Incremental Call Graph Construction for the Eclipse IDE

University of Waterloo Technical Report No. CS-2009-07

Usman Ismail
David R. Cheriton School of Computer Science
University of Waterloo Waterloo, ON, Canada
uismail@uwaterloo.ca

1. INTRODUCTION

A call graph is defined as a set of directed edges connecting *call sites* (statements invoking method calls) to corresponding *target methods* [6]. It is a very powerful tool for program analysis and can be used to: help plan testing strategies, reduce program size (by eliminating sub-routines that are not invoked) and help programmers understand and debug large programs. Often the method invoked due to a specific call is determined at runtime based on the context in which the call is made, hence in a call graph a single call site could have multiple target methods. This is especially evident in object oriented languages where inheritance and polymorphism make method calls highly dependent on the execution context. To get the set of target methods associated with a call site we can either observe one or more executions of the program and note all methods invoked from a call site (dynamic call graph generation) or statically determine the possible methods (static call graph construction). Dynamic call graphs tend to under-estimate the number of target methods for a call site where as static call graphs tend to over-estimate this this number. A theoretically ideal call graph is the union of the dynamic call graphs over all possible executions of the program. Dynamic call graphs are not safe and generating static call graphs is computationally expensive. To ameliorate the overhead we propose an incremental call graph generation approach which will compute graphs for fragments of the program as they are being developed. It will then recursively combine fragments until a graph for the whole program is generated. The graph will be as precise as corresponding traditional algorithms and will present, upon completion, a safe call graph.

2. MOTIVATION

The theoretically ideal call graph is not computable for arbitrary programs and the dynamic call graph is not guaranteed to be safe; it may not include method calls that are possible in some executions. This is undesirable if we are using the graph to eliminate unused subroutines or compute the coverage of a testing strategy. Therefore for the purposes of this project we are concentrating on static call graphs. As stated earlier static graphs overestimate the number of target methods for each call site. The result of the static graphs is still safe but not precise. This imprecision will curtail the possible benefits of any optimizations applied on the program using the graph and will under-report coverage of a testing strategies. There are many techniques that generate static call graphs, each has a different computational overhead and each overestimates the number of target methods to a different degree. We discuss several call graph generation techniques in section 3.1; however it is intuitively obvious that techniques that are more precise also tend to be more computationally expensive as they require complex analysis. Generating safe and precise call graphs for large programs consumes considerable time and system resources. This motivates the need for an incremental call graph generation scheme that can amortize the work during the development of the program. This approach maps well to modern software engineering methodologies where an initial skeleton code is developed for a project and then method stubs are filled in by various developers over time. We can generate call graphs for small fragments of the program and then combine them as the program nears completion. The work done for generating the graph fragment will not be repeated. This makes it feasible to apply complex and expensive call graph generation techniques to get more precise call graphs.

3. RELATED WORK

3.1 Call Graph Generation Techniques

- *Reachability Analysis (RA):*

  The most basic call graph construction scheme is called Reachability Analysis (RA) and has many variations in literature such as [9]. RA only takes into account the name (or sometimes the signature) of a method, hence a method named 'm' is marked as reachable if a reachable method contains a virtual call 'e.m()' where 'e' can be any object. This procedure is recursively applied on all methods starting with the program entry point to generate the complete call graph. The RA call graph is very conservative as it ignores class hierarchy and hence links a call site to all methods throughout the program which share a name. However it is computationally inexpensive as the only

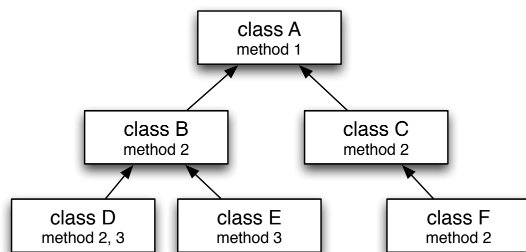

Figure 1: Sample Class Hierarchy

state required is a set of all methods in the program. This list can be pre-computed and stored in a hash table, mapping names to actual methods. At each call site RA needs to lookup the map and add all methods matching the call to the set of target methods for the call.

- *Class Hierarchy Analysis (CHA):* Note that RA generates an imprecise set of target methods because it lacks information about the class hierarchy of the object on which the method is called. CHA [5] addresses this problem, for every call 'e.m()' we filter the set of target methods to only those defined in static type[1] of object 'e' or one of its subclasses. For example, Figure 1 shows a sample hierarchy. If method 2 is called on an object of static type class B then B.2() and D.2() are both target methods. If neither the class or any of its subclasses contain a method 'm', then the target method is the one defined in the nearest ancestor of 'e' that defines 'm'. For example, if method 1 is called on an object of static type class C then only A.1() is a target method. CHA is more precise then RA but also requires more state; a representation of the class hierarchy and all methods belonging to each class. At each call site the algorithm needs to lookup the class hierarchy and use inheritance rules to determine the target methods.

- *Rapid Type Analysis (RTA):* CHA is considerably more precise then RA but it still overestimates the number of target methods as it includes methods from all subclasses. Not all of the subclasses have instantiated objects in the context of the call site, The uninstantiated subclasses cannot contain target methods. RTA [2] determines the set of classes instantiated in the context of the call site and uses this information to filter the number of possible target methods further. For example in Figure 1 if method 2 is called on class C but and there is no live object of class F in the method where the call originates then F.2() is not included in the call graph. RTA is more expensive in terms of computation and memory overhead as it needs to maintain a list of all instantiated object types for each call site in addition to the state needed by CHA . However the added complexity can lead to a substantial increase in precession especially in programs with extensive use of polymorphism.

[1]"Static Type" refers to the declared class type of pointer/refernece variable, the actual object pointed to by the variable could be any sub-class, implementing class in case of an interface.

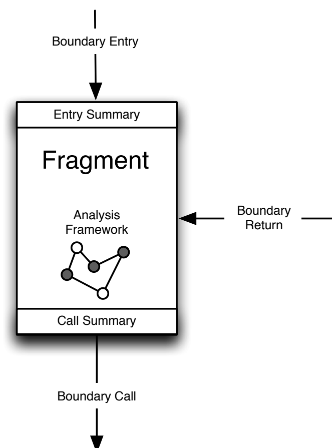

Figure 2: Program Fragment

- *Advanced Techniques:*

  There are a myriad of advanced techniques to generate more precise call graphs (XTA [10], V^a-DataReach [11]) however the contribution of this work is to implement an incremental approach call graph construction as opposed a highly precise algorithm. Consequently we use RTA to implement our approach as opposed more optimized approaches. We implement our approach in such a way as to allow for the integration of other schemes at a later date.

3.2 Call Graphs of Program Fragments

Rountev et al. [8] introduce the concept of data flow analysis on program fragments as opposed to whole program analysis. They define a program fragment as an arbitrary subset of procedures contained in a program. Each fragment has an analysis framework defining the calls within the boundaries of the fragment. Each fragment can also have one or more; incoming calls (Boundary Entries), outgoing calls (Boundary Calls) and returns from boundary calls (Boundary Returns). The boundary entries, calls and returns define the interaction of the fragment with the rest of the program[2]. Associated with every boundary entry is a summary of the rest of the program before entering the fragment. Conversely associated with every boundary call is a summary of

[2]Ignoring the side affects of global state

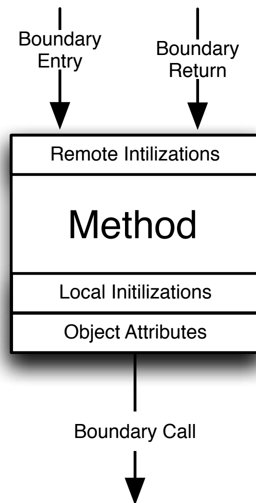

Figure 3: Program Fragment

the state of the fragment before a call is made outside the fragment. The exact contents of the summary are dependent on the type of analysis being implemented. It is important to note however Rountev et al. envisage a pre-computation stage applied to the whole program prior to fragmented analysis. Furthermore computation of program call graph is a necessary part of this pre-computation stage. Without a call graph it is not possible to compute the boundary entry and the associated summary. In our approach, section 4 we address the issue of incomplete information.

4. OUR APPROACH

We implement an incremental call graph construction mechanism integrated into an IDE such as Eclipse [1]. To the best of our knowledge the most comprehensive call graph generation tool for eclipse is *Call Hierarcy* [7]. Call Hierarchy generates a call graph based solely on the static types and ignores late binding of objects and methods. In addition it generates a call graph only on user demand and does not reuse results of one invocation to optimize the next.

To implement incremental call graph construction we borrow the concept of a *Program Fragment* from Rountev et al. [8] see section 3.2. We however note that the concept of a fragment does not map to incremental call graph generation. There is no set of methods that we can assume are consistent and hence there is no clear demarcation of which methods should be included within the boundary of the fragment. Hence we assume each method is in its own fragment (See figure 3), each call into a method is a boundary call and each return a boundary return.

We use the RTA algorithm to generate call graphs but modify and extend it to handle incomplete programs and to reuse computation from earlier invocations. The non-incremental RTA is insufficient for a number reasons, it needs a set of reachable methods which is initialized to program entry points. The program entry points may or may not exist when the program is incomplete hence we cannot rely on the reachability set. Furthermore RTA assumes a traversal of the graph such that all call sites are processed before their target methods and therefore information about the live types of parameters passed to methods is known. With incomplete programs the call site may not yet be defined and hence there is no information available about the possible live types passed into a method as parameters. Lastly incomplete programs can expect changes to the class hierarchy, method definitions, and even the source code of methods at anytime and the current RTA algorithm is unable to handle such changes.

We propose an approach called Incremental Rapid Type Analysis (IRTA) that extends RTA to incrementally incorporate changes. After its first invocation IRTA will build a complete call graph of all method calls reachable from the entry points (if they are defined) and also all method calls reachable from an arbitrary method selected by the user. We envisage that users will run an iteration of the incremental algorithm on a method immediately after they make changes to it. We can automate this process by hooking a callback to the relevant event. Each iteration will then output a correct and consistent call graph although the graph may have several distinct connected components. Furthermore some of the components may not be reachable because they do not contain a program entry point. As more methods and method calls are added some of these components will merge into larger components. At the end of any iteration the user can isolate the reachable components and these collectively form the program call graph that would have been output by static RTA. A high-level pseudo code for our approach is presented in algorithm 1 and details are provided in subsequent sections.

Algorithm 1 Pseudo code for IRTA

```
 1: Build / Update Class Hierarchy
 2: Find Program Entry Points (Add to work list)
 3: Find Last Edited Method (Add to work list)
 4: Pull method from head of list
 5: Parse method for local declarations
 6: Parse method for method calls
 7: Find target methods corresponding to call site
 8: Add Links between methods for calls and returns
 9: Propagate live types across calls and returns
10: Find methods that need to be reprocessed and place back
    in work list
11: Output all connected components
```

4.1 Class Hierarchy

The first step in the algorithm is to analyze the class hierarchy and build a representation in memory for use by later

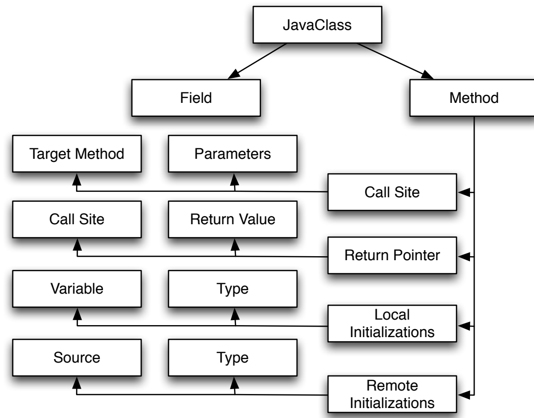

Figure 4: System Overview

stages of the algorithm. In this stage we identify, for each class, its fields, methods (as shown in Figure 4) and store them in memory. We need to identify which classes have been added to or removed from the hierarchy since the previous invocation. We therefore traverse all classes in the program and for each check if it exists in the representation. We also check if any class listed in representation is no longer present in the program. In addition a class defined previously may have fields or method added to it or removed from it. To account for these we compute a hash over the signatures of fields and methods and store it with the class. We recompute the hash every time IRTA is invoked and if the stored hash differs from the computed hash we know that we need to re-parse the class to extract field and method definitions. Note that if a user changes the signature of a method it is modeled as a deletion of the old method and addition of a new one. While parsing method signatures we identify the program entry points and add the corresponding methods to the work list. Furthermore we identify the last edited method and add that to the work list as well. These changes are updated in the representation before we start processing source code within methods to generate call graph.

4.1.1 Adding and Removing Methods

In addition to an updated class hierarchy the previous phase of the algorithm outputs a set of methods that are added to the code base (the addition of a class translates to addition of all its methods). A new method affects the call graph output from the prior iteration in several distinct ways: (1) it contains new call sites that must be parsed, (2) it contains new initialized types that it propagates to other methods and (3) it may be a target method for an existing call site. 1 and 2 are handled when the method is processed as described in Section 4.2. The third effect however requires the reprocessing of methods that are already processed. For example call site ClassA.Method1 maps to target method ClassB.Method2 and ClassB is a sub-class of ClassC. If we now add a definition of Method2 to ClassC then any call sites mapping to ClassB.Method2 may now also map to ClassC.Method2. To incorporate these new mappings we reprocess all call sites that map to a method that *matches* a newly added method. We define two methods as matching if the class one method is defined in, inherits from the class defining the other. In addition, both methods must have the same name and corresponding parameters must be *compatible*. Parameters are said to be compatible if the class type of either inherits from the other. Removing method calls is much simpler because we already have references of call sites and they reference target methods for the purposes of propagating return values (see Section 4.2.2). We remove all calls emanating at referencing the deleted target method by following the these return pointers. The details of removing a method call are given in Section 4.2.1.

4.2 Method Parsing

The algorithm has a worker thread which pulls a method from the head of the work list and parses its source to extract all variable declarations, object instantiations and call sites. The variable declarations and the object instantiations are used to determine the set of live types for RTA. We also maintain a set of key-value pairs mapping variable names to their static types. If methods are overloaded to accept different parameters, the set is useful for determining which call site maps to which target methods. We also maintain a key-value pair set of live types storing all initialized types, local and remote. We initially have no information about remote initializations (Types that are received from non-local sources, e.g. method parameters and return values). Therefore we initially assume there are no remote initialized types. Method parameters and return values from method calls are initially assumed to return only their statically declared types. As the concrete types of interfaces may not be statically determinable, in fact the concrete type may not even be defined we leave support for interface declarations to future work.

Once methods are processed they propagate their initializations, as described in section 4.2.2 and we can update the call graph accordingly. Note that remote initializations are stored as a key value mapping from source of the initialization to the type. Therefore a method may get the same type from multiple remote sources. This is important because if one or more of the sources are removed we need to know if that remote type is still propagated from alternate ones. Lastly the set of live types also contains the fields defined in the type containing the method and public fields of other classes. The current algorithm ignores the existence of public fields of non-local classes and assumes that fields of the containing class only take their statically defined values. We make these simplifying assumptions because it is very difficult to determine which methods are editing publicly available fields using getters and setters or aliased pointers. Without identifying the methods that edit a field we cannot deter-

mine the set of live types that a field can take. Perhaps we can integrate a points to analysis [3] into our approach to identify the possible initialized types of fields.

Once we have the live types we parse the method source code to extract call sites emanating from the method. Each call site is of the form $\alpha.\beta(\gamma_1,\gamma_2,...\gamma_n)$. Where α could be a variable, field, method call or expression; β is a method name. Each of the parameters $\gamma_1,\gamma_2,...\gamma_n$ could be a variable, field, method call, expression or a constant. We need to determine the static type of α; if it is a variable or field, we can lookup the type from the live types or fields key-value sets, if it is a method call we can retrieve the return type of the method and if it is an expression we recursively evaluate it until we can resolve its static type. Similarly we resolve the static type for each parameter γ. All methods which define a method named β, accept parameters $\gamma_1,\gamma_2,...\gamma_n$ and are defined in a class from the hierarchy of α are candidates to be target methods. For example in Java Object.method(String p) will map to all methods named "method" accepting a "String" or one of its subtypes as a parameter, defined in classes which are in the hierarchy of Object. As per RTA we filter the matches to only those that are live by consulting the local initializations, remote initializations and the fields key-value sets. For each of the remaining candidates we create a new method call instance and add a link from the call site to the target method. We also add a back-pointer from the target method to the call site so that the target method can return information about the live types of its return value when they are available.

4.2.1 *Adding and Removing Method Calls*

After extracting the call sites and live types and determining the target methods that link to a call site we need to add the calls. This is a two step process: first, we need to add the forward link, i.e. the method call and then we add a reverse pointer to the target method pointing back to the call site. We propagate information about known live types along the forward call as described in Section 4.2.2. We also place the target method on the work list because it may need to be reprocessed to account for remote types we are contributing. We also mark the target method as *Needs Propagation*. This ensures that when the method is pulled off the work list it will propagate information about the live types it can return through the return value. For example if the return type of the method is "List" it may contribute back initializations of "ArrayList" or "LinkedList".

To remove a method call we need to first remove all initializations that the call site contributes to target methods. This is possible because each remote initialization has information about its source. We therefore ask the target method to remove all initializations sourced at the call site. This may or may not require the target method to be reprocessed. After this we remove the reverse pointer at the target method linking it to the call site. We also remove all remote initializations at the call site which were contributed by the target method as possible return values. This may or may not require the method containing the call site to be reprocessed. We remove the forward link and add the method containing the call site and the target method both to the work list.

4.2.2 *Live Type Propagation*

After all calls, return points and initializations are resolved we need to propagate live type information to remote methods. For each of the calls we extract the static types of the parameters. The only live types that can be propagated to the target method are the ones that are in the type hierarchy of one or more of the parameters. Hence we filter all live types to include only those that are part of the type hierarchy of one of the parameters. Similarly for each call we expect a return value, therefore whenever a method is processed we traverse the list of back pointers and propagates live types back. Again we filter the live types to those that are in the hierarchy of its return type of the method.

If the propagation of types results in a change to the list of remote initializations of the target method (or call site in the case of back pointers) then the method is marked as dirty. The method has already been placed on the work list when the target method was processed hence it will be processed in turn. Note that the propagation of types to a method may not require the method to be reprocessed. For example if the target method of a call receives a "String" parameter from call site A and a new call site (Site B) is added which also contributes a string parameter then no new types have been learned. In such a scenario the target method or call site does not need to be reprocessed. Hence when the method is pulled from work list it will not be reprocessed. By reprocessing only when required we are able to contain the effect of any change in source code to only those methods that need to be changed. To keep track of which call site or target method contributed a remote initializations, all such initializations are stored as key value pairs mapping source of the initialization to contributed type. Another subtle point to note is that call site B expects the target method to propagate back information about its live return types. If the method was reprocessed this would take place automatically. If however the target method is not reprocessed it still needs to propagate its live types along back pointers. This is why the call site processing marks all target methods as *Needs Propagation*, hence regardless of whether a target method is processed or not it returns its live type information to call site.

5. IMPLEMENTATION DETAILS

5.1 Overview

We implement our approach as a plugin for Eclipse SDK Version: 3.3.2 (Europa) called ICallGraph which adds a View to the Eclipse workbench; see Figure 5. The user initializes the generation of call graphs by selecting the corresponding icon and the plugin runs in background generating the call graph and adding calls to the display as they are dis-

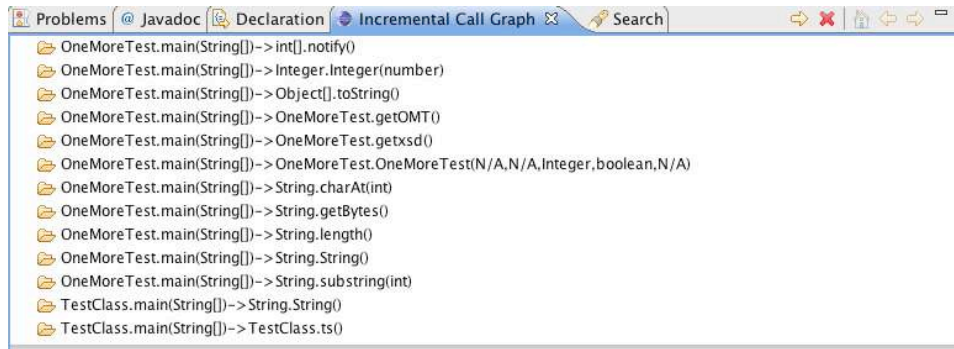

Figure 5: Screen Shot of Sample Call Graph

covered. The plugin runs as a background process hence the user is free to make changes while the plugin is running. After making changes to a method the user must place her cursor within the method and reinvoke the plugin. The plugin will add the current method to the work list and when it is processed the changes will be incorporated into the call graph. If at any time the work list becomes empty then the plugin's processing (worker) thread goes into sleep mode. If any changes are made to the source code, the affected methods are added to the work list and the worker thread is restarted. We will automate the processes of adding modified methods to the work list but for our prototype we rely on the user invoking the call to add each edited method to the work list.

5.2 Limitations

There are several limitations of the in the current implementation, some are inherent short-commings of the approach where as others are problems in the implementation that will be addressed in future iterations of prototyping.

- The first problem with the current implementation is that it is unable to handle Java source level 5. This is due to the support for generics and and auto-boxing in Java five which makes it very difficult to resolve types. Variables and methods defined using generics have no static type and hence our current approach of assuming only static types are live and then incrementally adding types cannot work with generics. Auto boxing allows function calls on primitive types, for example 5.3.toString(), which which makes it difficult to analyze call sites.

- Another shortcoming of our implementation is that it is unable to propagate remote types contained in lists. For example if a method call accepts a type parameters of type ArrayList then the call site only propagates live types in the inheritance hierarchy of ArrayList. If the source code was such that it places objects of other types in the list before passing it to the target method then these types need to be added to the remote initializations list of the target method. We currently have no way of handling such cases and propagating the types. Propagating all live types will eliminate this problem but will cause many unnecessary types to be propagated. We plan to update the parser to keep track of all live types inserted into dynamic data structures and use this information to propagate types.

- Our current implementation uses the Abstract Syntax Tree (AST) Parser support built into eclipse to read source files for analysis but currently our implementation only parses java files. We have no support for parsing class files or jar files. Hence we currently are not able to correctly handle calls to standard library functions. The current implementation lists static calls to standard library functions but does not de-virtualize the call to all types. This is because we do not know which types in the hierarchy define an overriding method corresponding to the call. We hope to resolve this in future releases by adding parsers to operate on class files and jar files.

- Our current approach to storing class hierarchy requires classes to be named hence we are currently unable to handle anonymous inner classes.

- There are several limitations of our current parser that we were unable to eliminate due to shortage of time. Currently our parser requires all control constructs (Loops and If statements) to be in block syntax. Inline statements for such constructs are not resolved and any method calls in inline statements will be missed. For similar reasons we are unable to processes Switch-Case statements properly. The switch statement does not use block syntax because of fall through, i.e. if there is no break at the end of one case the subsequent case is also invoked. We hope to correct these issues in the parser shortly.

- Lastly we note that there is no integrated graph rendering tool for the Eclipse IDE and we are forced to use tree or list views. We currently output the graph

as a list of edges of the form call site → target method. This visualization method is not intuitive and will build or integrate a graph visualization tool into eclipse for display of the call graph in subsequent version of our plugin.

6. EVALUATION

Our approach is designed to generate exactly the same graph as would be generated by running static RTA and therefore is guaranteed to be safe. We verify this by running our algorithm on a completed program to generate the complete Call Graph in a single iteration. We then remove several classes and methods from the code and generate the the call graph from a clean state. Once we have a call graph we re-insert the removed portions one-by-one and reinvoke the algorithm for each insertion. The call graph generated in this fashion matches the original call graph.

We evaluate our approach in terms of the time taken to recompute the call graph after changes to the source code. With the static algorithm we would need to recompute graph every time the source changed but using the incremental approach we wish to recompute the minimum required graph subset and complete in the minimum required time. For our testing we run our algorithm on the Antlr benchmark which is part of the Dacapo Benchmark Suite [4] on a 2.2 GHz dual core intel machine with 1 GB of RAM. This is a small benchmark with approximately 200 classes and the static algorithm had to parse approximately 2300 methods. The total time taken to generate the call graph is approximately 62 seconds as shown in Figure 6. The first 200 events correspond to the time taken to generate class hierarchy information for the classes and the rest shows the time taken to parse methods. The plot shown is the average across ten runs of the algorithm. We draw the 90% confidence intervals but they are too tight to be seen.

To study the effectiveness of the incremental approach we first generate the static RTA call graph and then make several changes and see the time taken to recompute the call graph. The first change we study is that of a new class being added to the hierarchy. A new class requires all its methods to be parsed and their calls integrated into graph. However the more time consuming process is recomputing all call sites that may now target the new methods. If there are call sites that target methods which are overridden by the new class then those call sites have to be reprocessed. The most difficult class to integrate into the graph is the one with a large number of methods that override methods of other classes. To simulate such a troublesome class being added to the program we select the three largest classes in the benchmark and for each we add a sub-class which overrides all methods in the original class. In addition we also note that if we override classes with deep hierarchies we will disturb a larger number of methods. Hence we also select two classes with a large number of subclasses apply. The time taken to process each of these additions is shown in Figure 7. The results are promising as they reduce the cost of computing the graph to just 10% of original cost. Figure 8 shows the cost of adding the same classes in terms of number of methods that have to be reprocessed. The results show that about 5% of the methods have to be reprocessed to account for a new class being added to the hierarchy.

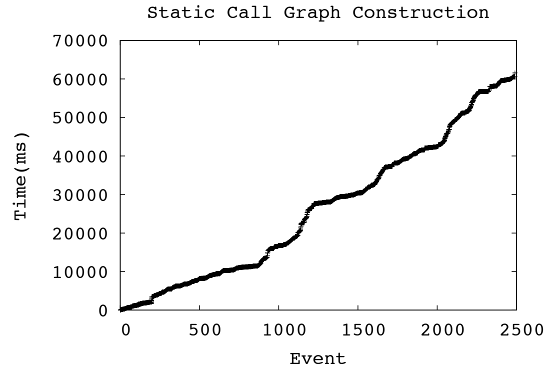

Figure 6: Time taken by Static RTA

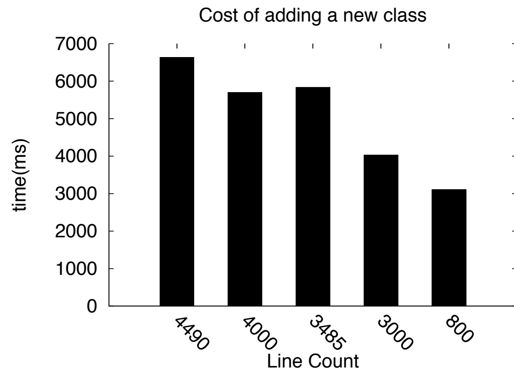

Figure 7: Time taken to add a new class

We also study the cost of removing calls from the source in terms of both the time and the number of methods that have to be reprocessed. Removing a method not only requires all calls emanating from the method to be deleted but also requires all remote types contributed by that method to be deleted. This implies that all methods that receive remote types from the removed method have to be recomputed. This in turn may remove some calls and contributed remote types leading to the changes percolating in the system. To simulate such a scenario we randomly select 5 methods from the benchmark and remove all code within the methods. We do not delete that method as it could lead to compilation errors which can render analysis invalid. The time taken to processes each of these removals is shown in Figure 9. The

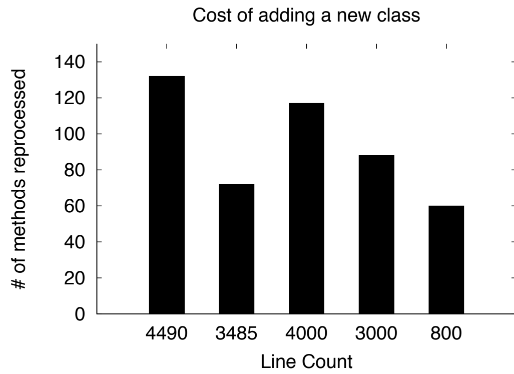

Figure 8: Number of methods parsed to add new class

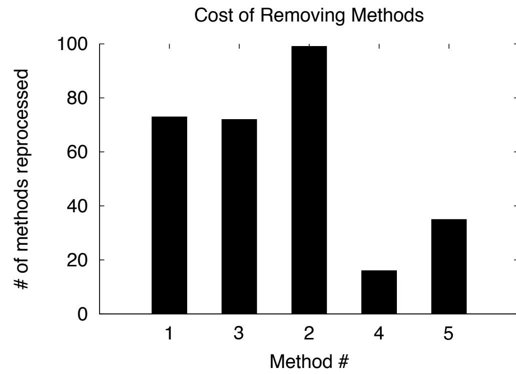

Figure 10: Number of methods reprocessed to remove a method

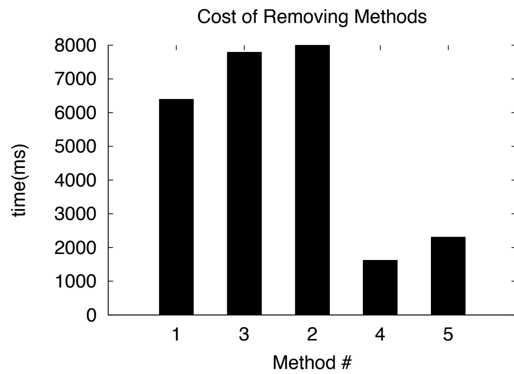

Figure 9: Time taken to remove a method

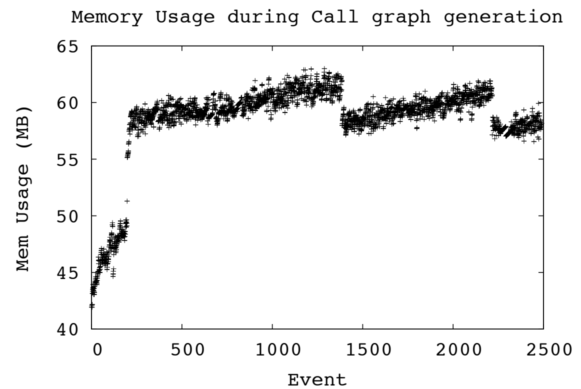

Figure 11: Memory Usage in generating a call graph

results show that our approach reduces the cost of computing the graph to just 12% of original cost. Figure 10 shows the cost of removing a method in terms of number of methods that have to be reprocessed, about 4% of the cost of recomputing the whole graph.

The results are somewhat promising as we reduce the time taken to generate a call graph by as much as 90% and reduced the number of methods that have to be reprocessed by as much as 95% or more. However we note that the cost of updating the graph is not linear in terms of the percentage of the source code changed. We think this is because multiple changes map to the same set of reprocessed methods hence incur the reprocessing cost only once. We need further investigation of the number of absolute minimum necessary re-computations and spurious re-computations based on ordering of methods on work list.

Lastly we want to find out the overhead in terms of system resources of running our algorithm in parallel with the IDE. In particular we want to know the overhead of updating the graph in response to a change in the code base. If the algorithm consumes large amounts of memory for the user to use the non-incremental version of call graph generation algorithm. We compute the memory utilization in generating a call graph using our approach and the results are shown in Figure 11. As can be seen the growth in memory utilization is limited to the first two hundred events (building class hierarchy information) and the growth in memory is linear in terms of number of classes. Eclipse IDE uses approximately 42MB upon initialization and loading of the project and our algorithm uses a maximum of 62 MB. This comes to approximately 100KB for each of the 200 classes. Modern computers commonly have main memories of one or more gigabytes and hence this will not be overly burdensome. We must note that our readings were taken while graphical output was disabled, adding several thousand entries to a view in eclipse will take much more memory. We hence propose that the graph be stored in memory but the display be updated in an as needed basis. For example the the users be allowed to select a method and view all calls emanating too or terminating at that method. They can then browse the call graph as needed.

A related question is; with todays powerful computers is

an incremental approach really needed? As we have mentioned that the delay of the benchmark under consideration was approximately sixty seconds and can be much higher for larger benchmarks. This delay can be brought down by the use of more optimized data structures but will still be significant baring advances in processing power. It is an open question whether processing power will increase faster then program complexity or not. We do however note that by allowing for incremental call graphs we can allow for more complex graph generation algorithms in the same time frames as current algorithms with static generation.

7. FUTURE WORK

- The first avenue we can explore in this space is incremental parsing of methods. Currently we only distinguish between the cases where re-parsing is required and not required. However we do not use information from when the method was previously parsed. As much of our processing cost is incurred when parsing methods we can significantly improve running times if we can use incremental parsing.
- Currently we use only information from the source code to detect which parts of the code have changed. If we integrated IDE events that monitor classes and methods being added and deleted we would not need to parse code and detect changes.
- As mentioned earlier currently we processes methods in the order they were placed on the work list but we note that often many changes affect the same methods. Hence the same methods are placed on the list again and again. If we can reorder methods on the work list such that each method has to be processed the minimum number of times we can greatly reduce the cost of generating the call graph.

8. SUMMARY AND CONCULSION

As we have discussed call graphs are a very useful tool for programmers and compilers but generating precise call graphs for large softwares is very expensive. We can however apply a standard call graph generation technique incrementally to small program fragments and combine the results from the fragments to get the entire call graph. The incremental steps can be applied in the background while the program is being written hence reducing the apparent overhead of generating a call graph. Our approach is be integrated into the an IDE and thus will not require the programmer to be actively involved in the process of generating the graph other then initializing the. As it is the first implementation of a call graph generation algorithm for eclipse which provides analysis beyond static type we feel it will be a useful tool for developers and its incremental and extensible nature will allow for the integration of more complex analysis not available to developers today.

9. REFERENCES

[1] Eclipse - an open development platform, http://www.eclipse.org/.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. pages 324–341.

[3] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114, New York, NY, USA, 2003. ACM.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[6] Ondřej Lhoták. Comparing call graphs. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, New York, NY, USA, 2007. ACM Press.

[7] Jesper Kamstrup Linnet. Call hierarchy plugin for eclipse, http://eclipse-tools.sourceforge.net/call-hierarchy/.

[8] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 235–252, 1999.

[9] Amitabh Srivastava. Unreachable procedures in object-oriented programming. *ACM Lett. Program. Lang. Syst.*, 1(4):355–364, 1992.

[10] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, 2000.

[11] Weilei Zhang and Barbara G. Ryder. Automatic construction of accurate application call graph with library call abstraction for java: Research articles. *J. Softw. Maint. Evol.*, 19(4):231–252, 2007.