to `"B.foo"`. Hence, the addition of a new method `B.foo` impacts this knowledge query. In general, adding a method `X.m` will impact `resolveStaticCall(Y, "m")` for all classes `Y` extending `X`, directly or indirectly. Since no other method in the program asked for `resolveStaticCall(C, "foo")`, the addition of `"B.foo"` does not trigger reoptimization of any other part of the program.

Removing a method will have the same impact, since it is the reciprocal of adding it.

In a program with inheritance, we identify the following possible core changes:

- Change an existing method's body

[4] This depends on the format and specifications of the compiled files. For example, it is not true for `.o` files in C++. For the `invokespecial` instruction of the JVM, it is true.

- Add/remove a method in an existing class
- Add/remove an empty class without child classes (another pair of reciprocal changes)

Note that other changes can be represented as composition of these changes:

- Adding a non-empty class is equivalent to adding it empty, then adding methods.
- Similarly, removing a non-empty class is equivalent to removing its methods, then removing it.
- Changing a class' parent, i.e., moving it in the hierarchy, is equivalent to deleting the class and all its subclasses from the previous parent, and adding them back to the new parent.

Further, note that when a class is removed (including when it is moved), all its subclasses are removed as well.

We have already seen the impact of the two first core changes. Adding an empty class `C` that does not have any child class does not impact either `getMethodBody` (since it does not have any method) nor `resolveStaticCall` (because no existing code could possibly have a call `C.m()`, since `C` did not exist in the previous run). It follows that removing an empty class, which is the reverse operation, does not have any impact either.

As a recap, we have the following:

- Changing the body of `C.m` impacts `getMethodBody("C.m")`.
- Adding or removing a method `C.m` impacts `resolveStaticCall(D, "m")` for all `D extends C`.

That is it. With a simple two-step design, we added support for inheritance. First, we listed knowledge queries needed by the optimizer. Then, we identified which program changes can impact these queries.

3.2 Polymorphic Dispatch

So far, we have limited our discussion to static calls. However, Java, Scala, and other object-oriented languages have virtual calls, where dynamic dispatch is required. The exact target of a call is not known at compile-time, since it depends on the run-time type of the receiver. To support inlining with polymorphic dispatch, we again find out what knowledge queries are needed, and what program changes impact them.

3.2.1 Eliciting Knowledge Queries

When encountering a dynamic call of the form `x.m()`, the optimizer has to determine the target of that call. Unlike with static calls, there can be multiple targets, if `x` is of a type `C` that has several subclasses overriding `m`. We therefore need the following new knowledge query:

```
Set<MethodID> resolveDynamicCall(
    ClassID class, String methodName);
```

With the set of possible targets, the optimizer can decide whether or not to inline (e.g., if the set is a singleton). As was the case with static calls, it can follow up with `getMethodBody` to obtain the body of a given target to inline.

3.2.2 Impact of Program Changes on Queries

Consider the following base program with an instance method `foo`:

```
class A extends Object {
  void foo() {
    println("A");
  }
}
class B extends A {}
class C extends B {}
```

and consider the calls `a.foo()` and `c.foo()`, with `a` (resp. `c`) statically typed as `A` (resp. `C`). The corresponding knowledge queries, `resolveDynamicCall(A, foo)` and `resolveDynamicCall(C, foo)`, both return the singleton {`"A.foo"`}.

We now add a method `B.foo`. Similarly to the query `resolveStaticCall`, `resolveDynamicCall(C, foo)` is impacted, since its result becomes {`"B.foo"`}. However, in this case the query `resolveDynamicCall(A, foo)` is also impacted, with a result of {`"A.foo", "B.foo"`}. This follows from the fact that `B <: A`. So a value statically typed as `A` can hold an instance of class `B`.

In general, adding (or removing) a method `X.m` impacts the queries `resolveDynamicCall(Y, m)` for all `Y` such that either `Y <: X` or `X <: Y`.

3.3 Interfaces

The addition of interfaces (without default methods) complicates dynamic calls. Now, in a dynamic call `x.m()`, `x` can be statically typed as an interface `I`. Consider the following program:

```
interface I {
  void foo();
}
class A extends Object {
  void foo() {
    println("A");
  }
}
class B extends A {}
class C extends B implements I {}
```

and the call `x.foo()` where `x` is statically typed as an `I`. The result of the query `resolveDynamicCall(I, "foo")` would return the singleton {`"A.foo"`}.

Let us now add a method `B.foo`. This changes the result of the query from {`"A.foo"`} to {`"B.foo"`}, even though neither `B <: I` nor `I <: B`. Here, the query is impacted because there exists a subclass `C` of `B` that implements `I`.

In general, adding (or removing) a method `X.m` now also impacts `resolveDynamicCall(I, m)` for all `I` such that

there exists a class `Z` such that `Z <: X` and `Z <: I`. Taking a step back, the rule for classes that we saw in the previous section is a special case of this one. We therefore combine both rules as one: adding (or removing) a method `X.m` impacts `resolveDynamicCall(Y, m)` for all classes and interfaces `Y` such that there exists a class `Z` such that `Z <: X` and `Z <: Y`. This looks heavy to compute, but we can easily reformulate it as follows: for all subclasses `Z` of `X`, for all ancestors `Y` of `Z`, `Y.m` is impacted. If we maintain a data structure that allows us fast access to all subclasses of a class, and all ancestors of a class, the computation becomes straightforward.

We are not done with interfaces, though. The addition of interfaces to the language also introduces a new kind of program change:

- Add/remove an interface `I` to the ancestors of a class `C` (pair of reciprocal changes).

Adding `I` to the ancestors of `C` implies that, now, `C <: I`. A variable `x` of type `I` can therefore hold a value of class `C`, where previously it could not. In terms of dynamic calls, this means that the target of `x.m()` can change for any `m`. This change (and its reciprocal) will thus impact the queries `resolveDynamicCall(I, m)` for all `m`.

Note that it is not necessary to track the methods defined in the interface itself with this approach.

3.4 Other Object-Oriented Features

There are a couple of other typical object-oriented features that do not require any additional support.

Multiple inheritance is entirely covered by the above treatment of interfaces, as far as knowledge queries are concerned.

Overloading is a compilation issue. When they reach the optimizer, overloaded methods have already been disambiguated, either with mangled names (e.g., in C++ or the Scala.js IR) or because they are identified by their full signature (such as on the JVM).

Similarly, operator overloading is transformed by the compiler into method calls, and is therefore covered.

3.5 Eliminate Subtyping Checks

Now that we have a full JVM-like language, we can move on to support other kinds of optimizations, besides inlining. We begin with eliminating runtime subtyping checks, i.e., `instanceof` and casts. Due to other whole-program optimizations, such as inlining, we can be left with tautological subtyping checks, such as

```
interface Foo {}
class Bar implements Foo {}

Bar x = ...
(Foo) x
```

Because `Bar <: Foo`, the cast is redundant, and can be eliminated, giving `x`. Similarly, a test `x instanceof Foo` can be optimized as `true` (taking care of preserving the side-effects of computing `x`). However, to do this, we need to know that `Bar <: Foo`, which is global knowledge of the program. It must therefore be requested as a knowledge query.

3.5.1 Eliciting Knowledge Queries

When encountering `x instanceof Foo`, with `x` of type `Bar`, the optimizer must test whether `Bar <: Foo`. The obvious knowledge query is therefore the following:

```
boolean isSubclass(ClassID subclass,
    ClassID superclass);
```

with the understanding that a `ClassID` can also refer to an interface.

3.5.2 Impact of Program Changes on Queries

The query `isSubclass(subclass, superclass)` only depends on the list of all ancestors (classes and interfaces) of `subclass`. More specifically, whether `superclass` is in this list or not. Because the parent chain of a class cannot change (it would be removed and added instead), only interfaces can be added to or removed from the ancestor list. We extend the program change we introduced with interfaces:

- Add/remove an interface `I` to the ancestors of a class *or interface* `J` (pair of reciprocal changes).

which now also impacts the query `isSubclass(J, I)`.

3.6 Scalar Replacement

Scalar replacement, also known as stack allocation, is an optimization that replaces a reference value (to an object allocated on the heap) by a set of values for all fields of the given object. For example, replacing:

```
class Point {
  double x, y;
  Point(x, y) {
    this.x = x;
    this.y = y;
  }
  double abs() {
    return Math.sqrt(x*x + y*y);
  }
}
void foo(double y) {
  Point point = new Point(5, y);
  println(point.x);
  println(point.abs());
}
```

by

```
void foo(double y) {
  double point_x = 5;
  double point_y = y;
  println(point_x);
  println(Math.sqrt(point_x*point_x +
      point_y*point_y));
}
```

Note that we inlined the call to `point.abs()`. If we cannot inline `point.abs()`, the optimization is canceled, since the

`Point` must be allocated to call `abs`. Instead, we could use partial escape analysis as developed by Stadler et al. [18].

This optimization improves several aspects:

- Memory consumption and GC pressure, because less objects are allocated
- Execution speed, because less pointer indirections are involved
- It is an enabler for other optimizations, because we can often have more precise static information on the fields (e.g., here, we can constant-fold `point_x`).

Scalar replacement obviously needs global knowledge: what fields are defined in the class `Point`, as well as the body of its constructor. We can use knowledge queries to introduce this optimization in an incremental framework.

3.6.1 Eliciting Knowledge Queries

Although scalar replacement has several implications down the line, its need for global knowledge is actually confined to the `new` invocation. Indeed, once the allocation is replaced by individual variables, the fact that `point` has been scalar-replaced into `point_x` and `point_y` becomes part of the local state of the optimizer.

To keep things minimal, we decompose an allocation such as `new C(args)` into the allocation of the object of class `C` itself, and the call to its constructor. The latter is a standard application of a static call, as described in Section 3.1. The interesting part is the allocation. To replace it, we only need to know its fields. The body of the constructor itself is not needed, as it is part of the static call. We therefore derive the following knowledge query:

```
List<Field> getScalarReplacement(
    ClassID class);
```

where `Field` is a description of a field, with its type and potentially other properties, such as mutability.

3.6.2 Impact of Program Changes on Queries

We introduce the following program change:

- Change the fields of a class `C` (including those inherited)

which impacts the query `getScalarReplacement(C)`.

3.7 Closure Elimination

Closure elimination is an important optimization for languages with higher-order functions, obviously including functional languages. We do not need any other query (and therefore no other program change) to support this optimization. Indeed, closure elimination derives from inlining, local constant propagation, and either beta-reduction or scalar replacement, depending on whether the IR supports closures directly or encodes them as anonymous classes. All of these optimizations are either local optimizations, or we have already shown how to support them. Hence, closure elimination is trivially supported by our approach.

4. Diffing the Program Between Runs

In the previous sections, we showed how knowledge queries apply to a variety of whole-program optimizations. To do so, we relied on *program changes* as small differences between two versions of a program. As a recap, here are the various kinds of changes we relied upon:

- Change the body of a method `C.m`
- Add/remove a method `C.m`
- Add/remove an empty class `C` without child classes
- Add/remove an interface `I` to the list of ancestors of a class or interface `J`
- Change the fields of a class `C`

If the earlier steps of the compiler were incremental themselves, they could communicate these changes directly to the incremental optimizer. However, this is not the case in Scala.js: the smallest unit of change that the optimizer receives is a compiled class (or interface), each being stored in a compiled file. In this section, we sketch how to diff two object-oriented programs so that we can derive a list of the above changes.

To guarantee the correctness of the incremental optimizer, we need to make sure that we produce an *exhaustive* list of program changes. If we miss one, we might not detect the impact of changes on some knowledge queries, which in turn, would cause some optimized method not to be invalidated when they should, making the optimizer unsound. Note that it is not necessary for correctness to produce a *minimal* list of program changes, i.e., we are allowed to emit more changes than necessary, although that would invalidate more methods than required. We do not argue for the precision of our diff, and instead focus on the overall running time in Section 6.

Since every program change is related to some class or interface `C` or `J`, we can divide and conquer the problem by class/interface. For each class or interface in the program, we seek to produce an exhaustive list of program changes related to it.

To compute our structured diff, we maintain two data structures. First, a map from each class/interface `J` to the set of its ancestors. Second, a tree of the classes in the program mirroring the class hierarchy. Each class keeps track of:

- its parent and child classes, forming the tree structure
- its methods and their bodies
- its fields, including inherited ones

Note that, together, those two data structures completely define the entire program. Computing the program changes caused by changes to these structures therefore takes all possible changes to the program into account. Other than in the ancestor map, interfaces and their member methods are not part of this program representation, because they are

neither used by the optimizer nor the code generator that follows it, and are therefore irrelevant at this point.

If mixed with other changes, additions and removals of classes are hard to get right, especially because of classes that move around the hierarchy. Therefore, the diffing algorithm uses 4 main steps:

1. Class deletions: remove classes that existed in the previous run, but do not exist in the new version of the program. Recall that this includes classes that are moved in the hierarchy.
2. Changes to the sets of ancestors.
3. Class changes: add and remove methods, change method bodies, and change fields.
4. Class additions: add classes that did not exist in the previous run, but do exist in the new version of the program. Again, this includes classes that have moved.

This separation also allows for a simple and efficient implementation of the batch mode (the first run, when we come from an empty program): simply run the last step.

Note that there is no step dealing with interface addition and removal. It turns out this is not necessary, since no program change is dependent on those changes.

Steps for Class Additions and Removals. The first step is a post-order traversal of the hierarchy tree. For each deleted class `C`, we first remove all its methods (emitting "Remove a method `C.m`" changes), then remove the class itself (emitting "Remove an empty class `C` without child classes"). Since classes moving around the hierarchy are considered deleted then added, we know that if class `C` is deleted, so are all its subclasses. Therefore, in a post-order walk, it indeed has no child classes anymore. There cannot be any other program change for `C`, so we have exhaustively listed all program changes for `C`.

Similarly, the last step for class additions is a pre-order traversal of the hierarchy tree. By a similar argument, we list "Add an empty class `C` without child classes" and "Add a method `C.m`" program changes for `C`, and no other.

Ancestors Update Step. In this step, we update the sets of ancestors of all classes and interfaces in the program. Since classes are not removed, added or moved during this step, only interfaces can be added to or removed from the sets of ancestors. For each class or interface `J` in the new program, we compute its new set of ancestors by transitively following the direct parent class and/or implemented interfaces. Unless `J` did not exist in the previous program, we diff this set with the old set of ancestors. For each difference `I`, which must be an interface, we emit the program change "Add/remove an interface `I` to the list of ancestors of `J`".

Since the interfaces implemented by a class do not influence its methods nor its fields, a change of ancestors need never emit program changes other than "Add/remove an interface `I`". Therefore the emitted list of program changes for changing the ancestor list is exhaustive.

Class Changes Step. The third step is more complicated, and takes care of all the changes in classes that are neither removed, nor added (nor moved around the hierarchy). Obviously, in this step, no program change "Add/remove an empty class `C` without child classes" is emitted.

For the other program changes, we walk the class hierarchy data structure. For each class node `C` in the class hierarchy, we compute the values of its new constituents, and emit the appropriate program changes. Note that, in this step, the parent class and the list of child classes cannot change.

The new values of the constituents of a class `C` are computed from a mix of information coming from the compiled file for `C`, and other parts of the class hierarchy data structure. By diffing the old and new values, we know what program changes to emit. Here is what happens for each of the three constituents of `C`:

- The parent class and child classes are never modified, since during step 3, no class is added, removed, or moved around the hierarchy.
- The set of methods depends only on the content of the compiled file for `C`. Diffing the set itself emits "Add/remove a method `C.m`" changes, while for the bodies of changed methods, we emit "Change the body of a method `C.m`".
- The list of fields depends on the fields of the parent class in addition to those directly declared in `C`. If the list of fields changes at all, we emit "Change the fields of a class `C`".

Since the list of methods of `C` does not influence fields in `C` and conversely, nor the set of ancestors, there is no other program change that need emitting, therefore we have exhaustively listed the appropriate program changes. Since only those two constituents can change in step 3, the above changes entirely characterize the changes in class `C`, and the algorithm produces an exhaustive list of all the program changes for `C`. Finally, since we process all classes in this way, and since we have initially divided the set of all program changes per class `C` or class/interface `J`, we conclude that we produce an exhaustive list of all the program changes for the entire program.

Note that there are data dependencies between the processing of each class, since some parts of the algorithm read data from the rest of the class hierarchy data structure, which imposes constraints on the order in which we process classes. When processing a class node in the class hierarchy, we perform the following kinds of data manipulations:

- Read the file for that class, and that class only.
- Read data in its parent classes in the class hierarchy.
- Read and write data stored in this class.

In particular, observe that:

- we do not read data from child classes, nor any other classes not in the parent chain, and
- we modify data only in the class node being processed.[5]

Therefore, there are only top-down data dependencies, i.e., from parent classes to their children. A pre-order traversal of the class hierarchy guarantees that each class is processed after all its parent classes, and that all data dependencies are satisfied.

Adding multiple inheritance to the language (or default methods à la Java 8) turns the hierarchy tree into a directed acyclic graph. A class must therefore be processed only when *all* its parents have been processed. Adding or removing a parent class must be viewed as moving the class in the graph, and must therefore be deleted and added again. The algorithm is otherwise unaffected.

5. Parallel Implementation

To take advantage of multicore architectures, we parallelized the incremental algorithm. The two phases of the algorithm (invalidation and reoptimization) must run sequentially with respect to each other, i.e., the invalidation phase must complete before the reoptimization phase can start. However, we can parallelize within each phase. Doing so turns out to be very easy.

5.1 Parallelizing the Invalidation Phase

The first phase is trivial to parallelize. We already established that data dependencies flow top-down in the class hierarchy. Moreover, this phase does not modify the dependency graph. We can therefore parallelize trivially down the inheritance tree (or down the graph if we have multiple inheritance).

5.2 Parallelizing the Reoptimization Phase

The second phase is a bit less easy to parallelize, but not much. We receive a set of methods that need to be reoptimized from the first phase. Recall that every method is processed by a different optimizer instance, which holds only state local to itself. Any knowledge of the program it needs must be requested through knowledge queries. Moreover, the result of every query does not change during one run. We would therefore like to run all optimizer instances in parallel.

Since knowledge queries read data that are immutable for the duration of the second phase, the computation of their result need not be synchronized. The dependency graph is the only shared mutable state.

Since an optimizer for a given method m only removes and adds edges (m, q), i.e., for its own method, different optimizers cannot act on the same edges. Moreover, during this phase, the graph is never read, only written to. It is therefore sufficient to implement the graph with a concurrent data structure that supports atomic addition and removal of edges, while keeping the order of operations coming from a single thread. This can easily be done with the non-blocking concurrent hash tries developed by Prokopec et al. [14].

[5] Recall that, during step 3, no class is added or removed, hence the list of child classes is not modified.

6. Results

We evaluated the implementation of the parallel incremental algorithm in Scala.js along the following two axes:

- How the incremental version improves over the batch version of the same program.
- How the parallel implementation scales with the number of threads.

When benchmarking, we measure the running times of the invalidation phase and the reoptimization phase separately. Since the invalidation phase would not be needed at all if the algorithm did not have to support incremental compilation, we compare the time of the reoptimization phase in batch mode to the cumulative time in incremental mode. This is actually slightly biased in favor of the batch mode, since part of the job of the invalidation phase is also to construct the class hierarchy data structure, used to resolve calls, and in general answer knowledge queries.

6.1 Batch Mode vs Incremental Mode

Comparing the running time of the batch and incremental modes is the most important aspect of our contribution. We have measured this with two very different approaches: on the one hand, in a controlled environment with synthesized changes, and on the other hand, in real life by actual users of the optimizer during their day-to-day development.

6.1.1 Measures in a Controlled Environment

To evaluate the speedup obtained from running the incremental optimizer rather than a batch optimizer, we took the codebase of Papa Carlo [8]. This codebase contains 4974 reachable methods (the number of methods that the batch mode has to optimize), which is a relatively small though non-trivial Scala.js codebase. In batch mode, we link the full program with a clean optimizer. In incremental mode, we first optimize a slightly changed version of the program (with a fresh optimizer) and then optimize the original version of the program. This ensures that the resulting program is the same for batch and incremental mode.

We performed the measurements on an Intel Core i7-3770K clocked in at 3.5GHz, allocating 4GB of RAM to the JVM running the experiments. This CPU has 4 cores able to run 8 threads.

We compare the running time of the batch mode against the following synthesized incremental changes to the program:

1. Modify the body of an inlineable method
2. Modify the body of a non-inlineable method

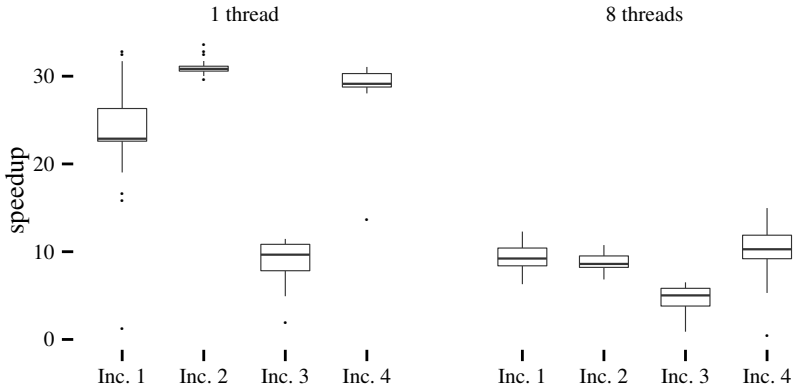

Figure 3. Speedups in incremental mode with respect to batch mode

	# meth.	time [ms]		speedup		MAD [ms]	
Batch	4974	246	(97)	1	(1)	3.2	(27.3)
Inc. 1	2	11	(11)	23	(9)	1.7	(1.6)
Inc. 2	1	8	(11)	31	(9)	0.1	(1.2)
Inc. 3	8	25	(19)	10	(5)	5.1	(4.9)
Inc. 4	17	8	(9)	29	(10)	0.2	(1.9)

Table 1. Batch mode vs incremental mode for various changes. Running with 1 thread (8 threads).

3. Add a method used in a polymorphic dispatch
4. Mark an existing class as eligible for scalar replacement

These changes may seem arbitrary, and it might appear that using the history of a version control system (VCS) would give more realistic results. However, this is a fallacy, because commits in a VCS are much too coarse, with respect to what our incremental optimizer is trying to achieve. In a typical development workflow, we run a compile-optimize-run/test cycle many times to eventually create a single commit. It is common practice to perform continuous testing (running unit tests on every file save) of the optimized build with Scala.js, since it requires no setup.[6] The time between two cycles might go from a few seconds to a couple of minutes, while the time between two commits can cover much longer development times. VCS commits are therefore not good representatives at all. These very small synthesized changes are in fact much closer to the reality.

Figure 3 shows the speedups of incremental mode with respect to batch mode for the sequential version of the algorithm and the parallel version with 8 threads (absolute running times of batch mode are in Figure 5).

Table 1 shows the number of methods that were invalidated, median running time, speedup factor and median absolute deviation (MAD) for the running time.

[6] The build command `sbt ~test` performs optimized continuous testing by default.

For the single-thread case, we observe speedups up to a factor of 30 for Inc. 2 and Inc. 4. The lowest gain can be seen on Inc. 3, which only exhibits a speedup of 10x. Multi-threaded setups offer smaller speedups, because there is more contention on the parallelization of the first phase, which causes the batch optimizer to benefit more from multiple threads, relatively to the incremental optimizer. Improving the parallelism of the first phase will be future work, this mostly involves improving current work stealing techniques. It is remarkable that despite this, we can still observe speedups up to a factor 10.

6.1.2 Real Life Benchmarks

The previous section analyzed reproducible benchmarks in a very controlled environment. However, the real purpose of our incremental optimizer is to reduce the time actual developers have to spend waiting during their development cycle. We have therefore benchmarked the optimizer in real life situations: the developers of 6 Scala.js codebases have used an instrumented version of the optimizer for several days during their day-to-day programming tasks. The instrumented optimizer measured the running times of both the incremental and batch optimizer on the codebase every time it was invoked (up to several times per minute on some codebases).

These benchmarks were run in unknown and uncontrolled environments, during normal development tasks. We therefore expect that other processes were running at the same time. Moreover, every measure is unique and non reproducible.

Although non reproducible, we consider these measurements much more important in practice, since they are real life usages of the optimizer.

Table 2 lists the 6 codebases that have been benchmarked. The number of methods is a rough approximation, since it changes from one measurement to the next, and serves only as an estimate of the size of the codebase. Figure 4 shows, for each codebase, the speedup of the incremental optimizer wrt. the batch optimizer. Table 3 shows the associated numeric data.

	name	# methods	# data points
A	scalajs-games demo	5,000	414
B	react-ext test suite	9,500	31
C	ScalaCSS playground	9,000	54
D	Scala.js land	5,500	42
E	Scala.js land admin	9,500	32
F	Anon. web app client	20,000	41

Table 2. Codebases benchmarked in real life.

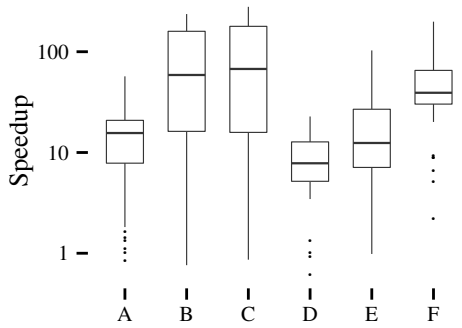

Figure 4. Speedup in incremental mode wrt. batch mode (log scale, higher is better)

	Batch [s]	Inc [ms]	speedup
A	0.17 ± 0.0	10.5 ± 4	16 ± 9
B	3.08 ± 0.3	24.5 ± 27	59 ± 68
C	4.43 ± 0.3	65.9 ± 63	67 ± 83
D	0.28 ± 0.2	35.0 ± 19	8 ± 5
E	0.44 ± 0.4	42.9 ± 44	12 ± 10
F	3.74 ± 0.4	92.9 ± 47	39 ± 30

Table 3. Measurements on real life codebases (median $\pm$ MAD)

# threads	time [ms]	speedup	MAD [ms]
1	4755.9	1.0	127.0
2	2689.1	1.8	194.8
4	2467.3	1.9	228.8
8	2465.7	1.9	169.8

Table 4. Scaling with the number of threads – batch mode

On these benchmarks, we can observe speedups from 10x to 100x, which is a huge improvement given that these are real life measurements.

6.2 Scaling with the Number of Threads

We also compare the batch mode running times with different numbers of threads. These measurements were performed on the same Intel Core i7-3770K clocked in at 3.5GHz, on the Scala.js test suite. This codebase contains 12,311 reachable methods.

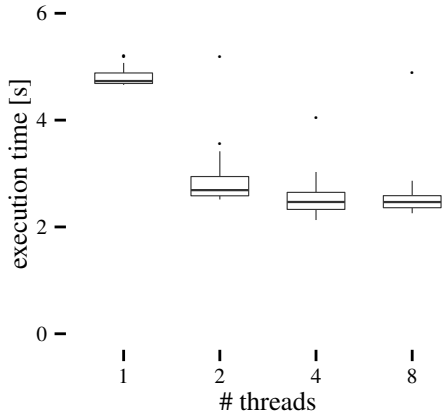

Figure 5. Batch mode running times

Benchmark	without opt.	with opt.	speedup
Richards	3.47	1.04	3.34
Tracer	4.63	1.11	4.17
DeltaBlue	5.49	1.24	4.42

Table 5. Run-time performance of the emitted programs. The reported figures are the running times normalized against the hand-written JavaScript version (lower is better). The last column shows the speedup brought by the optimizer (higher is better).

The results are shown in Figure 5 and Table 4. We report the median running times, the speedup factor compared to sequential execution and the median absolute deviation for the running time. A decent amount of scaling can be observed, although it seems there is little to be gained beyond two threads. We believe bigger gains can be obtained with further work, replacing the simple usage of the Scala parallel collections by more targeted work stealing.

6.3 Performance of the Resulting Programs

It might be argued that the set of optimizations we have discussed and implemented are not representative of real-world optimizers. We have analyzed the run-time performance of the programs produced by our optimizer on three of the Octane benchmarks [1], a suite of benchmarks for the V8 JavaScript virtual machine: Richards, Tracer and DeltaBlue. The benchmarks were run in 3 different configurations, using the same setup as above:

- The original benchmark, hand-written in JavaScript,
- The benchmark ported to idiomatic Scala.js code, without using the optimizer, and
- The same code optimized by our optimizer.

The results are shown in Table 5. Running times with and without optimizer are normalized against the hand-written JavaScript version. We can see that the performed optimizations bring significant speedups, from 3x to 4x.

Additionally, the resulting programs are competitive with hand-written JavaScript code.

7. Limitations

The approach presented in this paper focuses on speed for the incremental optimizer, and has some limitations.

7.1 Dynamic Language Features

Dynamic language features such as reflection and dynamic loading can invalidate the assumptions made by our optimizer pretty quickly. However, this is not so much a limitation of the *incremental* nature of the optimizer as of the whole-program assumption. In fact, having an incremental optimizer can be beneficial if such language features are used, provided a Just-In-Time compiler can incrementally reoptimize parts of the program that are invalidated.

In the context of Scala.js, however, this is a non-issue. Scala.js does not provide any reflection nor dynamic loading by design. Dynamic loading is typically avoided in JavaScript environments, to enable bundling the entire application as a single JavaScript file. In larger applications, dynamic loading is used to improve startup times, but the whole program is still known before distribution; it is only fragmented after the fact (after whole-program optimizations) to enable lazy loading. Run-time reflection, on the other hand, is heavily used by dynamic languages, but is virtually never necessary in a language like Scala, which supports advanced compile-time metaprogramming features [3]. In practice, Scala.js developers replace run-time reflection by compile-time reflection for features such as automatic JSON serialization [11], statically typed RPC calls with the server [9], and so on.

7.2 Other Kinds of Whole-Program Optimizations

The optimizations we cover are essentially type-based. We have not yet considered other kinds of whole-program optimizations such as those requiring flow-based analyses. In theory, this is a severe limitation, but Section 6.3 has shown that the optimizations we support are enough to bring idiomatic Scala.js code to a competitive level with respect to hand-written JavaScript code. The particulars are out of the scope of this paper, but the general insight is that we focus on generating code that is as friendly as possible to the next compiler in line, i.e., the JIT compiler. The optimizations we implement allow to remove the overhead of typical Scala code, down to imperative, first-order monomorphic JavaScript code. JavaScript virtual machines are good at optimizing such code further using run-time profiling information.

In other contexts, the limitations currently imposed by our approach can be lifted with a two-staged process. During iterative development, only optimizations that lend themselves to being incrementalized are used. This already provides a major improvement over the non-optimized code, therefore reducing the time spent on compile/test cycles. We then add the other optimizations when emitting the final, production executable, to compensate for the limitations of the approach.

8. Related Work

Incremental reoptimization of generated programs was pioneered by Pollock and Soffa as early as 1985 [12]. This first work was concerned with optimizations strictly local to basic blocks. They later extended their algorithm [13] to accommodate procedure-level optimizations.

At about the same period, Cooper et al. [5] introduced a first framework for incremental recompilation with interprocedural knowledge, which was extended later by Burke and Torczon [2]. Their framework tracks the assumptions made by the optimizer in the form of sets describing what callees *may* do without necessitating reoptimization. If any of the assumptions are broken, the procedure is marked for recompilation, ensuring the validity of the resulting program. However, their framework does not handle the reverse operation: should any new opportunity for optimizations arise, it is not detected. They also acknowledge the difficulty of recording all necessary assumptions. Knowledge queries can be viewed as a generalization of this idea with several improvements:

- They support non-boolean queries, for example, the set of possible targets of a dynamic call. This is critical to support object-oriented patterns, or simply inlining.
- They automatically record all the relevant assumptions, since the queries are the only interface between the optimizer and the program.
- They detect new opportunities for optimizations.

Chambers et al. [4] proposed an impressive framework that can be viewed as more general than our approach, since it applies to the entire compilation pipeline. Filtering nodes in their framework are basically equivalent to knowledge queries in terms of dependency tracking. However, as we explained in the Section 1, the measures and evaluations of this framework were focused on *accuracy*, omitting any discussion of the run-time overhead of their detection algorithm. Our approach has significant differences geared towards good run-time performance of the detection algorithm itself. In particular, it does not require factoring nodes, computed at run-time, to avoid memory blow-up, because we automatically compute the set of queries impacted by program changes instead of storing them in the dependency graph. Our algorithm also easily parallelizes, as we have shown in Section 5, which is an important property nowadays. To achieve this, we sacrificed the fully automatic nature of the change detection algorithm. Since the evaluation of [4] only shows the number of methods that need to be reoptimized, rather than the actual run-time performance of their framework (e.g., how the incremental algorithm performs with respect to the batch algorithm), we cannot draw any measurable comparison with that work.

A related although somewhat different area is program analysis. There have been numerous works in that area in the past decades, including incremental whole-program analyses, e.g., [10, 15, 17, 19, 20]. The results of such advanced analyses can be used by optimizers, among other tools, to

produce more efficent code. Incremental analyses can form a very powerful combination with incremental optimizers such as ours, since the whole pipeline can be incrementalized. Knowledge queries can be used to ask facts about the result of the static analysis. The changes to the static analysis results can be incrementally used to compute the set of knowledge queries that are impacted, which in turn will reoptimize only the appropriate set of methods. Actually, our program diffing algorithm can be viewed as a very weak form of incremental class hierarchy analysis. Combining state-of-the-art incremental program analyses with our optimizer could provide better results in the future.

9. Conclusion

We have presented a new approach to designing incremental whole-program optimizers using knowledge queries, and showed how to apply this approach to common optimizations in object-oriented and functional languages. Knowledge queries abstract away the details of the optimizer when analyzing changes to the program, and therefore create a modular interface between the optimization logic and the incremental logic.

The implementation of an incremental whole-program optimizer for Scala.js with this approach shows speedups from 10x to 100x with respect to batch processing. This means that, in the broader context of the compilation pipeline, whole-program optimizations take negligible time.

There are, however, limitations to the technique, essentially because methods must be optimized independently. In other words, the optimization of a method may not depend on the result of optimizing other methods of the program. This prevents some advanced optimization decisions, e.g., inlining a target based on its optimized size, or optimizations that must be applied consistently in several methods or not at all. A work-around for this limitation is to apply additional, non-incremental optimizations only when producing the final executable, but not on every compile cycle.

Further work includes partially removing the above limitation, notably to enable scalar replacement of class fields, and integration with incremental program analysis techniques.

Acknowledgements

This work has been supported by the Swiss Commission for Technology and Innovation (CTI) grant 16613.1.

References

[1] Octane, the JavaScript benchmark suite for the modern web, 2015. [Online; accessed 30-November-2015].

[2] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Trans. Program. Lang. Syst.*, 15(3):367–399, July 1993.

[3] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013.

[4] C. Chambers, J. Dean, and D. Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 221–230, New York, NY, USA, 1995.

[5] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 58–67, New York, NY, USA, 1986.

[6] J. A. Dean. *Whole-program Optimization of Object-oriented Languages*. PhD thesis, 1996. AAI9716832.

[7] S. Doeraene, T. Schlatter, and N. Stucki. Scala.js, 2015. URL `http://www.scala-js.org/`. [Online; accessed 30-November-2015].

[8] I. Lakhin. Papa carlo, 2015. URL `http://lakhin.com/projects/papa-carlo/`. [Online; accessed 30-November-2015].

[9] H. Li. Autowire, 2015. URL `https://github.com/lihaoyi/autowire`. [Online; accessed 30-November-2015].

[10] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[11] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 183–202, New York, NY, USA, 2013.

[12] L. L. Pollock and M. L. Soffa. Incremental compilation of optimized code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 152–164, New York, NY, USA, 1985.

[13] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Trans. Program. Lang. Syst.*, 14(2): 173–200, Apr. 1992.

[14] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 151–160, New York, NY, USA, 2012.

[15] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005.

[16] A. Sewe, J. Jochem, and M. Mezini. Next in line, please!: Exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 317–328, New York, NY, USA, 2011.

[17] A. L. Souter and L. L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 682–, Washington, DC, USA, 2001.

[18] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014.

[19] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 35–46, New York, NY, USA, 2001.

[20] J.-S. Yur, B. G. Ryder, W. A. Landi, and P. Stocks. Incremental analysis of side effects for C software system. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 422–432, New York, NY, USA, 1997.