*IOSR Journal of Computer Engineering (IOSRJCE)*
*ISSN: 2278-0661, ISBN: 2278-8727Volume 5, Issue 4 (Sep-Oct. 2012), PP 16-24*
*www.iosrjournals.org*

# De-virtualizing virtual Function Calls using various Type Analysis Techniques in Object-Oriented Programming Languages.

Sajad Bhat[1], Dr. Jatinder Singh[2]

***Abstract:*** *Object-oriented paradigm has become increasingly popular from past few decades due to their incredible and exciting features. The features like polymorphism, inheritance, abstraction, dynamic binding etc. made object-oriented languages widely acceptable. However at the same time the same features are responsible for degrading the performance of programs written in these languages. These features are responsible for making object-oriented programs harder to optimize than the programs written in the languages like C and FORTRAN. One of the main factors that make object-oriented languages slower is the frequent use of indirect calls to methods (virtual functions). to address the problem of how to convert virtual function calls to direct calls and to target where is the possibility of inlining, a number of techniques and algorithms have been designed and put into practical use in many of the optimizing compilers. In this work we will we put forward a study about the reducing or elimination of virtual function calls for statically typed object-oriented languages using various proposed Analysis algorithms. This study will bring to the front the best and worst about various proposed algorithms for resolution of virtual function call methods.*

## I. Introduction

The programs written in object-oriented languages are harder to optimize than the programs written in other language like C and FORTRAN. There are two main reasons for the performance penalty, First one that object-oriented programs encourages code factoring and differential programming [1], which results in smaller size of functions but frequent function calls. Second is that due to dynamic dispatch the it hard to optimize the calls in object-oriented languages; the function invoked by a call is not known until the run-time since it depends on the dynamic type of receiver. Therefore it is not possible for the compiler to directly apply the standard optimizations such as possible inlining and interprocedural analysis to these calls. One of the most powerful ideas in statically typed object-oriented languages like C++ is polymorphism which is provided by virtual functions. It is obviously a powerful concept but it carries a lot of performance overhead with it at run-time. A virtual function call is usually implemented as an indirect function call through Virtual Function Table (VFT) which is generally a table of functions. Creating and loading VFT no doubt has more run-time overhead than direct calls. Since the compiler cannot apply the inline substitution for those virtual functions that do not qualify for it, as the inlining would have reduce the calling overhead of these functions. More ever the inheritance we know is the corner stone of the object-oriented paradigm, as it provides good application design and code reused, but at the same time degrades the performance. The most visible overhead is the invocation of virtual functions and their lookup. In this work we will be considering C++ and Java for giving an overview of traditional program analysis techniques for method de-virtualization. So we may use the alternative terms between Function De-virtualization and Method De-virtualization.

The main goal of most of the optimization techniques for object-oriented languages is to make dynamic dispatches execute quickly, most desirable is to eliminate them at all [5]. A large number of people are researching around only this. As a result of a number of method have been proposed and some methods has been implemented to a great extend as well. These proposed techniques can be static, dynamic or a combination of both in nature. Static techniques rely on data flow and control flow information that can be extracted by compiler from source code of a program, and used to determine conservatory determine the concrete type of receiver of object. This technique is also called as Type Analysis.

## II. Motivation

In this paper we will use the mixed examples of both C++ and Java. Java is interpreted (JIT) Virtual Machine Languages and C++ is a compiled languages. For better understanding the concept of the virtual methods let's consider the following code segment:

```
public class test {
public test ()  { }
public Static void main (String args[]) {
test.Method1 (); // consider call site1 here
```

```
testob= new test ();
ob.Method2 (); //consider the call site 2
ob.Method3 () // consider the call site 3
ob= new childClass ( );
ob.Method3 ();// consider the call site 4
}
Public static void Method1 () {
System.out.println ("This will be a static method");
}
Public void Method2 () {
System.out.println ("This will be a normal instance method");
}
Public void Method3 () {
System.out.println ("This will be a private instance method");
}
}
Public class childClass extends test {
Public childClass () {super () ;}
Public void Mthod2 () {
System.out.println ("This will be an overridden normal method");
}
}
```

Example code 1: various types of method calls in Java.

When a program is compiled the, the compiler tires to statically bind as many method calls as it can bind. In our example program the call stie1 and call stie3 are all static calls and can be easily bound to the methods of class *test*because of the fact that static methods and private methods are not overridden. Hence the location of the code is executed as statically known.

With non-private method at call site2 and call site 4 the situation gets complicated. The declared typed of the object is statically known, the actual runtime type can be the declared type itself or any of its sub-classes. Since the runtime class can override the non-private methods implies that the actual method that needs to be called can vary from one runtime type to another. As the actual it is not known which method needs to be called at runtime, compiler cannot bind any certain method but it has to insert a lookup routine to find the correct target at runtime. These dynamically bound methods are called the virtual methods calls or sometimes called *as dynamic message sends.*

One more thing that is clear from above example1 is that the same call ob.Method2 () will result in invoking the method test#Method2 () at call site2 and childClass#Method2 () at call stie4. By closely observing we may come to know that both the call sites always call the same target procedures, and could therefore be statically bound. De-virtualization is the technique of replacing virtual method calls with static procedure calls at those call sites that always pointing the same procedure. The De-virtualization can be applied only if the complier known that no multiple targets are possible at a particular call site. For this various program analysis methods like Class Hierarchy Analysis Rapid Type Analysis etc. can be applied.

For optimizing compilers De-virtualization serves with two important roles: Firstly, replacing the dynamically bound method call with statically bound calls to avoid performing costly method lookup routine on each method invocation. Secondly, statically binding a call site to the invoked method enables the use of the methods to be inline. As it is always better to find out where the target inlining is possible, because the inlining eliminates the overhead of passing parameters and return values [2].

Call graphs are one of the traditional concepts to be used as De-virtualization techniques. By definition a call graph represents a calling relationship between the methods. A call graph generally includes a node for each reachable method. An edge is drawn from method A to method B, if call site in method A calls method B. it has been observed that a call graph with less number of edges and less number of nodes is considered the most accurate call graph. The call the call graph containing less number of edges has potential to De-virtualize more number of call sites.

## III. Cost of Virtual Function Calls

A virtual function call is an indirect call that is generally implemented through a table which is maintained for each class andthis table is called the Virtual Function table and mostly referred as VFT [3]. The VFT are sometimes called as Virtual tables, so for notational convention we will be using Vtable term to for a VFT. The VFT contains the address of the virtual function implementations, and in certain cases it will contain an offset to case receiver i.e. the calling object to the class type in which the function is implemented. Consider

below code that shows a simple single inheritance hierarchy with two classes and their VFT layout based on a standard implementation.

```
Class X {
Public virtual fun1 ();
Public virtual fun2 ();
IntA;
};
Class Y: public Y {
Public virtual fun2 ();
Int B;
};
```

Example code 2: Vtable for class Y

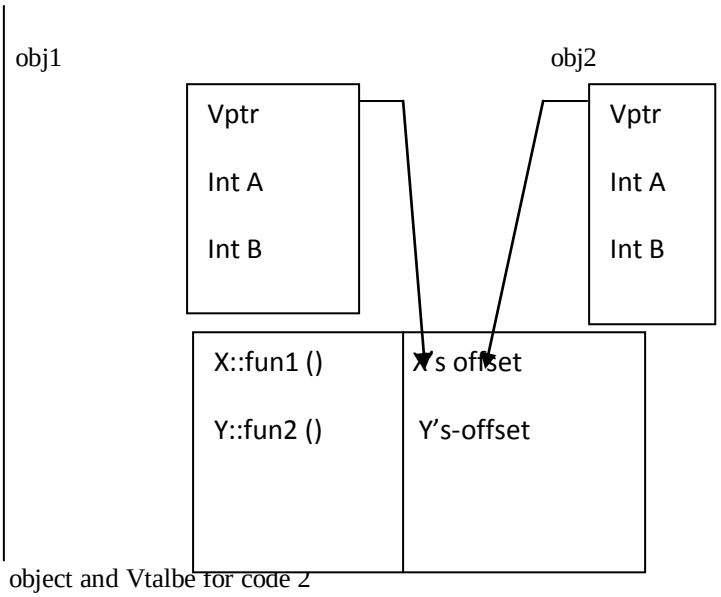


Fig 1: object and Vtalbe for code 2

The H.Srinivasan and P. Sweeney in 1996 have shown that the steps required to execute a virtual function call, which they referred as Dynamic Dispatch Sequence are as follows:

a. Loading Vtable which contains the entry for the called function through a Virtual pointer (Vptr).
b. From Vtable accessing the function entry point to branch to.
c. Making necessary changes to set the reference to the object through which the function is being called, to refer to the sub-object that contains the definition of the function that will be called. This processes this known as late caste.
d. Brach to the entry point of the function.

The steps above in most of the cases include the memory loads and simple ALU operations. These operations and the dependencies between them, a virtual function will obviously will take longer to execute than a simply statically-bound function call. This extra time of the executing a virtual call than against a normal function call is referred as *Direct Cost* of a virtual function call.

## IV. Unique Name (UN) The Earliest Work

One of the earliest works for virtual function call resolutions was published by Calder and Grunwald [6]. They used nine benchmark programs and were trying to optimize the C++ programs at link time. At link time the information at is available in object files only hence they had to restrict themselves on information available in object files only. They come to know that in some cases there is only implementation of a particular virtual function anywhere in the program. This can be detected by comparing the mangled names (the name of a function used by the linker) of the C++ functions in the object files. A functions that has a unique name or we can say that a function with really a unique signature, then an indirect call is replaced with direct call. The advantage of the Unique Name (UN) is that it does not require accessing the source code and has the capability to optimize virtual calls in library code. However when used at link time, Unique Name operates on object code, which inhibits optimizations such as inlining [7]. After the Unique Name a number of static analysis algorithms which are advanced, fast and cost effective than the Unique Name have been put into work. So in this work we will not go into the roots rather just provided a glimpse here.

## V. Treatment of Indirect MethodCalls Using Class Hierarchy Analysis (CHA)

CHA is a static analysis technique that can statically bind some virtual function calls based on the applications complete class hierarchy. CHA starts with construction of class inheritance hierarchy graph and combines this information with the declared types of the target objects at each call site [8][10]. By doing this, a set of all possible target types is estimated and this in helps in constructing the conservative call graph of the program. In case, if only one target type of some call site is yielded, then this call site can be easily resolved by replacing the dynamic method invocation with an optimal static procedure call of the only possible candidate. The optimization using CHA is based on a simple observation: if x is an instance of a class X or any subclass, then the call x->f( ) can be statically bound if none of X's subclasses overrides f. To go into more precise details let's use the following code as an example:

```
public class school {
public static void main (String args[]) {
Staff staf =getDetail();
staf.isPresent();   // call site 1
Teaching teach= fetchDetail();
tech.isPresent();            // call stie2
}
Private static Staff getDetail() {
return new Teaching();
}
Private static TeachingfetchDetail() {
return new Teaching ();
}
```

Example code 3: sample code in Java

In above sample code, the return types of the methods getDetail() and fetchDetail () are the Staff and Teaching respectively , hence are returning the instances with declared types Staff and Teaching. By looking at the method signatures, the exact runtime types returned are known. Now let's draw the Class inheritance Hierarchy for the code listed in Example code 3 as follows figure 2:

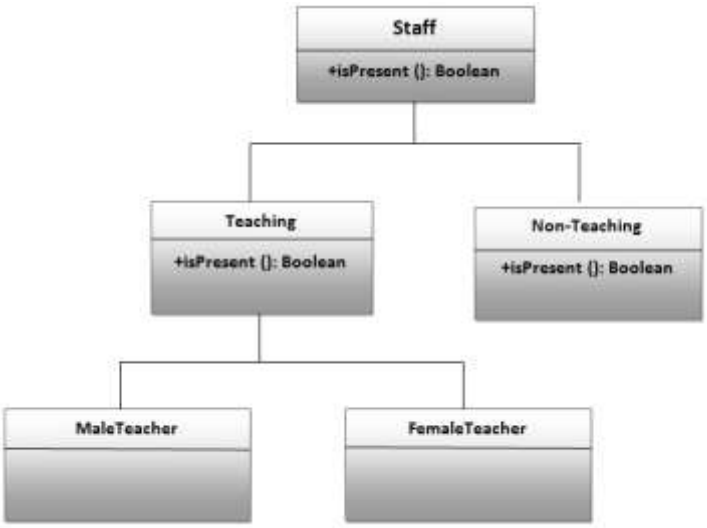


Figure 2: The Class inheritance hierarchy used by for the program in Example 3

If we start investigating the class hierarchy shown in figure 2, it can be clearly understood that, without performing any program analysis, the compiler have to implement all the calls to instance methods on any target object as Dynamic Method Invocations (Dynamic Message Sends),as any object's real type can always be a subclass of the declared type, and the called method could have been overridden in the subclass. So a reference point can be taken that the method invocations to "isPresent()" are dynamic at both call site1 and call site2 in the example3 code.

Now in the sample program of Example 3, the call site 1 has the declared type "Staff", however when CHA starts matching it with the inheritance hierarchy, it will come to know that its actual runtime type can any one of the {Staff, Teaching, Non-Teaching,MaleTeacher, FemaleTeacher}. When we observe closely, this set contains the subclasses like "Teaching" and "Non-Teaching" that override the method "isPresent ( )". This actually puts CHA into a sort of confusion, where CHA remain unable to determine that which version of the method should be invoked and hence leaves the call site unresolved.

Now consider the situation at the second call site. The call site2 in program of Example3 has a declared type Teaching. When CHA will trace down the Class hierarchy it will come that to know that the set of possible return types can be {Teaching, MaleTeacher, FemaleTeacher}, and in this set none of the subclasses overrides the method "isPresent ( )". Now the things can put into action and the dynamic method invocation can be replaced by a simple static method call like Teaching#isPresent( ). Now if we modify the Class inheritance of figure2 as shown in figure3, it would have been a little visible to know that whether to bind the call to Teaching#isPresent() or AssistantTeacher#isPresenet().

Now if we observe the code in example 3 more closely, we will come to know that the implementation of the method getDetail( ) and fetchDetail( ) is such that they can't return instances of type other than that of "Teaching". However if this would have been taken into consideration, then call site1 and call site2 both could always be resolved without any regard to the appearance of class inheritance hierarchy. But unfortunately the CHA is unable to capture the details like this, and providing a room for the improvements.

D.F. Bacon and P.F. Sweeney implemented CHA and carried the experiments on a suite of nine C++ programs [7]. Their experimental results show that the CHA comes up with good results as compared to that of UN. They found that compared to UN that resolves an average 15% virtual calls the CHA has resolved an average 51% virtual function calls.

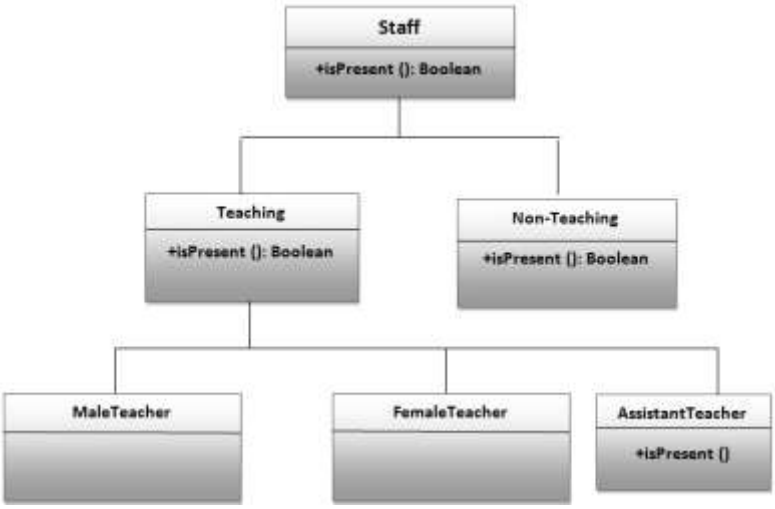


Figure 3:The modified Class inheritance hierarchy of figure 2.

## VI. Rapid Type Analysis (RTA) And Its Effectiveness:

Rapid Type Analysis is like an advanced version of CHA, it goes extra mile than CHA and reduces the set of possible target types by investigating which types are instantiated overall in a particular program [7][9][10]. In addition to that of class hierarchy, RTA also considers the whole program's class-instantiation details as well [10]. The set of instantiated types is built by traversing the call graph generated by CHA. For each call site the global set of instantiated types is intersected with the local set of possible call targets that are found by CHA, so that more number of target types under considerations arereduced. Now let's consider the application of RTA for the example code3, let's consider that the constructor Teaching () is simple and doesn't affect anything except what is expected to do, then we can say that RTA will compute a global set of instantiated typed as {Teaching}. Now the set obtained by CHA for example at call site2 is {Teaching, MaleTeacher, FemaleTeacher}, by interesting the two set we will gain the following the results:

{Teaching} ∩ {Teaching, MaleTeacher, FemaleTeacher} = {Teaching}

The result will remain unchanged even after the modification in Class Hierarchy as shown in figure3 ,that means the result for the call site 2 and call stie1 will remain same, hence in all the cases the while considering all the situations the RTA will resolve all the call sites. Compared to CHA, RTA is fast and cost effective [10], but at the same time it fails at some situations. Now let's consider the example as shown in Example code4, if we replaced the getDetetail( ) method in Example code 3 with the one as shown in Example code4, it will create a confusion for RTA. On such replacing the method implementations, the global set of instantiated types will be {MaleTeacher, FemaleTeacher}. Now the problem is that resolving call site1 is not possible now, even if it is visible that actually the only possible target type is still "Teaching".

```
private static StaffgetDetail() {
Teacher tch=new FemaleTeacher ( );
return new Teaching ( );
}
```

Example code 4:A method implementation.

There are number of researchers who presented a compare and contract about the effectiveness of RTA. Bacon and Sweeney have compared the CHA and RTA [7]. They found that compared to CHA, the RTA comes with good results. They used nine C++ benchmark programs to test the RTA. They found that when CHA resolves only 51% of total number of virtual function calls while as at the same time RTA resolves an average of 71% of total calls and run at average speed of 3300 non-blank lines per second [7]. A more detailed comparative study has been given in [10] with testes performed on nine Java programs. Here again the RTA shows good results when compared with predecessors.

## VII. Variable-Type Analysis (VTA):

As mentioned in the previous sections, CHS and RTA are simple Analysis algorithms. These algorithms come with some advantages and disadvantages as well. One of the most important positive sides of these algorithms is that they yield good results at relatively low cost. However at the same time they suffer from the problems as mentioned in section 6 and section 7. There are situations when more expansive analyses are required for getting more accurate results.

CHA is takes class hierarchy information into account for performing the analysis and to find all the implementations of the called method [10]. RTA further reduces this set by only keeping the types that are actually instantiated somewhere in the analyzed program.

To get a more fine-grained algorithm, Variable Type Analysis was proposed [9]. VTA is a flow-insensitive inter-procedural full program analysis. VTA uses the "name" of variable as its representative [9]. Forevery variable in aprogram, whether it is a global variable or a local variable in a function body VTA tries to find the set of types that could reach this variable. To do find this set of reachable types for a variable, a graph called Type Propagation Graph is constructed. In type propagation graph each node represents a variable, with description of each type of node given as follows:

- A Node of the form X.a represents the instance variable "a" of a class X.
- A Node of the fromX.m.v denotes the local variable "v" of the method "m" of class X.
- A Node X.m.this represents the reference to the current instance inside instance method m of the class X.

All the nodes are associated with a set ReachingTypes, this set ReachingTypes contains all the types that could reach the represented variable. All kinds of assignments in the graph are represented by directed edges, were the rules for the assignment are as follows:

- For the assignment like a=b, an edge is added from the node representing the variable b to the node representing the variable a.
- For any method invocation statement o.m(a,b,…..) to the method C.m(arg1,arg2,….) , an edge is added from the node representing variable a to the node C.m.arg1, etc. And also an edge is added from the node representing o to the node C.m.this.
- For the assignment like x=o.m(a,b, …), and edge is added from the node o.m.return to the node representing variable "a".
- For all the statements of type return a in a method C.m and edge is added from then node representing variable "a" to the node C.m.return.

Now let's consider the following the following example for construction of type propagation graph using the example code 5 as an example. Let's again consider the class hierarchy of figure 2 as the basic model for this construction. To start the construction of the call graph all the node we have to first identify all the nodes then look for the edges between them. We assume the initially the ReachingTypes sets of the nodes are empty. Then the type propagation graph is *initialized* by finding all the object instantiation statements e.g. new TestClass( ) within the code. For each such statement the instantiated type is added to the ReachingType set of the particular node that represents the variable that the new instance gets assigned to. At last the graph is processed starting from the sources i.e. the nodes that have no incoming edges, and the process continues for the nodes that have all their predecessors already processed. The processing of any node is done by copying all the types from its ReachingTypes set to the ReachingTypes sets of all the nodes that have an incoming edge from original node.

```
Public    class test {
Private static Staff x,y;
Public static void main (string args[]) {
 x = fun1( );
y =  fun2 ( );
Staff  s = new staff FemaleTeacher ( );
Fun3 (s);
x.Ispresent ( );//call site1
y.IsPresent ( );// call stie2
}
Private static Staff fun1(){
Return new non_Teaching ();
}
Private static Staff fun2 ( ) {
y = new Teaching ( );
Teaching t = new MaleTeacher ( );
Return t;
}
```

```
Private static void fun3 ( Staff s1) {
y = s1;
Staff g1 = y;
x = g1
}
}
```

Example code 5: an example code segment for which type propagation graph is shown in Figure 4 and figure 5.

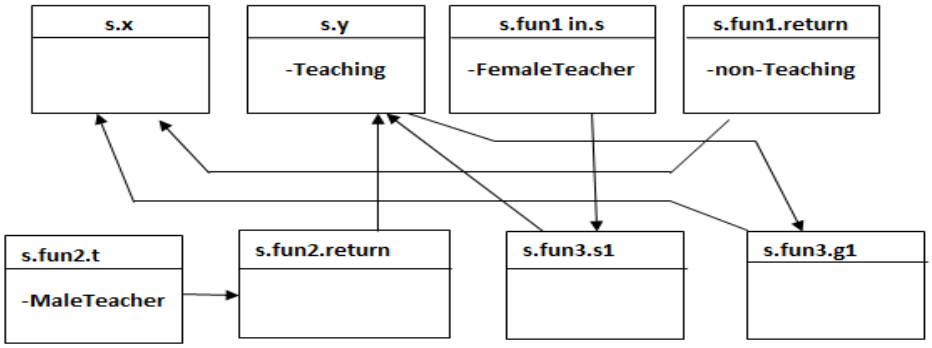


Figure 4: type propagation graph initialized but not propagated.

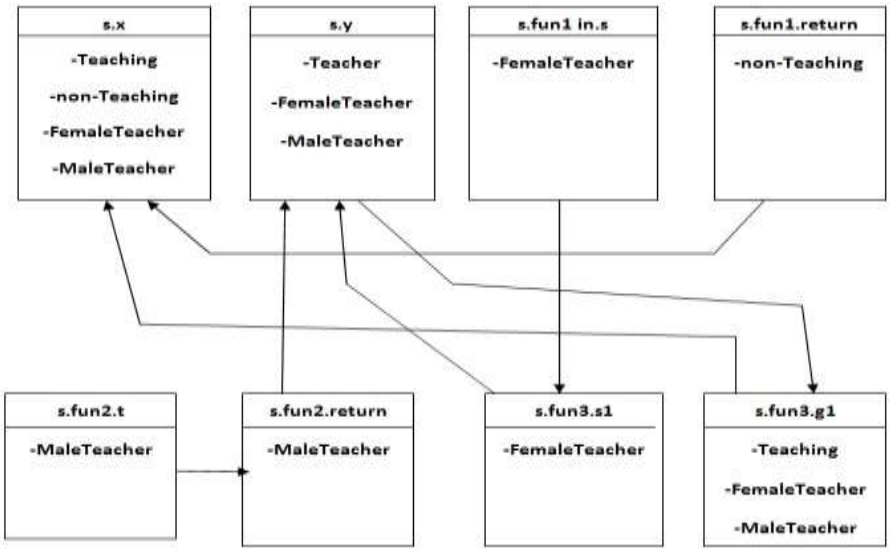


Figure 5: Type propagation graph after the propagation of types.

The processed type propagation graph for the example code 5 has been shown in figure 5. As compared to CHA and RTA the VTA gives a more accurate estimate of the sets of target types at call sites by using the ReachingTypes sets of the corresponding variables. We see from the type propagation graph shows that the call site1 cannot be de-virtualized as the instance variable x is reachable by types Teaching and non-Teaching that both define the method IsPresent ( ).but the call site2 can be easily seen to be devirtualized , as although it is reachable by three types Teaching, FemaleTeacher and MaleTeacher and all three correspond the same implementation of the method IsPresent( ),namely Teacher#IsPresent( ).

## VIII. Applications of the presented Type Analysis approaches:

In this section we will see that how really the techniques presented above are applicable in real situations. Basically applying the optimizations inside the compiler to tradeoff between the compilation speed verses the quality of the code [2]. We compilation time would have been of less concern if the compilation would have been wholly on vendor's side, as is the case in C++ applications. However the tradeoff is crucial if the compilation happens on end user's machine that too at run time, as the case is with Java programs. Breaks in the programs normal workflow due to the JIT- compiler performing some heavy optimizations have to be avoided. This implies that form environment to environment the optimizations differ. In some environment more sophisticated optimization techniques could be applied, while in another more lightweight optimizations would be more appropriate.

### 1.1. Applications with respect to Compiled Languages like C++:

In the compiled languages and linked languages like C++, since the whole program is available to the compiler for the analysis purpose, which makes the implementation process of the techniques like CHA and RTA straight-forward. Here the native code generations doesn't happen at runtime time and obviously will not affect the end user, give it a room to apply the most expensive optimization techniques during the compilation process.

### 1.2. Applications with respect to Interpreted Virtual Machine Languages like Java:

To apply the type analysis techniques presented in above sections in context of java, we will come to know that certain restrictions. This fundamental thing that we will come to know is that the JIT compilation happens during the run time, hence time too time consuming analysis can't be used. Because the usability of an interactive program greatly suffers when program's response time grows, that means a user has to sit and wait while the JIT performs it expensive analyses. At least the cost of expensive analysis has to be distributed over a longer period of program execution. The next issue the dynamic class loading capability of JVM, this feature prevents the whole program analysis [11].

#### 1.2.1. Optimizations for a just-in-time (JIT)- compiled language:

The languages like Java, which are running on a high-level Virtual Machine (Java VirtualMachine(JVM))[12] are built around a more complicated compilation model. The same applies to the other languages of this family.

The fact is that when the Java programs or the programs written in any JVM are compiled, they produce JVM bytecode. This JVM bytecode is a binary, intermediate-representation that makes JVM languages platform-independent and is executed by JVM. The semantics of the JVM bytecode make it to resemble with the native Java source code, for example it still have high-level concepts like objects, classes, methods etc. within it. This indicates that the first compilation is simple and very straight-forward and applies almost no interesting optimization techniques.

The interesting things come to the front during the execution of the bytecode that is when JVM JIT compiler converts the bytecode into native machine code. Here the compilation of any segment of code happens to be only on demand, immediately before it actually needs to be executed for the first time. There two possibilities of compilation and optimization, the compiler can perform the compilation and optimization at the moment of first compilation, or come back later to this code to perform the optimization and recompile it.It is the most common practice to first compile all the demanded code with a fast compiler, then gather profiling information when the program us running, identify the most critical methods (that is most intensively used), and later re-compile those methods with a slower optimizing compiler [2].

#### 1.2.2. Optimizing Languages with Run Time Class Loading:

With the languages that has the dynamic class loading capability, the simple analysis algorithms like CHA and RTA can't be used in a straight-forward fashion, mainly because the class hierarchy changes when the new class are introduced to it at run time. To get a better insight lets conside the example of section 5. Lets consider that the class AssistantTecher was dynamically loaded after the compiler has performed the class hierarchy analysis, built the call graph and decided that call site 2 can be statically bound to Teaching#IsPresent ().Now assume that if fetchDetail () should now return an instance of AssistantTeacher, this optimization will eventually lead to an erroneous behavior (however the fact is that this never happen). With this example it very clear that CHA has to much more careful with the dynamic class loading environments. It will be necessary to rebuilt and validate the call graph as the changed to the class hierarchy gets loaded. It is also compulsory to on-fly replace the optimized native code with all the de-virtualized method calls and inlined methods with the original virtual method invocations. As an example , the Microprocessor Research Lab Virtual Machine uses a technique called *dynamic inline patching* to revert the inlined methods that have become invalid [2]. Then compiler initially assumes that the dynamic class loading will not occur, so inlines some of the virtual function calls that have only one implementation. However if at some later stage a new class is loaded dynamically that runs some of those method inlinings invalid, the call site with inlined methods are patched with an additional *jump* instruction to fall back to virtual method invocation.

## IX. Interpretation:

The presented techniques have been evaluated by a number of researchers on the basis of various parameters. But in all the respects we will come to know that the usefulness of method like CHA or the RTA can be doubted. However many benchmarking results have shown that they are wonderfully effective in many cases because of their cost effectiveness. There are many compare and contrast analysis studies revealing the effectiveness of above methods [10][9] etc. however in [9], CHA,RTA and VTA are compared using seven benchmarks that include a Javac compiler as well. The [9] included a measurement that show big a percentage

of call sties that were always bound to one method during the actual program execution, had been resolved by those methods. CHA and RTA come up with almost equal results, de-virtualizing 50-100% of the monomorphic call sites. However VTA resolved 60-100% of monomorphic call sites and come out most effective results for all benchmarks. But the difference between the CHA and RTA was not that big, it was only ranging between 0-15%.

## X. Conclusion:

One of the main objectives of the modern compilers for object-oriented languages is to compilation process faster and cost effective. That is these compilers must perform the compilation at less performance penalty. De-virtualization is an important optimization applied in modern compilers for object-oriented languages. For the languages like C++ that are executed in a tradition fashion, the De-virtualization can be applied directly and is a somewhat straight-forward fashion, but for the virtual machine platforms like Java Virtual Machine it has be to applied carefully in a tricky way.

The point to be noted here is that all the program analysis techniques that are presented above are not the only options available here, rather there a variety of options to go with, whoever the CHA and RTA the most traditional ones are. One of the advanced variants of the VTA is Declared Type Analysis, and most result oriented variant of VTA [9]. One more sophisticated static analysis techniques is Points-to Analysis, as one their application is to resolve the virtual call sites.

As we have seen in above sections that compared to the languages C++, the languages with dynamic capability are more complex to analyze, because the class in such languages can be loaded at run time at any moment, which in changing the existing inheritance hierarchy. The traditional whole program analysis has to be modified to absorb these settings.

## References:


[1] Peter Deutsch. Reusability in the Smalltalk-80 system.Workshop On Reusability InProgramming, Newport, RI, 1983.

[2] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. ACM SIGPLAN Notices, 35(5):13_26, 2000.

[3] M. A. Ellis and B. Stroustrup. The Annotated C++ Reference Manual.Addison-Wesley 1990.

[4] H. Srinivasan and P. Sweeney.Evaluating Virtual Dispatch Mechanisms for C++.Technicalreport, IBMResearch Division, Jan 1996.

[5] J. Dean. Whole-program Optimization of Object Oriented Languages. Technical report,University of Washington,1996.

[6] Calder, B., and Grunwald, D. Reducing Indirect Function Call overhead in C++ programs. InConference Record of the Twenty-First ACM Symposium on Principles of ProgrammingLanguages (Portland, Ore., Jan. 1994), ACM Press, New York, N.Y., pp. 397{408.

[7] D .F. bacon and P.F. Sweeney, Fast and Static Analysis of C++ virtual function calls, IBMWatson research center.

[8] J. Dean, D. Grove, and C. Chambers. Optimization of Object-OrientedPrograms Using Static Class Hierarchy Analysis.Lecture Notes in Computer Science, 952(77-101):72, 1995.

[9] V. Sundaresan, L. Hendren, C. Raza_mahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C.Godin.Practical Virtual Method Call Resolution for Java.ACM SIGPLAN Notices,35(10):264_280, 2000.

[10] SajadBhat and JatinderSing , A Practical and Comparative study of call Graph Construction Algorithms, IOSR-Journal of Computer Engineering, volume 1 issue 4, May June 2012.

[11] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. ACMSIGPLAN Notices, 35(10):294_310, 2000.

[12] T. Lindholm and F. Yellin. Java Virtual Machine Speci_cation, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.