Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Devirtualization in GCC

Mircea Namolaru
IBM Haifa Research Lab - HiPEAC Member
namolaru@il.ibm.com

Abstract

A major optimization for object oriented languages is converting dynamically bound function calls into (statically bound) direct calls, a process called devirtualization. This saves the dynamic dispatch overhead, and more importantly, enables further inlining of these function calls. For devirtualization we designed an extension of the Rapid Type Analysis (RTA) algorithm, a fast and effective algorithm [1]. The resulting algorithm combines RTA with a simple data-flow analysis.

We have an initial version of devirtualization for C++, implemented in GCC. We describe how the implementation makes use of the existent GCC code, and how it deals with the complexities of the C++ language. Finally we discuss the major implementation issues for completing the implementation.

1 Introduction

In order to support abstraction, object-orientated languages support dynamic binding of methods based on the run-time of the object. This presents a compiler with a problem, as it has to generate code to activate the method at run-time. In addition as the method invoked is unknown at compile-time, valuable optimizations opportunities may be lost. As dynamic binding is extensively used by object-orientated programmers, this may cause a significant overhand in performance.

In order to statically bind a method, we must be able to statically determine the type of the object upon which the method is invoked. A number of different analyses have been developed for C++ in an attempt to solve this problem.

One of them is Rapid Type Analysis (RTA), introduced in [1]. An in depth description of the algorithm, its performance and its implementation can be found in [2]. The RTA algorithm is a simple algorithm, and yet its performance compares very well versus other much more complex type inference algorithms.

Some simple data-flow analysis may succeed in devirtualizing calls for which RTA analysis have failed. For instance, a simple (intra-procedural and/or inter-procedural) type propagation algorithm may find that the object upon which a virtual method is invoked, is an object returned by a new operator (therefore its type is know statically). In spite of its simplicity, such an analysis may be quite effective, and complements RTA, creating a more powerful devirtualization algorithm.

We have designed an algorithm that combines RTA with a simple type propagation algorithm. The resulting call graph computed by our algorithm is more accurate then the one computed by RTA and/or class propagation applied separately and this may cause better devirtualization. The differences between the original RTA algorithm and this algorithm are discussed further.

We have a partial implementation of this algorithm in GCC for C++ and we are working to complete it. RTA is conceptually simple, but due to the complexity of C++ and to the current GCC infrastructure, its implementation raises a number of issues that are discussed further.

2 The algorithm

As the RTA algorithm is the basis of our work it is presented first in a form adapted to our needs, which is slightly different from the one appearing in [2](e.g. we build the call graph on the fly)

RTA assumes that a Class Hierarchy Graph (CHG) that describes the inheritance relationship between classes is available. Another prerequisite for RTA is a call graph with only the virtual calls not resolved. The RTA analysis will compute the possible targets for a virtual call and the ones that have a single possible target can be devirtualized.

The RTA algorithm is outlined in Figure 1. It maintains a list of methods. This list is initialized with the root node of the call graph. Initially, no class in the program is considered as being live. The algorithm also maintains a list of the already visited call sites called the live call sites.

Each method in the list is analyzed in turn. For each virtual call site in the method, the static class of the object upon which the method is invoked and its live subclasses are used to find the possible targets of the virtual call and to build the corresponding edges in the call graph. As other subclass may be marked live at a later time, we maintain information about live call sites.

We mark as live all the classes instantiated in this method. When a new class is found to be live, new methods may be reached via live virtual calls invoked on a base of the new instantiated class. The information on live virtual call site is used to find these methods and to build the corresponding edges in the call graph.

At the end of the algorithm we have found all the live classes and resolved the virtual call sites in the call graph. Virtual call site with a single target can be devirtualized.

In our algorithm we integrate RTA with a simple data-flow analysis that propagates information about the types of objects passed as arguments. The algorithm is outlined in Figure 2.

If an argument of a call is an object returned by a new operator or it is a formal parameter of the method containing the call, the argument is marked as `df-dep` (data flow dependent). A virtual call invoked on a `df-dep` object (considered to be the first argument of the call) is marked as `df-dep` virtual call.

The lattice of values used for type propagation is `unknown`, `df-dep` and `rta`, where `unknown` is the minimal element and `rta` is the maximal one. For `df-dep` values there is a secondary value with the classes reaching the `df-dep` argument. The rules for propagation are described in Figure 3.

One alternative propagation rule would be that if two different classes are propagated to a `df-dep` argument, then the value propagated further is `rta`. But as the propagated classes are from the same hierarchy, maintaining all the classes reaching a `df-dep` argument could be implemented rather efficiently.

```
build_virtual_calls_targets:

 for each method m in the call graph
  visited (m) = false;
 for each class cls in the program
  live (cls) = false;
 list_methods = root_method;
  live_call_Sites = empty

 while list_methods is not empty

  remove m head of list_methods
  if (visited (m))
   continue

  for all call sites cs of m
   if is_virtual (cs)
    /* Based on the static type at cs and
       its live subclasses, find the possible
       targets at cs. */
    build_edges_cv (static_type (cs), m)
    add cs to live_call_sites
   else if is_not_virtual (cs)
    for each target tm in targets (cs)
     add tm to list_methods

  for each class cls instantiated in m
   if (live (cls)
    continue
   live (cls) = true
   for each call site cs in live_call_sites
    /* Find the method invoked at cs if
       dynamic type is cls.  */
    tm = target_cs_cls (cs, cls);
    if (tm == NULL) continue
    build_edge (m, tm)
    add tm to list_methods

  visited (m) = true;
```

Figure 1: The RTA algorithm

Initially, each class parameter of a method has the value `unknown`. A `df-dep` argument of a call that is a formal parameter of its enclosing method is initialized to the value `unknown`. A `df-dep` argument that is an object propagated from a new operator is initialized to the value `df-dep` (and the secondary value to the class instantiated by new). A non `df-dep` argument receives the value `rta`.

In the original RTA algorithm, every instantiated class assumes that all the live virtual calls may be reached by the new created object. Our algorithm optimistically assumes for a new instantiated object that `df-dep` virtual calls are not reached, letting data-flow analysis to find out if the call is reached or not.

For a `df-dep` virtual call, the computation of possible targets of the call is based on the type information propagated to the call, and not on the live classes information as in RTA.

Our algorithm requires that type information reaching the formal arguments of a method be propagated further in the call graph. Each time a method is reached, the type information of its arguments needs to be propagated to its call sites, and from there to the current targets of the call. A method is added to the method list each time new type information reaches its parameters.

Basically, this algorithm tries to infer the type of `df-dep` arguments via a simple data-flow analysis (propagation of type of objects returned by a new operator). For the rest of the arguments, the RTA analysis is used.

As in RTA, at the end of the algorithm we have found all the live classes and resolved the virtual call sites in the call graph.

3 An example for modified RTA

To illustrate the algorithms from the previous section and the differences between them we will use the example from Figure 4.

In the example, we have a simple class hierarchy, where B is a subclass of A and C a subclass of B. The class B overrides the methods foo and foo1 (A::foo and A::foo1 not shown), and the class C overrides the method foo1.

There are two virtual call sites, one in foo2 and the other in B::foo which is a `def-dep` call site.

```
build_virtual_calls_targets:

 for each method m in the call graph
  visited (m) = false
  df_init (m)
 for each class cls in the program
  live (cls) = false;
 list_methods = root_method
 live_call_Sites = empty

 while list_methods is not empty

  remove m head of list_methods
  if (visited (m))
   for all call sites cs of m
    for each target tm in targets (cs))
     /* Propagate type information from the
     formals of the method to the df-dep
     arguments in the call site.  */
     df_prop (m, cs)
     /* Check if new type information is
       propagated to the target.  */
     if (df_merge (cs, tm))
      add tm to list_methods
   continue

  for all call sites cs of m
   if is_virtual (cs) && not_df_dep (cs)
    /* Based on the static type at cs and
       its live subclasses, find the
       possible targets at cs. */
    build_edges_cv (static_type (cs), m)
    add cs to live_call_sites
   else if is_virtual (cs) && df_dep (cs)
    /* Based on the type information
       propagated at cs, find the possible
       targets at cs.  */
    build_edges_cv (propagated_type (cs), m)
   else if is_not_virtual (cs)
    /* Check if new type information is
       propagated to the target.  */
    for each target tm in targets (cs)
     if (df_merge (cs, tmp) or not visited (tm)
       add tm to list_methods

  for each class cls instantiated in m
   if (live (cls)
    continue
   live (cls) = true
   for each call site cs in live_call_sites
    /* Find the method invoked at cs if
       dynamic type is cls.  */
    tm = target_cls_cs (cs, cls);
    if (tm == NULL) continue
    /* Check if new type information is
       propagated to the target.  */
    if (df_merge (cs, tm) or not visited (tm))
     add tm to list_methods
    build_edge (cs, tm)

  visited (m) = true;
```

Figure 2: The modified RTA algorithm

```
prop (unknown, v) = v, v any value

prop (v, rta) = rta  , v any value

prop ((df-dep,A),(df-dep,B)) = (df-dep,(A, B))
```

Figure 3: Type propagation rules

The method list is initialized with foo2. First we find that the class C is instantiated and that B::foo is reachable. Then we discover that C::foo1 is reachable and that the class B is instantiated.

From this point onward, the two algorithms differ. RTA will consider that B::foo1 is reachable (via the call site from B::foo), and that the class A is instantiated. This will find two new methods reachable via the two call sites A::foo and A::foo1. So in this case, RTA will find that the classes A, B and C are live and no call site can be devirtualized.

If type propagation is done afterward, it will succeed to devirtualize the call site from B::foo. However, since the class live information remains the same, the other virtual call site could not be devirtualized.

With the modified RTA algorithm, after class B has been instantiated, we will not consider the method B::foo1 reachable. As again B::foo is reached (via the call site from foo2), we will check if new type information is propagated to it. This doesn't happen in this example, and the algorithm ends. The algorithm will find that the classes B and C are live. Since both call sites have a single target in the call graph, both can be devirtualized.

4 RTA issues

Since our algorithm is based on RTA, it shares its requirements and limitations which are de-

```
// B subclass of A
// C subclass of B
static A *pn;

B:: foo1 (A *p) {
  new A;
}

C:: foo1 (A *) {
  new B;
}

B:: foo (A *p) {
  p->foo1 ( );
}

foo2 () {
  p = new C;

  pn->foo (p);
}
```

Figure 4: An example for modified RTA

scribed further.

4.1 Type safety

RTA assumes that the dynamic type of the object upon which a virtual call may be invoked is a subclass of its static type. In C++, this assumption may be invalidated via a downcast as it is possible to see in Figure 5(a). This code is not type-safe, and it may cause a run-time exception if foo is not defined in A. If foo is defined in A, many C++ implementation (including GCC) may invoke it, which may or may not be what the programmer expected. But in this case RTA will decide that the target of this virtual call is B::foo and change the behavior of the program. As we see in Figure 5(b), in the presence of a downcast RTA may work perfectly well. To statically differentiate between such cases is not always possible.

Two possible solutions are discussed in [1]. The first is to not apply RTA if a downcast is detected in the program. As downcasting is a common C++ practice, this may restrict too

```
// B subclass of A
void *obj = (void *)new A
B *obj1 = (B *) obj
obj1->foo

    (a)


void *obj = (void *)new B
B *obj1 = (B *)
obj obj1->foo

     (b)
```

Figure 5: Downcast examples in C++

much this optimization. A better solution is to print a message if a downcast is encountered. The message will indicate that RTA may change the behavior of the program for truly unsafe downcasting.

4.2 Whole program analysis

In order to ensure the correctness of RTA, the entire program code must be analyzed (otherwise some instantiation of a class may be missed). This is not always possible. In many cases, part of the code is supplied in libraries whose code is not available. Following [2], we show the modifications required to RTA to handle incomplete programs.

We differentiate between classes internal to the program and classes exported by libraries. The classes internal to the program are not know to the libraries, hence they cannot be instantiated there, but their methods can be called from libraries. This may happen if their address have been taken in the program. Therefore, we need to consider all the methods whose address have been taken in the program as roots in the call graph (in the algorithms described previously, the initial list of methods should include them). Another change required is that classes exported by the libraries are initially considered live.

We must also consider the case when the program subclasses a library class and overrides one of its methods. When such a subclass in instantiated, all its methods that override methods in the library code must be assumed to be reachable (as they may be invoked by a virtual call in the library) and inserted in the list of methods.

The whole class hierarchy of the program is known. The compiler may analyze the class hierarchy and mark the standard C++ libraries classes as exported. The rest of the classes are considered internal to the program. If the program exports also other libraries beside the standard C++ libraries, information about their exported classes should be provided by the user.

5 Implementation

In this section, we describe several implementation problems and discuss the current stage of the implementation, as well as future work items.

5.1 Class instantiation

The algorithm needs to detect all the instantiated classes in the program. In order to do this, we find all invocations of constructors in the program. Constructors for sub-objects part of a base in a derived type are not considered to instantiate a class.

The constructors are identified at the gimple (or in the development branch used at the SSA) level, with the help of a C++ hook. The advantage of this approach is that an easy adaptation of this analysis to other languages is possible. The problem is that some constructors may be inlined by the front end and are not appearing at gimple (or SSA) level. However, we thought that would be easier for a given language to modify the front end, and let the regular inlining at gimple (or SAA) level to inline the constructors.

5.2 Class Hierarchy Graph

The implementation of the RTA algorithm needs a data structure to represent the class hierarchy graph (CHG). In this graph all nodes represent classes, and the edges describe the hierarchy relationship. We need to be able to reach from a base all its immediate subclasses, and from a subclass all its direct bases. Also, we want to annotate the nodes with the specific information needed by our analysis. We have implemented the CHG as a separate data structure internal to our analysis. There is a hash table that provides a mapping from a type to its corresponding node in the CHG.

The information about the bases of a given class is already available in GCC in the binfo (base info) trees build by the front end. As a class is instantiated, we complete the CHG with edges from a base to its subclasses. This is done by an upward traversal of the CHG starting with the class instantiated. For each edge (subclass, base) traversed, an edge (base, subclass) is constructed. If a node in CHG has no downward edges to its subclasses, none of its subclasses have been instantiated (or it is a leaf node). The information about instantiated classes is also kept in the CHG nodes.

5.3 Resolving a virtual call site

In this section, we show how the information about instantiated (or live) classes is used to resolve a virtual call site in the call graph. For such a call, the static class of the object upon which the method is invoked is available. The instantiated classes in the subgraph rooted at this class provide the possible dynamic types for this object. For every couple (static type, dynamic type) we find the method that will be invoked at run-time. These methods are the targets of the virtual call analyzed.

We have implemented a method that given the static and dynamic type upon which a virtual call is invoked returns the method invoked (a simple variant of this method was already existent in GCC for the case when the dynamic type and the static type are the same). Such a method is dependent on how the object model (the layout of objects, the virtual method tables etc) is implemented. This is the reason for which this method is provided as a C++ hook.

5.4 A special issue for C++

There is a special issue in C++, for virtual methods invoked on the `this` object in a constructor. For the example in Figure 6, if an object of type B is created, the constructor of A is invoked (via the constructor of B), and then the virtual method foo is invoked on `this`. The C++ reference manual specifies that in this case, the type of `this` is A and not B (as we would expect, since the A constructor was invoked on a B object). In order to preserve the correctness of RTA we must consider that class A is live (even if no explicit instantiation of this class have been found).

A possible solution to this problem is to do in constructors an escape analysis of `this`. If it is passed as an argument to another method (other than the implicit `this` pointer) or it is copied, it is considered as escaped. In this case we will consider the class defining the constructor as live.

```
class A {
  public A::A() { foo ();}
  virtual void foo ();
}

// B subclass of A
class B: public A {
  public void foo ();
}
```

Figure 6: A C++ problem

5.5 A simple variant of RTA

We have already implemented a simple variant of RTA to help us to build the infrastructure needed by the more complex algorithms described in this paper. It is a RTA algorithm that assumes that all the methods are reached.

In the first stage, all the methods are scanned and the classes instantiated are marked as live. In this stage we construct the CHG as shown in a previous subsection. In the second stage for all virtual calls we find their targets in the call graph, and build the corresponding edges.

In the case when a virtual call has a single target, it is replaced by a direct call. This made possible the further inlining of this method.

5.6 Call graph

In GCC, a call graph has been already implemented. At this point no analysis is done to try to infer the possible targets for a call done via a pointer. In the absence of this information we need to make very conservative assumptions regarding the methods reachable via such calls. This was another reason for which we implemented the simple variant of RTA from the previous section that assumes that all methods are reachable. For a complete RTA we will need to address the issue of calls via pointers first.

5.7 Inter-module analysis

RTA requires inter-module analysis. GCC already has an inter-module analysis capability for C (enabled by the option -combine), that at this stage is not functional for C++. The problems that prevent this have been detailed in [5]. The lack of this capability for C++ is a problem that needs to be solved, in order to make possible the implementation of powerful inter-procedural algorithms in GCC C++.

5.8 RTA implementation

For an efficient implementation of the RTA, we need to be able to reach from a newly instantiated class all the live virtual calls that may be affected. This can be done by maintaining a mapping from a class to the live virtual calls that are statically invoked on it. For an instantiated class, we will start an upward traversing in the CHG starting at the class instantiated. During this traversal, the mapping will provide all the virtual call sites that may be affected by this class instantiation. This is a work item.

5.9 Type propagation

The inter-procedural constant propagation optimization already implemented in GCC [4] computes information about formal parameters of a method used as arguments in a call in the method. It provides an inter-procedural propagation engine, that could be extended to also propagate type information.

For determining that an object returned by a new operator reaches an argument in a call site (found in the same method as the new) the intra-procedural constant propagation implemented in SSA may be used.

An implementation, based on the existent GCC infrastructure, of the type propagation needed by the modified RTA algorithm is another work item.

5.10 Other languages

We have seen that the implementation of RTA (or one of its variants) make use of only two language hooks, one for detecting constructors in the language and the other for finding the method invoked by a virtual call given the static and the dynamic type upon which the call is invoked.

6 Future work

We intend to complete the implementation of the modified RTA algorithm. As we have seen, there are parts in the infrastructure that are missing (an accurate call graph and the inter-module capability for C++). We will see how we can help (together with GCC community) to make them functional.

We intend to asses and to tune the performance of this algorithm on the new SPEC2006 that has much more C++ benchmarks then SPEC2000 (where eon was the single C++ benchmark).

7 Conclusions

We have presented an algorithm that integrates two simple, but effective analysis, RTA and a simple data-flow analysis (type propagation). The resulting analysis is more accurate then if these analysis are applied separately.

We have described a partial implementation of this algorithm and outlined the issues for completing its implementation.

8 Acknowledgements

I would like to thank the IBM Haifa team for advising with implementation and design issues, to Peter Bergner for reviewing this paper, as well as to Daniel Berlin and all other GCC developers that provided helpful comments.

References

[1] David F. Bacon and Peter F. Sweeney. *Fast static analysis of C++ virtual function calls.* OOPSLA'96.

[2] David F. Bacon. *Fast and Efective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California at Berkeley, 1998.

[3] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. *Interprocedural Constant Propagation.* Symp. on Comp. Construct, 1986.

[4] Razya Ladelsky and Mircea Namolaru. *Interprocedural Constant Propagation in GCC.* GCC Developer's Summit, 2005.

[5] Geoff Keating. *Inter-module analysis in GCC.* GCC Developer's Summit, 2005.

[6] CodeSourcery and others. *Itanium C++ ABI (Revision: 1.86).* `http://www.codesourcery.com/cxx-abi/`.

[7] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. *A study of devirtualization techniques for a Java Just-In-Time compiler.* OOPSLA, 2000.

[8] David Detlefs and Ole Agesen. *Inlining of Virtual Methods.* European Conference on Object-Oriented Programming, 1999.

[9] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[10] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.