Institut für Programmstrukturen
und Datenorganisation (IPD)

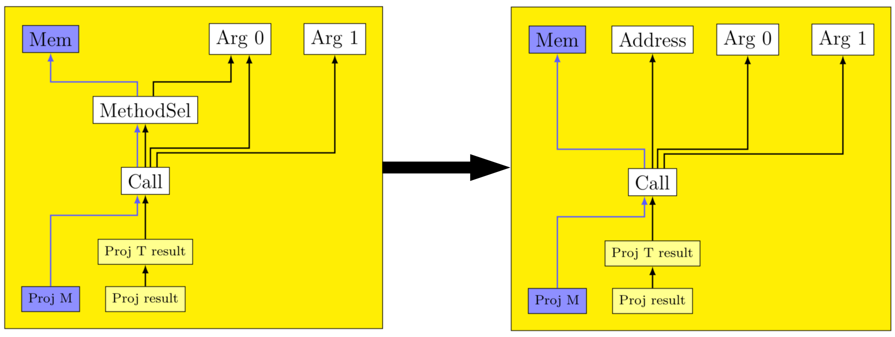
Lehrstuhl Prof. Dr.-Ing. Snelting

Improved Devirtualization in libFirm

Bachelorarbeit von

Daniel Biester

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M.Sc. Andreas Fried
Abgabedatum:	19. Oktober 2018

Zusammenfassung

Devirtualisierung beschreibt das Ersetzen dynamischer Methodenaufrufe durch statische Methodenaufrufe, wenn zur Kompilierzeit ein einziges Ziel des dynamischen Aufrufes ermittelt werden kann. In libFirm wird dies aktuell mittels Rapid-Type-Analysis (RTA) erreicht. In dieser Arbeit geht es darum, den Devirtualisierungsprozess mittels XTA zu verbessern, damit mehr dynamische Aufrufe zu statischen umgewandelt werden können. Das Ergebnis ist eine präzisere Analyse, die weniger Methoden als erreichbar markiert.

Devirtualization describes the replacement of dynamic method calls with static method calls when, during compilation, it is possible to determine that a dynamic method call has only one target method. Currently, in libFirm this is achieved via Rapid-Type-Analysis (RTA). This thesis is about improving the process of devirtualization via the introduction of XTA, so more dynamic method calls can be replaced with static ones. The result is a more precise analysis which marks less methods as reachable.

Contents

1 Introduction

Many programs nowadays are written in object-oriented programming (OOP) languages. One common feature of these languages is that methods can be called based on the dynamic type of the object they are called on. Thus, the target of a dynamic function call is, without further analysis, not known at compile time. But there are two advantages of knowing the target at compile time:

First there is the immediate time advantage gained by replacing a dynamic call with a static one. This is because the dynamic call works via a v-table, where all methods' addresses for one object type are stored. So calling a method via dynamic dispatch involves loading the v-pointer to a v-table, loading the method's address using the v-pointer and then calling the function at the specified address. In contrast, a static call involves only the last step: calling the function at the specified address.

The other advantage of replacing a virtual call with a static one is that further optimizations, e.g. inlining of a devirtualized call, can be done.

Especially the latter improves the performance of a program significantly.

This bachelor thesis is about improving the devirtualization process in the compiler framework libFIRM. libFIRM features a graph-based intermediate representation in SSA form and several front- and backends as well as many optimization options. One of these is the Rapid-Type-Analysis to devirtualize method calls. The idea behind RTA is to search for all object instantiations and to then collect the type of the created object. By reading the whole program once, all used object types are known as well as the class hierachy. Thus, one can use this information to infer for each dynamic method call, whether there is only one target or several. If there is only one, the call's devirtualization is possible.

The problem is that RTA is imprecise as it is only recorded which object types are in use and not where they are actually used. Thus, the following call to `foo()` cannot be devirtualized using information obtained via RTA:

```
class A { void foo() {} }
class B extends A {
  void foo {}

  public static void main(String[] args) {
    A a = new A();
    bar();
  }

  static void bar() {
        A b = new B();
        b.foo(); //Call to foo()
    }
}
```

Listing 1.1: Example code where RTA cannot devirtualize call to `foo()` but XTA can.

RTA will collect the information that an object of type `A` as well as type `B` was created. When analyzing the call site to `foo()`, it walks the class hierarchy from the static type of variable `b`, which is `A`. Two methods can be called: `A::foo()` and `B::foo()`.

A more precise analysis is XTA. Instead of one set of created object types, a set is created for each called method and referenced field. These sets store all object types which are accessible in a method or through a field. The idea is to model the possible flow of object types between methods and fields as objects are passed on as parameters or returned to another method. Thus, the immediate effect of an object creation is local to one method.

Analysing the example using XTA, the following result is achieved: The call to `foo()` gets devirtualized, as only objects of type `B` are accessible inside `bar()`. The object creation of type `A` in the main method does not alter the results, as the object is not passed on to `bar()`.

The goal of this thesis is to improve the devirtualization process by implementing XTA in libFirm.

2 Basics and Related Work

2.1 libFirm

libFirm is a library providing a graph-based intermediate representation called FIRM, plus optimizations and assembly code generation to compilers [1]. As compilers are split into frontend and backend, libFirm can be used as a backend for different compilers. Generally, the frontend reads program code and creates an intermediate representation (IR), whereas the backend uses the IR to create executable assembly code. The advantage of using an IR is that multiple frontends and backends are independent from each other, and one backend can be used by different frontends and vice versa. Additionally, optimizations to the IR become source and target code independent.

libFirm [1, 2] uses FIRM (Fiasco's Intermediate Representation Mesh) as an IR, originally developed for the Sather-K compiler Fiasco, hence its name. It features a SSA-graph based representation of code. Single Static Assignment (SSA) [3] requires that every (local) variable is only defined once, eliminating the concept of variables altogether, replacing it with SSA values. This simplifies many possible optimizations. Converting a program into SSA form is done by renaming the variable each time a new assignment occurs. Additionally a special ϕ-function is needed, whenever two control flows merge and a value needs to be chosen from depending on the taken path.

SSA form allows for building a graph without variables: The graph links together each operation and the associated operands, the results of previous operations. Thus this graph shows the dependency between operations. Whereas IRs featuring an instruction list representation imply a total order of operations, a dependency graph representation only leads to a partial order, making the backend responsible for calculating a total order, by scheduling instructions.

A function in FIRM is represented by a program graph, starting with a start node and ending with an end node. All nodes represent an operation and the edges represent the dependencies between them. Edges are directed from a node to its dependencies. Thus the graph is reversed compared to a control flow graph where edges link a node to the subsequently executed node. A separate model is included in the graph to keep track of memory changes. The specific state of the memory is represented by a node, each change via load/store operations results in a new memory state, thus a new node. This is necessary, as memory-dependent operations have to be executed in the correct order.

[1] `https://pp.ipd.kit.edu/firm/`

Many different node types exists. Besides basic arithmetic operations, e.g. `Add`, `Sub`, `Mul`, `Div`, `Shift`, there are nodes for function calling, `Call`, as well as a `Return` node and the previously discussed `Load` and `Store` nodes.

Additionally, the nodes `Address`, `Offset`, `Member` and `Const` exist, representing addresses, constants or members of an entity.

There is also a projection node, `Proj`, for selecting a specific value out of a tuple of values. This is necessary, so edges do not need to store additional information, besides where they are pointing to.

To model the control flow, operations are grouped into blocks. `Jump` nodes, conditional and unconditional, exist to jump to a different block.

The `Call` node resembles a function call and depends on a memory state, the call target's address and a number of possible arguments. As functions in FIRM are entities, the `Address` node contains the call target's function entity. The output of a `Call` node is a new memory state and a result `Proj` node.

Thus the FIRM graph is the combination of a dependency graph and a control flow graph.

Figure 2.1 is an exemplary graph of a simple function. The blue edges are the memory dependency edges. As there is no operation involved changing the memory state, the original state is passed through. In the first block, the arguments are separated to different nodes to pass them to a compare operator. Based on the comparison result, either the left add or the right subtract node is chosen. These blocks consist of a simple arithmetic operation, dependent on both arguments, and a jump statement for getting into the last block. To merge the data dependency flow, a ϕ-node is placed in the last block. The last statement is the return, before the end node signals the end of the graph.

The FIRM graph is created iteratively using the algorithm described by Braun et al. [4]. During the process several optimizations are done on-the-fly, including Arithmetic Simplification, Common Subexpression Elimination and Constant Folding. Inherent optimizations, as of the SSA-graph's creation, include Copy Propagation and Dead Code Elimination. Copy Propagation removes unnecessary assignments. But as there are no local variables in a FIRM graph, there are no assignments either. As the graph is a dependency graph, dead code gets eliminated automatically.

Type information is stored in a separate graph. This graph represents the dependencies between different types. Several types of nodes exists: Primitive types store primitives, pointer types specify the type of the instance they are pointing to and method types list argument and return types. Additionally, Array, Union, Struct and Class types exist. Class types have a collection of fields and methods and a list of supertypes. Information about overwritten fields and methods is also stored. A special class GlobalType exists, to list global variables as well as static methods.

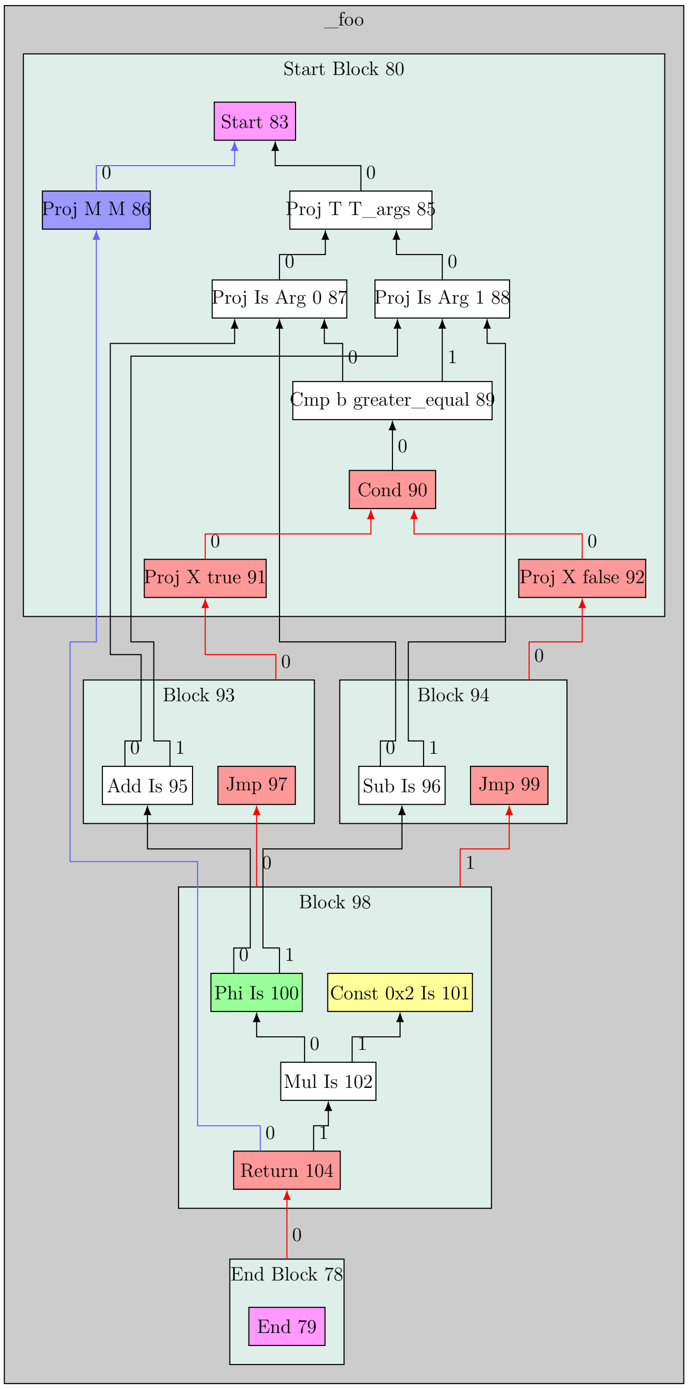


Figure 2.1: This FIRM graph shows the function foo which takes two arguments and returns the result of a multiplication. The two arguments `Arg 0` and `Arg 1` are either added up or one is substracted from the other one, depending which one is greater. Afterwards, the result of the addition or subtraction is doubled and returned.

2.2 Liboo

Liboo is a library, depending on libFIRM, which adds support for object-oriented programming constructs. Especially dynamic dispatch via v-tables and the creation of these are handled. During a lowering phase all added features are replaced by libFIRM instructions, making it compatible to all optimizations include in libFIRM.

Liboo introduces a new node named `MethodSel` to handle dynamic dispatch. This node hides the complexity of selecting a call target by reading the v-table. The `MethodSel` node takes the place of an address node, as its output is the address of the actual call target. A method in an object-oriented language resembles a function where the *this*-pointer is the first parameter. Thus, the `MethodSel` node depends on the call's first argument (index zero), as the method selection needs the v-table. Additionally, the node stores the *call entity*, a FIRM entity representing the static type's method, needed to search for the right method to be called, based on the *this*-pointer's dynamic type.

2.3 Bytecode2Firm

Bytecode2Firm is a frontend which translates java bytecode to the FIRM IR. It's class-file compatibility is tested for the versions 49, 50, and 51, which corresponds to the Java SE versions 5, 6 and 7. Therefore it lacks support for newer language features such as anonymous functions. Also exception handling is not yet included.

Bytecode2Firm has a runtime library included called SimpleRT which supports basic Java features. In order to access the full standard library, one has to depend on libgcj, the runtime-libraries of GCJ (GNU Compiler for Java).

2.4 Static Analyses used for Devirtualization

One way to determine whether dynamic method calls can be devirtualized is the use of static program analyses. Static program analyses examine the program code itself without executing it, while dynamic analyses draw conclusions by running the program. In order to accomplish the devirtualization of virtual call sites, static analyses often build a call graph. A call graph represents the calling relationships between methods. For every method a node is created in the graph, and for each static call site in a method an edge is created from the method to the call's target method. As the target of a dynamic call is only known at run time, the analysis has to approximate the list of possible target methods. For each possible target method an edge has to be generated from the caller to the callee. If the analysis is able to reduce the number of call targets at a virtual call site to only one target method, the virtual call can be converted to a static one. Thus, the call is devirtualized.

A method to which no edge is leading to can be removed as the method is not called. This optimization step of removing unreachable methods is called Dead Code Elimination.

The goal is to build a conservative approximation: A graph that may not be as precise as the optimal graph, but still includes the optimal graph, also called over-approximation. As building the exact call graph statically is an undecidable problem [5], the goal is to approximate the call graph conservatively. The result is a graph that may not be as precise as the optimal graph, but still includes the optimal graph.

A requirement for building a complete call-graph is to consider the whole program, not just one translation unit. In order to build a sound and precise call graph, and to be able to devirtualize as many calls as possible, the following assumption has to be made: The known code forms a closed world. Thus, nothing outside the known code base is used by the known code and vice versa.

Some of the static analyses that can be used for devirtualization are closely related to each other, with each analysis improving on another one. By using set-constraints and sets of live classes, i.e. classes in use, to formulate the analyses, the similarities and improvements are displayed very clearly. The goal is to find the least sets, that fulfil all constraints.

A basic approach to devirtualize dynamic calls is to replace every dynamic call with a static one where there is only one target method with the given signature. The set-constraints define the set of reachable methods R as follows:

1. $main \in R$
2. For each method M, each virtual call site $e.m()$ in M, and each method M' with the same signature as m:

$$M \in R \implies M' \in R$$

Thus, reachable methods are methods that can be reached from the main method and additionally all methods with the same signature as a method from a call site. So devirtualizable calls are virtual calls, where there is only one method with that signature.

This approach is called Reachability Analysis [6] and closely resembles the Unique Name Analysis [7], which compares mangled names of methods at link time to devirtualize calls. The prerequisite for correctly devirtualizing all calls is the knowledge of all method signatures, otherwise calls may get devirtualized that have target methods which are unknown to the analysis. This results in a corrupt program.

A slightly advanced version considers the class hierarchy, thus it is called Class Hierarchy Analysis [8]. Whereas rule Nr. 1 stays the same, rule Nr.2 changes to following:

2. For each method M, each virtual call site $e.m()$ in M, and each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ where $\mathit{StaticLookup}(C, m) = M'$:

$$M \in R \implies M' \in R$$

where $\mathit{StaticType}(e)$ is the static type of the expression e, $\mathit{SubTypes}(t)$ is the set of all declared sub types of type t. The notation $\mathit{StaticLookup}(C, m)$ is the static lookup, searching for a method named m starting from class C, then going upwards in the class hierarchy.

The difference to the Reachability Analysis is that CHA uses the class hierarchy to narrow down the search for matching methods: for a call $e.m()$ only methods with the name m which are inherited by a subtype of the static type of e will be regarded as reachable. Thus, the requirements for a correct analysis get extended to not only requiring all method names, but also the complete class hierarchy. While CHA does work only taking into account method names, using signatures instead does also work and enhances the outcome.

2.4.1 RTA

The Rapid Type Analysis [7] extends the CHA by using information about object-instantiations to further reduce the possible number of targets for a dynamic call. The idea is to track each object instantiation and save the type of the object. Therefore a new set S is introduced to keep track of classes which are considered in use. Now only methods of classes which are in S can get called, changing constraint Nr. 2 to the following:

2. For each method M, each virtual call site $e.m()$ in M, and each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ where $\mathit{StaticLookup}(C, m) = M'$:

$$M \in R \wedge C \in S \implies M' \in R$$

Additionally, a new constraint is needed to define which classes are in the set S for used classes:

3. For each method M and each object creation in M, e. g. "new C()":

$$M \in R \implies C \in S$$

This states that only object instantiations in reachable methods are taken into account.

In contrast to CHA and RA, RTA does need full knowledge of the code to track object instantiations. Thus the closed-world assumption has to be valid. This can be a major drawback, as compilers often link pre-compiled libraries to a program. These libraries cannot be analyzed, thus objects created in a library cannot be tracked. If these objects are used in the analyzed and optimized code, the program may malfunction. This does occur when a call gets devirtualized, but an object is created in non-analyzed code and changes the possible call targets for that call.

RTA is already implemented in liboo [9].

2.4.2 XTA

While RTA is fast and simple, the usage of only one set of live classes can be too imprecise. It gives only a global view of classes in use, but a more local view may help devirtualizing additional call sites. XTA [6] is a hybrid approach and the combination of FTA and MTA, which are seperately explained in section 2.4.3. To give a more local view, a set S_M of live classes is created for each method M and a set S_x for each field x. The set of a field contains the object types which are at some point stored in it. The set of a method has following classes in it:

- All types of objects which are initialized in the method.
- All possible types of objects which are returned from a call site inside the method.
- All possible types of objects passed to the method as an argument.

To fill the sets, the following constraints are used:

2. For each method M, each virtual call site $e.m()$ in M, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) = M'$:

$$M \in R \land C \in S_M \implies \begin{cases} M' \in R \land \\ SubTypes(ParamTypes(M')) \cap S_M \subseteq S_{M'} \land \\ SubTypes(ReturnType(M')) \cap S_{M'} \subseteq S_M \land \\ C \in S_{M'} \end{cases}$$

$ParamTypes(M)$ are the classes of the parameters, excluding the *this*-Pointer parameter, and $ReturnTypes(M)$ are the return types of the method M. $SubTypes(\cdot)$ is the combined set of subtypes of the given set of types.

This is a refinement to RTA's similar constraint Nr. 2 as it only considers target methods of classes which may be in use at the call site. Additionally, the flow of classes is captured by taking into account the parameters and the return type of a method. This is done by using the information gained about which classes are live at a call site and by only passing on live classes that are subclasses of the defined parameters. Handling the return type of a method is done the same way. As the *this*-Pointer is not seen as an argument, the last part of the constraint explicitly states that the type of the object the method belongs to is live.

The third constraint handles the construction of new objects:

3. For each method M and each object creation in M, e. g. "new C()":

$$M \in R \implies C \in S_M$$

As XTA not only has sets for each method, but also for each field, two new constraints are needed to model a field's reads and writes:

4. For each method M in which a read of field x occurs:

$$M \in R \implies S_x \subseteq S_M$$

5. For each method M in which a write to field x occurs:

$$M \in R \implies S_M \cap SubTypes(StaticType(x)) \subseteq S_x$$

A field read sets all the field's live classes live in the method the read is in. A field write adds all subtype's of the field's static type to the field's set of live classes, if they are considered live in the method the write is located in.

In order to deal with unavailable code, Tip and Palsberg propose the use of a set S_E of classes associated with library code. The following rules are introduced:

- Calling a library method M' in an application method M results in the propagation of the all possible parameters to S_E. Also all possible types of the *this*-pointer from virtual method calls are propagated. This results in the following set relation:

  $$M \in R \wedge C \in S_M \implies SubTypes(ParamTypes(M') \cup this) \cap S_M \subseteq S_E$$

  for each virtual call site $e.m()$, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) = M'$ and C is a library class.

- A write in method M to a field f defined outside the application leads to the propagation of $S_M \cap Type(f)$. A read propagates $S_E \cap Type(f)$ to S_M. Read:

  $$M \in R \implies S_E \cap SubTypes(StaticType(f)) \subseteq S_M$$

  Write:

  $$M \in R \implies S_M \cap (SubTypes(StaticType(f)) \subseteq S_E$$

- For each method M overwriting a library method the assumption is made, that a library method will call it. The set S_E is used to determine which method M can be called from the library. For such methods M' the flow of parameter and return types is modeled using $S_{M'}$ and S_E in accordance with rule 2.

2.4.3 CTA,FTA and MTA

There are several analyses using more than one set for approximating live classes, but which are less complex than XTA, including the aforementioned FTA and MTA.

1. CTA is an analysis using only a set for each class, unifying all sets S_M and S_x of a class C to S_C.

2. MTA also defines a set S_C for each class, but also a set S_x for each field x.

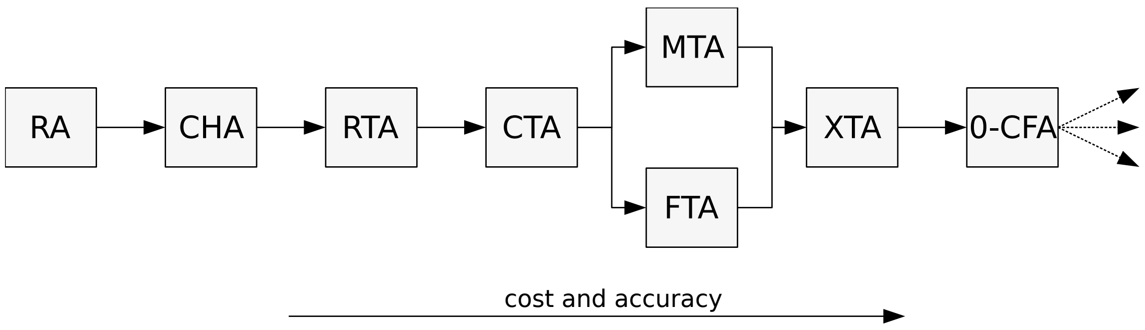


Figure 2.2: The relationship of the analyses in regard to cost and accuracy (adapted from [6])

3. FTA also defines a set S_C for each class, but also a set S_M for each method M.

Thus XTA combines MTA and FTA to grasp the flow of live classes between methods and fields.

Tip and Palsberg [6] classify the algorithms by how many sets are used for approximating run-time values of expressions. While the Reachability and the Class Hierarchy Analysis do not use such a set, the Rapid Type Analysis uses one to keep track of the global state. More precise analyses such as XTA, FTA, MTA and CTA use more than one set for the whole program.

CTA, FTA, MTA, and XTA can be executed in $O(n^2 \times C)$, where n is the number of used sets and C is the number of classes [10]. Thus, XTA can be run in $O((M + F)^2 \times C)$, where M is the number of methods, F is the number of fields.

The relationship between all analyses discussed is shown in figure 2.2.

2.5 Related Work

While this thesis is about devirtualization via XTA, there are several different approaches discussed in other papers. The following section is a brief overview about these.

2.5.1 Points-To Analysis

The Points-To Analysis [11] is classified as 0-CFA by Tip and Palsberg [6]. 0-CFA is the class of control-flow analyses using one set per expression to model the flow of objects. The Points-To Analysis' goal is to find the set of possible objects a pointer can point to, thus making it more precise than XTA. Still the procedure closely resembles the one of XTA, but now using one set for each pointer: At every site an object is created, it is inserted in the points-to set of the specified pointer. Each variable assignment results in a set-relation between the two involved points-to sets. Thus many set relations are gathered and resolving these can be done in $O(n^3)$, where n is the number of sets.

Several approaches exists to speed-up the Points-to Analysis. Steensgaard [12] proposed to use set equations instead of inequalities. On the one hand, it makes the analysis less precise, on the other hand, this version runs in almost linear time. Berndl et al. [13] propose the use of binary decision trees to improve speed and memory consumption.

To make the points-to analysis more precise it is refined to an object- and heap-sensitive one. To speed up these object- and heap-sensitive points-to analyses, one can only selectively apply a precise points-to analysis to program parts where the expected gain is high [14].

Sundaresan et al. [15] alter the points-to analysis by recording only object types instead of object representatives, thus reducing the number of elements in the points-to sets.

2.5.2 Devirtualization techniques in modern compilers

Much research went into devirtualization techniques suitable for Just-in-Time compilers. Regarding devirtualization, their big advantage is that they don't need to make the closed-world assumption, as they can cope with class hierarchy changes during runtime. Dynamic recompilation using on-stack replacement [16] can be used to handle dynamic class loading.

Another technique is the direct devirtualization with the code patching mechanism [17] where inlined code of a devirtualized call is executed as long as the current class hierarchy supports the devirtualization. When the class hierarchy changes and does not support the call's devirtualization anymore, the code is patched and the dynamic call site is reintroduced.

LLVM

LLVM is a compiler library featuring an SSA-based IR. Devirtualization is realized using load/store optimization. Using instruction metadata the frontend has to mark every load and store operation with a so-called *invariant group* metadata. This tells optimizations that they can assume that a load from a pointer annotated with that metadata will always return the same value.

Therefore pointers to the same memory location, but with different dynamic type must have different SSA values. This occurs in C++ for example, if placement new is used to create a new object.

If a constructor is not inlined the optimizer cannot know the value of the v-pointer. Thus, via an `@llvm.assume` intrinsic it is assumed that after a call to a constructor the v-pointer points to the v-table of the constructed object's type.

3 Design and Implementation

This chapter is about designing and implementing XTA in liboo. Section 3.1 is specifically about the design decisions made, section 3.3 is about how calls are devirtualized section 3.2 explains the propagation algorithm, section 3.4 is about finding all target methods to a virtual call, and section 3.5 explains how object creations and field reads and writes are recognized in a FIRM graph.

3.1 Design Decisions

Just like the already existing RTA [9], XTA is part of the liboo library. This is the lowest layer where object-oriented constructs, such as object creations, can be recognized in a FIRM graph. Placing XTA in liboo has the advantage that all frontends using liboo are able to run XTA and take advantage of the devirtualization of dynamic calls. Therefore, the XTA implementation has to work with different frontends and has to make as few assumptions as possible about the code structure.

As some information is only available to the frontend and cannot be retrieved from liboo, two callbacks have to be implemented by the frontend in order for XTA to work. One callback gets called every time the analysis finds a call to a method where no graph is known, these are also marked as external by the frontend. For example, this is the case if the method is a library method, or is written in a different language. As these methods cannot be analyzed, the frontend has to determine whether the external method calls methods which are to be be analyzed.

The other callback is for detecting constructors and is further explained in Section 3.6.

After configuring these callbacks, the analysis, including the devirtualization process, is started via one method call. The following information is passed on at start:

1. A list of entry points. Usually, this is only the main method of a program. When compiling libraries, this list must include all publicly accessible methods, as all of these can be called from an application. If not every possible entry point is passed to XTA, the flow of object types cannot be reconstructed in a correct way which may result in incorrect devirtualization.

2. A list of those classes which are to be set live initially. These classes will be considered live in all methods.

3. Information about object types returned from methods which cannot be analyzed can be provided to enhance the result. Specifically, this is used to set

the Java class `String` live wherever a string is created using a string literal surrounded by quotation marks. This is necessary as the called method returns a primitive type instead of a reference to a `String` object.

XTA can be split into two parts: Part one is the reading of the program's code and the gathering of information. For each method all callers, field read and writes, parameter and return types as well as object creations have to be tracked. This is explained in detail in section 3.5. Part two is to use the information gained to propagate possible classes in use according to the rules in section 2.4.2.

The idea is to only read code from methods which are reachable. These are called on objects whose type is considered live in that context. To do so, the analysis starts reading the entry point methods. Going through a method, call sites, field accesses, and object creations are recorded. At every static call site, the target method is added to a work-queue of still-to-be-analyzed methods. The same is done at a dynamic call site where every possible call target is recorded. But only the methods on objects which are considered in use at the call site are added to the work-queue. As a class can become live in a method later on, all target methods of classes which are not live yet, have to be saved too. If a class is set live in a method later on, the store of saved call targets for that method is used to update all target sets for each call site in that method.

As long as the work-queue is not empty, the methods in the queue are analyzed. This also involves collecting all possible return and parameter subtypes. We do not need to consider primitives, but references to objects and arrays. For each reference type we collect the object type first and then all subtypes. We treat references to arrays similar: we get the type of the array and afterwards all possible subtypes. Doing so produces correct results, but it is not as precise as treating arrays like classes with only one field, which is proposed in [6]. The passing of types not available in Java, such as `struct` and `union` and function pointers, is not supported.

Together with the set of already analyzed methods, the queued methods form the set of reachable methods R (see section 2.4.2).

If the work-queue is empty, the second part of the analysis is started to find new methods to be analyzed. The propagation algorithm goes through each analyzed method's set of information and manipulates the sets of live classes with the help of the gathered data according to the rules in section 2.4.2. This is repeated until either a new method is added to the work-queue or the propagation algorithm does not change any set at all, which marks the end of the analysis.

Following pseudo code describes the procedure:

```
do {
        while(!is_empty(workqueue)) {
                method = get_element(workqueue)
                add_to_set(done_set, method)
                analyze_method(method)
        }
} while(propagate_live_classes())
```

Listing 3.1: As long as the propagation algorithm changes the live classes of methods, the work-queue of yet to be analyzed methods is emptied and analyzed.

The result of the XTA analysis is the information about live classes for each method and field. Using this information the possible call targets for each dynamic call are determined. Afterwards the graph of each reachable method is read again and each dynamic call which has only one call target according to XTA, is replaced by a static call.

In contrast to the implemented RTA, the actual call graph is built as well, as we have to store each caller for a method to propagate classes. The call graph is built on-the-fly while reading method graphs and propagating live classes until a fixed point of live classes is reached.

While XTA can determine the possible flow of live classes, it can fail to devirtualize even a dynamic call which can obviously be replaced with a static call, as in listing 3.2. We assume there is a class `B` extending class `A`, both defining a method `foo`, and a method `main` in which `A` and `B` are considered live by XTA. If now an object of type `B` is created and subsequently a call to `foo` is made, XTA is not able to devirtualize that call, as both classes `A` and `B` are live and both define `foo`. But they can be easily detected in the FIRM graph, as the graph resembles a dependency graph and the call to `foo` is directly dependent on the object creation. Thus by searching for patterns of dynamic calls dependent on objects created locally in that method, calls can be devirtualized, without the analysis results of XTA.

```
class A { void foo() {} }
class B extends A {
    void foo {}

    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        b.foo(); //Cannot be devirtualized by XTA
    }
}
```

Listing 3.2: Example code where XTA cannot devirtualize call to `foo()`.

This search and immediate replacement of virtual calls on locally created objects is done before XTA starts. Thus, we reduce the number of dynamic call sites the XTA has to consider.

3.2 The Propagation Algorithm

The propagation occurs iteratively. In each step, a method's live class will only be inserted in sets of callers or sets of called methods, and it will not be propagated further until the next iteration. This has to be done, as we do not build a directed graph of related sets and do not walk this graph from top to bottom. In each iteration, we apply the set-relations defined in section 2.4.2 to all analyzed method's live classes. As we have to calculate the intersection of two sets very often during propagation, keeping the looked-at sets as small as possible is essential for improving the run-time. Therefore, we consider three groups of live classes: Firstly, the *new set*, containing classes which are newly-added during propagation. Secondly, the currently onwards propagated classes are stored in the *current set*. Thirdly, the *propagated set* contains already propagated classes, which do not have to be considered during propagation.

Thus classes from a *current set* will be propagated to the *new set* of another method. After a propagation iteration, the content of a method's *current set* will be moved to the *propagated set*, the *new set* becomes the *current set* and an empty set is the *new set* now.

Important to note is that now one cannot rely fully on the propagation method to propagate classes to newly read methods. Live classes in the *propagated set* have to be propagated to a newly read method outside propagation iterations.

Another thing to consider is that we cannot add a class to the *current set* while we iterate over it. This happens for example, when we iterate over the *current set* after an iteration. The classes in the set are now considered live, and therefore we have to go through the list of previously collected call targets. If a call target is now live because it is the member of a class in the *current set* and it's graph is already analyzed, we have to propagate possible return types to the method. As we iterate over the *current set*, we have to add them to the *new set*.

3.2.1 Example

To illustrate the propagation algorithm, we consider the code in listing 3.3. The first thing the XTA does is analyzing the entry point methods, in this case the `main`-method. While walking the `main`-method's graph, the class `A` is set live due to the object instantiation, and thus is added to the `main`-method's *current set*. When the call to `getB()` is discovered, `getB()` is added to the work-queue and analyzed afterwards. The result of the analysis is that `B` is added to `getB()`'s *current set*. Now the work-queue is empty, and the propagation starts.

When analyzing methods, we save the caller information at the callee's. Thus, during propagation we iterate over all callers of a method. As the `main`-method does not have any callers, nothing happens. But when iterating over `getB()`'s callers, the `main`-method is found. Parameter and return types are propagated. As `getB()` does not have any parameters, only the return types are considered. The intersection between all subtypes of the return type and `getB()`'s *current set* is built, which is the class `B`. This class is now inserted into the `main`-method's *new set*.

Afterwards the propagation is finished, which leads to the next step: all types in the *new set* and *current set* are copied one set further. This is shown in figure 3.1 for the `main`-method's sets.

```
class A { }
class B extends A {

   public static void main(String[] args) {
      A a = new A();
      a = getB();
   }

   static B getB() {
        return new B();
   }
}
```

Listing 3.3: Example code to illustrate the analyzing of methods and the propagation of object types.

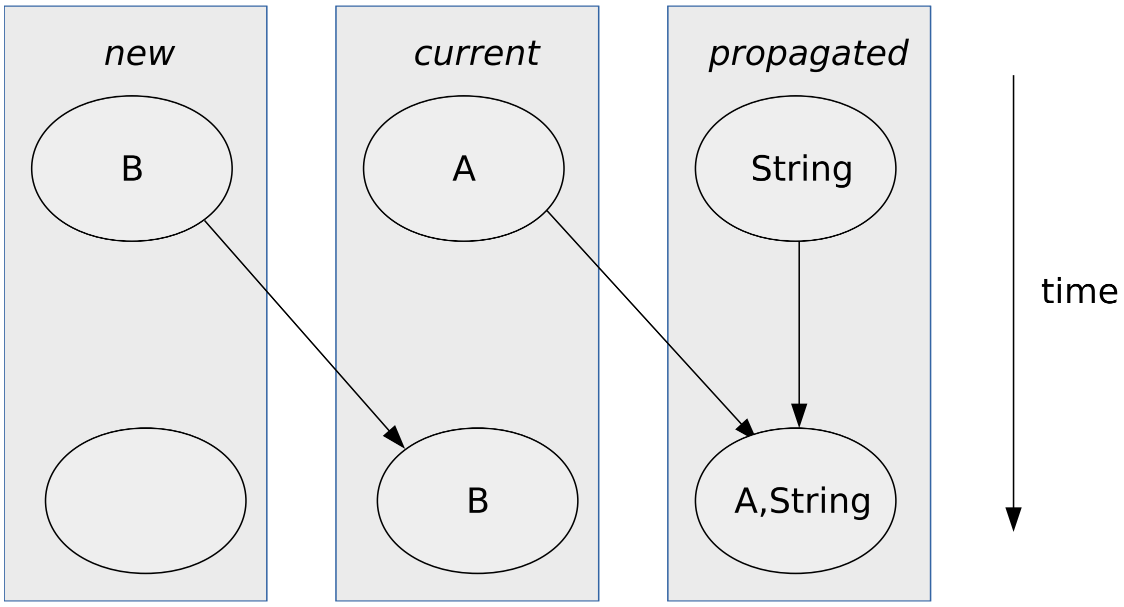


Figure 3.1: This figure shows how the classes in the three sets of the `main`-method are moved to the next one.

3.3 Devirtualizing Calls

After the execution of XTA, the actual devirtualization process starts. As information about which methods are possibly called exists now, only the graphs of active methods are read again. At each virtual call site we have to check whether more than one call target exists. If only one target exists, the call is to be devirtualized.

A call site can be identified in a FIRM graph via a `Call` node.

In order to devirtualize a method call, we have to identify every **`MethodSel`** node preceding a `Call` node and, given devirtualization is possible, replace the MethodSel node by an Address node. As devirtualization using XTA is the same as in RTA, the process is almost identical. The major difference is that XTA results in sets of possible call targets which not only depend on the *call entity*, but also the method where the call site is located.

3.4 Finding Possible Target Methods

As with devirtualizing calls, the procedure of finding possible target methods is the same as in the already implemented RTA. In theory, these are the steps needed to collect all targets: One uses the static type and walks the class hierarchy upwards until a method with a matching signature is found. This step is needed to cope with inherited methods which are not overwritten in the static type's class. After a fitting method is found, all subclasses of the static type are searched for methods overwriting the superclass's method. The details and FIRM-related differences are discussed in [9].

3.5 Recognizing Object Creations and Field Read and Writes

When walking through a method's graph, every node is passed, but only specific types of nodes are of interest. Creating an object can look differently in a FIRM graph, as the specific implementation is up to the frontend and the language specification. Usually the steps that are needed include memory allocation, setting a v-pointer to the virtual table, calling the constructor plus possible parent constructors and possibly the initialization of class members. Memory allocation and calls to a constructor are not feasible for identifying an object creation, as memory can be allocated without creating an object. Constructors are only called during object creation, but due to inheritance, multiple constructor calls can occur during the creation of one object. Also, as constructors are treated like methods in liboo, constructors cannot be easily identified without knowledge of the language and frontend used.

So the only option left to identify object creations is to recognize the setting of a v-pointer. Since this also marks the first possible occurrence of a virtual call afterwards, it is the best place to consider an object to be completely initialized for our purposes. The problem is that the v-pointer is a normal member of an object. Thus without knowing the member's name, which is chosen by the frontend, identifying the related store-operation is not feasible.

Therefore liboo introduces a flag node, called `VptrIsSet`, which marks the existence of a v-pointer [9]. So at every occurrence of a VptrIsSet node, we get the type information from the node and set the specific class live in that context.

Recognizing field reads and stores can be achieved without the help of a marker node. The reason is that for each read a load operation must occur and for each write there is a store operation. Thus at each load and store node the associated class member has to be retrieved. There are two possible cases: Either a member or an address node is directly connected to a load/store node. The class member information can be read from both node types. If the field stores a primitive value instead of an object, the read or write is skipped.

3.6 Dealing with Constructors

In liboo, constructors are normal method entities, and there is no flag to indicate that a specific entity is a class constructor. But as our XTA implementation is designed and tested with the Java frontend bytecode2firm in mind, we assume the following to be true: Firstly, constructors do not return references to initialized objects. Secondly, constructors call one or more super-class constructors if there is a super-class.

Without detecting and handling constructors specially, the following happens: For example, if there is a class `B` which extends a class `A`, the default constructor of `B` calls the constructor of `A` (see listing 3.4). In Java, the dynamic type of `this` in the constructor of `A` is `B` as an object of type `B` was created. Thus, class `B` is live in the constructor of `A`. But our propagation algorithm only propagates live classes from one method to the other if the parameter types or return types match. As the `this`-pointer is only considered when passing arguments to external methods (see section 3.7), class `B` is not set live in the constructor of `A` without treating constructors specially.

```
class A { public A() {} }
class B extends A {
   public B() {
        super(); //Call to A::A()
   }

   public static void main(String[] args) {
      B b = new B();
   }
}
```

Listing 3.4: Example code where the constructor `B` calls the constructor of `A`

In order to solve this problem, we have to detect constructor methods and treat parameter passing as with external methods. As only the frontend is capable of detecting constructors, a callback to the frontend is introduced. If the frontend identifies a method as a constructor, the parameter list is extended to the zeroth argument which is the `this`-pointer.

In other languages the initialization of objects is done differently. For example, base-class constructors are called first in C++. Thus, the `this`-pointer points to the created sub-object, so a virtual call in a base class constructor will never result in the calling of an overwritten method in the derived class [18]. Still, our special handling of constructors does not result in falsely devirtualized calls.

3.7 Dealing with Incomplete Programs

According to section 2.4 static analyses inspecting code must have the ability to read the whole program in order to produce a sound call graph. But the closed-world assumption is not tenable under realistic circumstances, as many programs use external libraries, especially language's standard libraries. One possibility is to fulfil the assumption and compile the library's[1] code together with the application code, thus eliminating unavailable code. But often libraries are only available as a compiled unit, rendering static code inspection impossible. Even if the library's source code is available, analyzing small programs using a language's standard library can make the analysis more time- and resource-consuming. For example, the call graph creation of a simple "Hello World"-routine in Java using SPARC [19] can take up to 30 seconds and produces a graph with 5,313 reachable methods and more than 23,000 edges [20].

The second possibility is to cope with unavailable method code directly, and to set up rules to approximate the effect of library code. In order to still be able to build a sound call graph, we cannot assume a closed-world anymore, but have to deal with application classes extending library classes, the calling of library methods and the overriding of library methods, thus making these methods accessible from within the library.

The idea is to replace the closed-world assumption with the separate compilation assumption [20]: A library can be compiled separately from the application. Thus, following rules are assumed to be valid:

1. A library class cannot extend an application class or interface.

2. A library method cannot instantiate an object whose type is an application class, except with the use of dynamic class loading. As the Java frontend bytecode2firm does not support dynamic class loading, we are not taking it into account further.

[1]Below, the singular *library* is used to refer to all linked libraries.

3. A library method can instantiate any object whose type is a library class, and can return it if the return type allows it.

4. A set S_E is introduced containing all library and application classes which can be accessed from within the library. An application class C is inserted into the set if one of the following conditions is met:
   - An object of type C is passed as an argument to a library method.
   - An application method called from within the library can return an object of type C.
   - An object of type C is stored in a field of a library object.

5. An application method m in class C can be called from a library class under following circumstances: m has to override a library method and C or subclass of C must be in S_E.

Another reason for not being able to read the whole program is code written in another source language. Java for example gives programmers the possibility of native methods which are not written in Java via the Java Native Interface [21]. As the Java Library uses native methods for system-specific code, one cannot assume that if the application code does not contain native methods no native method will be called.

In libFirm methods with no graph exist. These are either native methods, library methods or even native library methods. Whereas library methods can be distinguished from native methods by reading liboo's external flag, library methods cannot be distinguished from native library methods. Thus, these are treated like normal library methods.

Explicitly, five parts are to be considered:

1. native methods

2. an application method calling a library method, therefore passing arguments to S_E

3. a field read of a library field, or the write to one

4. an application method overwriting a library method, thus being accessible in the library

5. all library methods have to be inserted into S_E

We implement the rules stated by Tip and Palsberg section 2.4.2 to model the flow of classes between application and library.

In order to cope with native methods, we assume the following: a native method does not call a normal method, or writes to a field, thus the only interaction point is the return value. Secondly, a native method may return all possible classes, specifically all subtypes of the return type. Therefore, whenever a method with no graph is called, all subtypes of the return type are set live resulting in these to be passed to the callers.

The second part is handled by checking whether a library method is involved each time we propagate live classes between methods. If so, we propagate between S_E and the other method's live classes. An important difference is that we also propagate any type passed via the `this`-pointer to a method. A library method is recognized by checking whether the method's owner is external.

Field access to library fields is done the same way as with library methods: propagation takes place between S_E and a method's live classes. A library field's owner is either an external class, in case of an instance field, or the firm-specific `GlobalType` which stores all static fields and methods. Thus, a library class's static field cannot be identified as being from a library class due to libFirm limitations. The only way to still produce correct results is to handle all static fields as library fields. This may introduce more live classes in a method on average, but is necessary to stay correct.

The last two points are done by walking the complete class hierarchy before starting the analysis. Each external class is inserted into the set S_E. Afterwards, for each class in S_E we check whether an application subclass overwrites a method defined in the external class. At the end of each propagation iteration, we check whether one of these methods has to be considered in use because its class is in S_E now.

Often there are library methods which do not pass arguments to other library methods or fields. For example, the `Object.clone()` method does not call another library method, neither is the parameter stored in a field. Another example is the constructor of the class `Object` which is called whenever any object is initialized. If it is known that these methods do not leak the object to other library entities, we can treat these like application methods. Therefore, the frontend has the ability to specify a list of library methods which are to be treated like application methods.

4 Evaluation

This chapter is about evaluating the implementation of XTA. As the goal was to improve the existing RTA, this will be our baseline. The improvement of several aspects are measured and evaluated. Beside the success of the devirtualization process, these include the identification of unreachable methods, the handling with unaccessible library code, and the time XTA needs to analyze the program.

To be expected is the following: XTA is more precise, but takes more time to compute in comparison to RTA. Tip and Palsberg [6] determined that XTA was able to resolve 12.5% of the polymorphic calls RTA discovered to monomorphic ones. On average XTA was five times slower than XTA.

The experiments are run on an Intel Core i5 4770HQ running at a clock speed of 2.2GHz. The host system is a MacOS X 10.13.4, whereas the guest system is a 64-bit Linux Mint with kernel version 4.10.0-38-generic. 8GB of RAM are available to the guest system.

4.1 Evaluated Programs

In order to compare XTA to RTA, the following programs were compiled: jython version 2.2.1, from the benchmark suite SPECjvm98 [22] the benchmarks mpegaudio, and jack, and from SPECjvm2008 [23] the benchmark Tidy.

Other benchmarks from the SPEC suits could not be compiled due to missing support of features in bytecode2firm. One of the missing features was missing instruction support for `jsr` and `ret`. These are instruction which were being used for try-catch blocks.

Program characteristics such as number of classes, methods, and fields are shown in figure 4.1. In the table, external classes and methods are library classes and methods, and native methods are counted as normal methods, not as external ones. The number of fields are the number of reference-typed fields found by XTA.

All programs are written in Java and are compiled using bytecode2firm[1], liboo[2], and libFIRM[3] with the gcj option to turn on the GCJ runtime.

As one can see, while the benchmarks consist of less than 100 classes, jython has 743 classes. The different number of external classes result from the loading procedure of Java classes: Only necessary classes are loaded, thus library classes which are not in use are not loaded.

[1]140a0c95f7c39c5299ce95702e9331b9df38b0a8
[2]115c6d57896ba74427d0a47f87949a7b56f94a03
[3]b3212b7a72d3042564781d085f2eae496726f953

Name	#classes	#ext. classes	#methods	#ext. methods	#fields
jython2.2.1	743	226	6328	2293	636
mpegaudio	62	175	362	1371	87
jack	67	173	373	1386	78
Tidy	99	94	656	871	122

Figure 4.1: Characteristics of compiled programs.

Name	#reached methods		#static c.s.		#virtual c.s.		#interface c.s.	
	XTA	RTA	XTA	RTA	XTA	RTA	XTA	RTA
jython2.2.1	5274	6157	23333	24786	9840	13193	251	320
mpegaudio	159	330	638	1989	188	465	0	15
jack	382	385	2601	2465	1156	1292	83	83
Tidy	408	503	5120	5644	1073	1509	6	9

Figure 4.2: This table contains the number of reached methods, and static, virtual, and interface call sites reached by XTA.

4.2 Devirtualization Results

To evaluate XTA, the number of reachable methods and the number, or percentage, of devirtualizable calls has to be compared to the respective results of RTA.

In order for the results to be comparable, both analyses have to correctly handle libraries, as well as native methods. The current approach of handling native methods and libraries in XTA is explained in section 3.7.

To match the behaviour of XTA, we change RTA's procedure to cope with native methods: all subtypes of the return type are set live. In the context of RTA, this implicates that all subtypes are considered live to the whole program. As some native methods in the Java Class Library do return the class `Object`, this leads to the problem that RTA is as effective as CHA when these native methods are reachable. This is due to the fact that all classes inherit from the class `Object` in Java.

Handling libraries in RTA is done by setting all library classes live initially[9].

The numbers of reached methods, and the numbers of reached static, virtual, and interface call sites are shown in figure 4.2.

Figure 4.2 shows that XTA reaches less methods than RTA, but on average cannot devirtualize more call sites. A problem with the data collection is that more reachable methods equals more virtual call sites, thus possibly more devirtualized call sites (see figure 4.3). This means RTA can devirtualize more call sites than XTA, even in percentage terms. For example, XTA only devirtualizes 67% of virtual call sites in jython whereas RTA achieves 74%. This difference is even more severe when

Name	#virtual calls devirt		#interface calls devirt		#init devirt	
	XTA	RTA	XTA	RTA	XTA	RTA
jython2.2.1	6597 (67%)	9705 (74%)	139 (55%)	138 (43%)	1387	-
mpegaudio	88 (46%)	398 (85%)	0(100%)	15(100%)	64	-
jack	456 (39%)	622 (48%)	26 (31%)	27 (33%)	150	-
Tidy	572 (53%)	1185 (78%)	0 (0%)	0 (0%)	114	-
avg.	51%	71%	47%	44%		

Figure 4.3: Devirtualization results of XTA an RTA.

Name	XTA		RTA	
	mean	std. dev.	mean	std. dev.
jython2.2.1	39.91	1.09	1.10419	0.09546
mpegaudio	0.04571	0.00523	0.04613	0.00338
jack	0.06658	0.0067	0.0469	0.00419
Tidy	0.07939	0.01187	0.06991	0.00601

Figure 4.4: This table shows the mean run-time of XTA and RTA, and their standard deviations.

analyzing the benchmark mpegaudio, where XTA achieves 46% and RTA 85%.

Still, less reachable methods are the result of a more precise analysis.

Figure 4.3's last column are the numbers of devirtualized calls prior to XTA. We analyze the method and virtual calls on locally created objects are replaced with static calls. Thus, these calls are devirtualized, but are part of the number of static calls in figure 4.2. They are also not taken into account when calculating the percentage of devirtualized calls.

To consider is the fact that non-static, non-final Java methods are always virtual and can be overwritten. In contrast, C++ uses the keyword `virtual` for explicitly marking methods as virtual. Therefore, many virtual calls can be devirtualized using CHA or RTA which can be seen our results, too.

A detailed comparison between XTA and RTA is done by Tip and Palsberg [6].

Another aspect to compare is the run-time of the analyses which is shown in figure 4.4. To measure the time spent executing the analyses the FIRM internal `ir_timer` methods are used. These methods measure the wall-clock time spent in the analysis. The spent CPU time is not measured.

Noticeably, XTA is not much slower than RTA when analyzing the benchmarks. But the analysis of jython using XTA is 40 times slower than RTA. The reason for this is discussed in the next section.

Name	#methods		#static c.s.		#virtual c.s.		#interface c.s.	
	XTA	RTA	XTA	RTA	XTA	RTA	XTA	RTA
jython2.2.1	2323	3271	13353	15451	5000	7984	225	298
jython2.2.1 (n)	5274	6157	23333	24786	9840	13193	251	320

Figure 4.5: This table contains the number of reached methods, and static, virtual, and interface call sites reached by XTA and RTA.

4.3 Native Method Handling

In order to produce valid call graphs, we have to assume that a native method can return all subtypes of its return value. This is implemented in XTA, and to compare results this was adapted in RTA. This can lead to following scenario: A native method's return type is `Object` in Java, thus all classes are set live. If we consider this to happen in RTA, this means that the results are the same as those gained via CHA. In many object-oriented languages, there exists one type at the top of the class hierarchy all objects inherit from. Thus, we cannot assume this to be an edge case.

In this section, we evaluate the effect of setting all subtypes of native method's return value live. Therefore, we disable this feature in both, RTA and XTA.

It has to be mentioned that the result is not a valid call graph, and therefore calls may get devirtualized falsely. As jython is the biggest of the four programs, we are only considering jython in this section.

Figure 4.5, figure 4.6 and figure 4.7 have the same columns as in the previous section. But this time, we compare the results of jython compiled with RTA and XTA, each with native method support and without it. The rows *jython2.2.1 (n)* show the results for the version with native method support. The content is the same as in the previous section.

As expected, we see that all values of analyses without native method support in figure 4.5 are lower. Approximately 3.000 additional methods are reachable due to the handling of native methods. The results are 10.000 more static call sites and 5.000 virtual ones. About 3.000 more call sites are devirtualized.

While this is the only valid over-approximation of a call graph without assuming anything about native methods, the results show that the approximation loses a lot of precision. And we see that this is true for XTA as well as for RTA.

Figure 4.7 shows the mean run-time of both XTA and RTA versions. Clearly, XTA is slowed down significantly by the handling of native methods. Not taking native methods into account leads to a run-time 20 times faster. As most of the time is spent in the propagation algorithm, the issue seems to be that the sets become too big which leads to the longer run-time.

Calculating the average set sizes for the live classes of each method uncovers the following: XTA without native method support has an average set size of 20 for

Name	#virtual calls devirt		#interface calls devirt		#init devirt	
	XTA	RTA	XTA	RTA	XTA	RTA
jython2.2.1	3811	6576	145	155	1387	-
jython2.2.1 (n)	6597	9705	139	138	1387	-

Figure 4.6: Devirtualization results of XTA an RTA.

Name	XTA		RTA	
	mean	std. dev.	mean	std. dev.
jython2.2.1	1.7039	0.09424	0.93508	0.03865
jython2.2.1 (n)	39.91	1.09	1.10419	0.09546

Figure 4.7: This table shows the mean run-time of XTA and RTA, and their standard deviations.

methods and fields. With native method support this number reaches 298 which explains the longer run-time.

5 Conclusion and Future Work

This thesis was about improving the devirtualization process in libFirm. Therefore, we implemented XTA, an analysis which determines a set of live object types for each method. XTA is based on RTA which determines a set of live classes for the whole program. Obviously, by not only storing which object are in use, but also their locations, the outcome is a more precise call graph. Our test results show that this is true. XTA is able to reduce the number of reachable methods to almost the half. As RTA devirtualizes calls in methods deemed by XTA as unreachable, RTA has a higher number of devirtualized call sites.

The detection of unreachable methods leads to removing these, a feature currently not implemented in XTA or bytecode2firm. This reduces the size of the executable which can be an important objective.

While Tip and Palsberg [6] report that XTA is five times slower than RTA, our implementation is significantly slower than the currently in libFirm available RTA. The reason is that the goal was to implement an analysis which can built a sound call graph. Therefore, XTA has to deal with native methods. These are methods whose implementation is unknown, and therefore we have to assume that any possible type may get returned. This leads to a significantly increased average set size, which is the reason for the longer run-time.

Thus, if compile time is deemed to be important, either the current XTA implementation needs to be improved, or one has to switch to the still available RTA which ignores native methods. Since the problem can be narrowed down to the propagation algorithm, improving it will result in significant run-time improvements. An alternative is to introduce an option for turning off XTA's native method handling. This seems to be an easy solution but in fact, it does only circumvent the problem.

XTA was designed to work with the Java frontend bytecode2firm. Therefore, only Java language features are explicitly supported. Exceptions are Java features which are not yet supported in bytecode2firm. This includes, amongst other things, dynamic class loading, reflection, try-catch blocks and exceptions, and anonymous functions via lambda expression.

Additional languages were not tested, and the XTA result may be incorrect if used with these. Especially, the use of function pointers, available for example in C++, is not considered in the current implementation. But also structs and unions as parameter and return types are not supported. It needs minor code modifications for these to work.

5.1 Future Work

The basic XTA implementation was done in this work. But there are still things which can be improved:

1. Currently, only a simple propagation algorithm is implemented. As much runtime is spent propagating classes, it is one of the main parts where improving the implementation may lead to a considerable performance increase.

2. Tip and Palsberg [6] model arrays as classes with only one instance field. This may increase the result's precision.

3. Another possibility is to extend the idea of the already introduced devirtualization of calls to locally created object to an inter-procedural approach. Namolaru [24] demonstrates this approach in combination with RTA.

Additionally, other analyses show also promising results, for example the analyses presented in section 2.5. One could use the work of Sundaresan et al. [15], a modified points-to analysis, and combine it with one advantage of graphs in SSA form, i.e. the replacement of local variables with SSA values, to create an analysis where only set relations between fields, and return and parameter types are created.

Bibliography

[1] M. Braun, S. Buchwald, and A. Zwinkau, “Firm—a graph-based intermediate representation,” Tech. Rep. 35, Karlsruhe Institute of Technology, 2011.

[2] G. Lindenmaier, M. Beck, B. Boesler, and R. Geiß, “Firm, an intermediate language for compiler research,” Tech. Rep. 2005-8, Mar. 2005.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” vol. 13, pp. 451–490, 10 1991.

[4] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, “Simple and efficient construction of static single assignment form,” in *Compiler Construction* (R. Jhala and K. Bosschere, eds.), vol. 7791 of *Lecture Notes in Computer Science*, pp. 102–122, Springer Berlin Heidelberg, 2013.

[5] W. Landi, “Undecidability of static analysis,” *ACM Lett. Program. Lang. Syst.*, vol. 1, pp. 323–337, Dec. 1992.

[6] F. Tip and J. Palsberg, “Scalable propagation-based call graph construction algorithms,” in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’00, (New York, NY, USA), pp. 281–293, ACM, 2000.

[7] D. F. Bacon and P. F. Sweeney, “Fast static analysis of c++ virtual function calls,” in *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’96, (New York, NY, USA), pp. 324–341, ACM, 1996.

[8] J. Dean, D. Grove, and C. Chambers, “Optimization of object-oriented programs using static class hierarchy analysis,” in *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP’95)*, pp. 77–101, 07 1999.

[9] S. Knoth, “Optimierung von dynamic dispatch mit rapid type analysis für firm,” Apr. 2015.

[10] J. Palsberg and M. I. Schwartzbach, *Object-oriented Type Systems.* Chichester, UK: John Wiley and Sons Ltd., 1994.

[11] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language.* PhD thesis, 1994.

[12] B. Steensgaard, “Points-to analysis in almost linear time,” in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’96, (New York, NY, USA), pp. 32–41, ACM, 1996.

[13] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, “Points-to analysis using bdds,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI ’03, (New York, NY, USA), pp. 103–114, ACM, 2003.

[14] B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. Lu, “An efficient tunable selective points-to analysis for large codebases,” in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, (New York, NY, USA), pp. 13–18, ACM, 2017.

[15] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, “Practical virtual method call resolution for java,” in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’00, (New York, NY, USA), pp. 264–280, ACM, 2000.

[16] U. Hölzle, C. Chambers, and D. Ungar, “Debugging optimized code with dynamic deoptimization,” in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI ’92, (New York, NY, USA), pp. 32–43, ACM, 1992.

[17] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, “A study of devirtualization techniques for a java just-in-time compiler,” in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’00, (New York, NY, USA), pp. 294–310, ACM, 2000.

[18] “C++ using this in ctors.” `https://isocpp.org/wiki/faq/ctors#using-this-in-ctors`.

[19] O. Lhoták and L. Hendren, “Scaling java points-to analysis using spark,” in *Compiler Construction, 12th International Conference, volume 2622 of LNCS*, pp. 153–169, Springer, 04 2003.

[20] K. Ali and O. Lhoták, “Application-only call graph construction,” in *ECOOP 2012 – Object-Oriented Programming* (J. Noble, ed.), (Berlin, Heidelberg), pp. 688–712, Springer Berlin Heidelberg, 2012.

[21] “Java native interface specification.” `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`.

[22] “Specjvm98.” `https://www.spec.org/jvm98/`.

[23] “Specjvm2008.” `https://www.spec.org/jvm2008/`.

[24] M. Namolaru, “Devirtualization in gcc,” in *Proceedings of the GCC Developers’ Summit*, pp. 125–134, 2006.

Erklärung

Hiermit erkläre ich, Daniel Biester, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift