McGill University
School of Computer Science
Sable Research Group

Notes on VTA implementation in Soot

Sable Technical Note No. 2000-1

Felix Kwok

September 19, 2000

1 Introduction

The current implementation of Variable Type Analysis (VTA) in the Soot framework is based on the VTA algorithm presented in Vijay Sundaresan's Master's thesis, with minor changes to improve precision[Sun99]. The goal of these notes is to explain these modifications, as well as to compare the results of the current implementation against those presented in the thesis. A discussion on the limitations of the current implementation is also included.

2 Algorithm

The current algorithm is essentially the same as the one presented in the thesis. However, three significant changes have been made:

2.1 Array variables

In Vijay's thesis, whenever there is assignment between (a) two arrays, (b) one array and one variable of type `java.lang.Object` or (c) two variables of type `java.lang.Object`, edges are added in both directions. This is meant to encode the aliasing relationship between the two variables in case they are arrays. However, this treatment gives rise to many spurious edges between two variables of type `java.lang.Object`, since many such variables are never assigned array types, and will never be aliased with other variables.

In the current implementation, double edges are not added across the board. Instead, edges are added in two phases. In the first phase, only edges are only added in one direction (from r.h.s. to l.h.s.). Whenever a local is assigned a `newarray` or `newmultiarray` expression, the corresponding node is marked as an array node. In the second phase, we do a depth-first traversal starting at these marked array nodes. If there is an assignment from an array node to a node of array type or of type `java.lang.Object`, we mark that node as well, and add a back-edge to its parent. This is valid because only the nodes that are reachable from an array node can possibly be aliased with their parents. Hence, this algorithm avoids many spurious edges between variables which have nothing to do with arrays.

2.2 Exception edges

This modification is mainly to ensure correctness of the algorithm. Each method now contains a `throw` node, which will contain the types of exception that can be thrown by the method. These include (a) the exception explicitly thrown via a `throw` statement, and (b) all subtypes of `Error` and `RuntimeException`. The latter is included because these exceptions are unchecked by the virtual machine, and can be thrown implicitly by any instruction (e.g. an array reference can cause an `ArrayIndexOutOfBoundException` to be thrown). To complete the throw/catch pair, edges are added from the `throw` nodes to the `caughtExceptionRef` nodes in the exception handler. The purpose of exception edges is not so much to increase the precision of the analysis, but to ensure that all variables (even locals of type `java.lang.Exception`) have correct reaching types.

2.3 The native method `newInstance`

The Closed Application Assumption for VTA (as defined in Vijay's thesis) assumes that all object instantiations must be made by a `new` statement. This assumption is violated whenever the native method `newInstance` is invoked. The complete signature of `newInstance` is:

```
<java.lang.Class:  java.lang.Object newInstance()>
```

This instance method has a receiver of type `java.lang.Class`, and it creates a new object with the same type as the one indicated by the receiver object. Hence, the problem is that one cannot tell statically what type is instantiated by the method. A related problem is that even the class resolver in Soot cannot tell which class needs to be resolved!

One obvious (and naïve) approach is to approximate very conservatively, by including all types in the reaching types of the return node. Fortunately, there is a better way. Since the newly created object has declared type `java.lang.Object`, the program must first cast it to a more sensible type before it can access any type-specific fields/methods. Hence, a `newInstance` is closely followed by a cast instruction most of the time. Our algorithm exploits this fact to obtain a closer approximation.

After building the VTA graph, our algorithm starts by finding all the nodes that are successors of the return node of `newInstance`. Then, for each of these nodes (call it v), we do a breadth-first search starting at v to find the first edge that corresponds to a cast instruction. (There may be several "first" edges, since the control flow may split after the instantiation.) If we encounter a `virtualinvoke` or `specialinvoke` instruction before the first cast edge, we approximate by including all types in the reaching types of v. Otherwise, we approximate by including all subtypes of the types indicated by the cast instructions. Although this approximation is technically unjustified (since we have no reason to believe that the cast must be valid), this is acceptable in practice. This is because if the approximation is invalid, this must be because the cast instruction is about to fail. In that case, the reaching type of the variable is unimportant because control flow will then directed to the exception handler, and the invocations affected by the approximation will be bypassed.

The increased precision in the analysis of `newInstance` is important for two reasons. First, if the newly created object is subsequently stored in a data structure (e.g. a `Vector` or a `Hashtable`), the current algorithm is unlikely to affect variables in unrelated methods that happen to use the same data structures. In the naïve approach, on the other hand, will cause those variables to lose precision because there is a path from the newInstance to them via the data structure. The second reason involves the reachability of constructors. Since `newInstance` implicitly calls the constructor of the class it instantiates, the naïve approach must assume that all constructors are reachable, while most of them are in fact unreachable. We have observed that the current algorithm finds more unreachable methods than the naïve approach for this reason.

At this point, one might wonder about the additional cost required to analyze `newInstance`. In terms of time complexity, the algorithm theoretically may have to traverse the whole graph before finding a cast edge. However, we have observed that in practice the cast edge is discovered very quickly (the cast is very often executed immediately after the creation of the object). Hence this algorithm has negligible cost in time. However, it does cause VTA to use slightly more memory, since cast edges must be explicitly remembered when building the VTA graph.

A final remark must be made concerning dynamic classes in general. Since Soot is unable to decide which class to resolve if the class is dynamically loaded, the results of VTA is correct only if the user explicitly specifies which classes are dynamic. Slight changes have been made to the Soot command line for this purpose. See Section 5.2 for details.

3 Experimental Results

This section presents the results of VTA when applied to a set of benchmarks. The choice of benchmarks is made to facilitate comparison with the data in Vijay's thesis. Unfortunately, the two sets of numbers are not exactly comparable, since the benchmarks are slightly different (small discrepancies are found between the conservative call graphs, which should be the same if the same benchmarks were used). However, there are enough similarities between the data to conclude that both versions of the analysis yield similar results.

Name	Whole Application							Benchmark Only							
		Call Sites			Edges				Call Sites			Edges			
	$\lvert N\rvert$	mono	pot. poly.	total	mono.	pot. poly.	total	$\lvert N\rvert$	mono.	pot. poly.	total	mono.	pot. poly.	total
raytrace	2373	7638	387	8025	7607	2837	10444	217	2053	10	2063	2053	42	2095
	1729	6582	377	6956	6576	2591	9167	207	2037	12	2049	2037	46	2083
compress	2227	6509	381	6890	6478	2806	9284	83	940	6	946	940	30	970
	1583	5450	369	5819	5444	2556	8000	76	927	6	933	927	30	957
jack	2501	8246	795	9041	8215	3931	12146	345	2686	396	3082	2686	1046	3732
	1857	7191	779	7970	7185	3619	10804	337	2672	396	3068	2672	992	3664
javac	3863	13302	1931	15233	13254	16237	29491	1386	7601	959	8560	7600	11096	18696
	2821	10570	1276	11846	10564	13707	24271	1188	5933	848	6781	5933	10306	16239
sablecc	4944	15548	1471	17019	15509	25820	41329	1941	5917	879	6796	5916	20742	26658
	3737	11151	1332	12483	11140	24553	35693	1955	5920	889	6809	5920	20736	31690
soot	3806	13767	1987	15754	13762	22225	35987	2505	10494	1804	12298	10491	19487	29978
	2828	11653	1738	13391	11653	25331	36984	2001	9070	1545	10615	9070	22620	31690
pizza	3147	14450	778	15228	14448	6159	20607	1789	11143	575	11718	11143	4101	15244
	2660	13729	799	14528	13729	6024	19753	1756	11115	577	11692	11115	4069	15184

Table 1: Conservative Call Graph Characteristics

Table 1 gives a summary of the conservative graph characteristics based on CHA. For each benchmark, the first row contains data from the current implementation, whereas the second row contains those from Vijay's thesis (included here for comparison). The differences between the two sets of numbers can be due to the following reasons.

1. The java library code used for the two benchmarks are different. This is likely to be the reason why there is a significant discrepancy in the number of methods in the whole application, even though a discrepancy of similar magnitude is not noticed for the benchmark-only methods.

2. The benchmarks are packaged differently. This is likely to be the case for Spec benchmarks such as `raytrace`, `compress` and `jack`, for which the discrepancy in benchmark-only methods is typically less than 10 methods.

3. The versions of the benchmarks may be different. This can account for the 25% increase in the number of benchmarks methods for `soot`. An increase of this magnitude is unlikely to be due to packaging differences.

Table 2 summarizes the improvement of the call graph over the conservative call graph after applying the current implementation of VTA. For each benchmark, the row labelled "VTA" gives the improvement after applying VTA once, whereas "VTA+VTA" indicates the improvement after applying VTA twice (i.e. use VTA to get the first pruned graph and use it as the input to another pass of VTA). Table 3 gives a summary of the pruned graph characteristics (benchmark only).

One is invited to compare the improvement results with those obtained by Vijay in his thesis. We believe that since there are large discrepancies between the conservative call graph characteristics for whole applications, the percentages are of more relevance than the actual numbers. However, since the benchmark-only statistics are similar for several benchmarks, both the percentages and the actual numbers are important; thus, we also show some statistics for the pruned graphs, restricted to the benchmark only. One immediately observes that the restricted call graphs are almost exact matches for the first three benchmarks, meaning that the two versions of the analysis yield very similar results.

		Whole Application			Benchmark Only		
		Nodes Removed (%tot.)	Edges Removed (%tot.)	Call sites Resolved (%poly) (%tot.)	Nodes Removed (%tot.)	Edges Removed (%tot.)	Call sites Resolved (%poly) (%tot.)
raytrace	VTA	923 (39%)	4609 (44%)	261 (67%) (3.3%)	27 (12%)	76 (4%)	3 (30%) (0.1%)
	VTA+VTA	1191 (50%)	5581 (53%)	343 (89%) (4.3%)	29 (13%)	78 (4%)	3 (30%) (0.1%)
compress	VTA	921 (41%)	4628 (50%)	262 (69%) (3.8%)	22 (27%)	80 (8%)	4 (67%) (0.4%)
	VTA+VTA	1190 (53%)	5600 (60%)	344 (90%) (5.0%)	24 (29%)	82 (8%)	4 (67%) (0.4%)
jack	VTA	926 (37%)	5235 (43%)	659 (83%) (7.3%)	28 (8%)	630 (17%)	382 (96%) (12.4%)
	VTA+VTA	1189 (48%)	6163 (51%)	740 (93%) (8.2%)	30 (9%)	632 (17%)	382 (96%) (12.4%)
javac	VTA	701 (18%)	6248 (21%)	953 (49%) (6.3%)	36 (3%)	1705 (9%)	222 (23%) (2.6%)
	VTA+VTA	721 (19%)	6410 (22%)	956 (50%) (6.3%)	36 (3%)	1719 (9%)	222 (23%) (2.6%)
sablecc	VTA	408 (8%)	7455 (18%)	507 (34%) (3.0%)	51 (3%)	5139 (19%)	267 (30%) (3.9%)
	VTA+VTA	476 (10%)	7765 (19%)	532 (36%) (3.1%)	54 (3%)	5268 (20%)	277 (32%) (4.1%)
soot	VTA	456 (12%)	8993 (25%)	785 (40%) (5.0%)	157 (6%)	6529 (22%)	651 (36%) (5.0%)
	VTA+VTA	534 (14%)	10205 (28%)	954 (48%) (6.1%)	225 (10%)	7690 (26%)	820 (45%) (6.3%)
pizza	VTA	360 (11%)	3702 (18%)	230 (30%) (1.5%)	75 (4%)	1577 (10%)	100 (17%) (0.9%)
	VTA+VTA	367 (12%)	3836 (19%)	241 (31%) (1.6%)	75 (4%)	1645 (11%)	110 (19%) (0.9%)

Table 2: Improvement of Call Graph over Conservative Call Graph

4 Time and Space Complexity of the Algorithm

It is claimed in Vijay's thesis that the size of the type propagation graph grows linearly with the size of the program. Since the analysis uses only linear algorithms, one expects the analysis to run in linear time and uses linear space. This is indeed the case. Table 4 gives the size of the data structures used in the program, as well as the running time of the analysis. The two columns under "Type Graph" shows the size of the VTA constraint graph during the first pass (VTA) and the second pass (VTA+VTA) respectively. The corresponding data for the condensation graph is given under "SCC Graph". The three columns under "Running time" correspond to the execution times for the first pass (in seconds), the second pass (in seconds) and the total running time of Soot (min : sec) respectively. Note that for each stage, the given times indicate the number of seconds needed to (1) build the type propagation graph, (2) solve the constraint graph, and (3) trim the conservative call graph based on the results of VTA. The memory usage is the maximum program

Name		Current Implementation			Thesis Implementation		
		Nodes	Edges	Call sites	Nodes	Edges	Call sites
raytrace	VTA	190	2019	7	189	2015	7
	VTA+VTA	188	2017	7	189	2015	7
compress	VTA	61	890	2	60	887	2
	VTA+VTA	59	888	2	60	887	2
jack	VTA	317	3102	14	316	3099	14
	VTA+VTA	315	3100	14	316	3099	14
javac	VTA	1350	16991	737	1153	15103	713
	VTA+VTA	1350	16977	737	1153	15103	713
sablecc	VTA	1890	21519	612	1864	20713	578
	VTA+VTA	1887	21390	602	1863	20651	572
soot	VTA	2348	23449	1153	1912	26663	1035
	VTA+VTA	2280	22288	984	1892	25730	863
pizza	VTA	1714	13667	475	1724	13766	468
	VTA+VTA	1714	13599	465	1724	13766	468

Table 3: Pruned Call Graph Characteristics (Benchmark Only)

size, in megabytes, during the entire execution of Soot, as observed with the `top` monitor. These running times are obtained on a Pentium II 400MHz with 500 Mb of memory.

Name	Call Graph		Type graph		SCC graph		Running time			Memory
	$\|N\|$	$\|E\|$	VTA(1)	VTA(2)	VTA(1)	VTA(2)	VTA(1)	VTA(2)	Total	(Mb)
raytrace	2373	10444	20055	13558	14149	9163	95	62	10: 02	272
compress	2227	9284	18457	11959	12772	7787	85	55	8: 20	251
jack	2501	12146	21808	15310	15665	10678	120	89	12: 02	299
javac	3863	29491	36197	31594	25094	21547	349	246	26: 36	474
sablecc	4914	41329	43580	41356	32726	30917	432	*1515*	*61:57*	522
soot	3806	35987	36062	26458	26458	24338	459	301	32: 06	461
pizza	3147	20607	36947	34714	22363	20630	245	246	26: 08	404

Table 4: Size of Data Structures and Running Time

The data above shows that the running times scale linearly with the size of the benchmark for every benchmark, with the exception of `sablecc`, for which the running times for VTA+VTA and the total running time does not scale well with the size of the program. (See the italicized numbers above.) The reason for this is lack of memory. Since our computers only have about 500 megabytes of physical memory, and since `sablecc` requires more memory than what is physically available, memory swapping occurs, and performance is significantly affected. This shows that memory usage is a crucial factor in determining the performance of an analysis. While a degradation in performance is observed in only one benchmark, we certainly hope to improve the algorithm in the future so that it uses less memory.

5 Assumptions and Limitations

5.1 The Effects of Entry Points

For our purposes, an *entry point* of a program is a method that is called implicitly by the virtual machine; in other words, it is a method that is called at some point during the execution of the program, but there are no explicit invoke statements referring to that method in the bytecode. For example, the `void main(String[] args)` method in the main class is an entry point of the program.

It is important for any inter-procedural analysis to correctly detect the entry points of the program. In the case of VTA, complete and precise detection of entry points is needed in order to build a correct invoke graph (based on CHA) which includes as few spurious methods as possible. If an entry point is left undetected, the transitive closure of that method may be left out in the worst case, which causes the results of the analysis to be incorrect. On the other hand, if spurious entry points are included, unreachable methods may be thought to be reachable; this may cause the instantiation of spurious types, which can then affect the whole analysis.

The entry point detection problem is a non-trivial one. While in the benchmarks there is usually only one entry point (the main method), this is not the case in library code. For example, the creation of the default thread-group is done by instantiating the class `java.lang.ThreadGroup`, and this is called implicitly by the virtual machine. Entry points may also be vary from one VM to the next: the method `<java.lang.System: void initializeSystemClass()>` is an entry point for the Sun JVM, but not for the Kaffe VM. In general, entry points can be divided into several categories:

1. **Application entry points:** this includes the `main` method, as well as `start` and `run` methods for subclasses of `java.lang.Thread` and classes that implement `java.lang.Runnable`. These are the most important entry points, since the execution of benchmark code starts at these methods.

2. **Static initializers:** the static initializer of a class `<clinit>` is called when a static method or field belonging to that class is referenced for the first time.

3. **Finalizers:** if a class declares a `void finalize()` method, it will be called before an instance of that class is garbage-collected.

4. **Constructors of dynamic classes:** when a class is instantiated via `newInstance`, its default constructor (the one that takes no parameters) is implicitly called. So the default constructors of any dynamically loaded classes must be included as entry points.

5. **Event handlers:** a method that handles incoming events, such as a mouse click, will not be explicitly called by any code, since events can only be detected by the virtual machine at runtime. These entry points are important for any application that makes significant use of the `awt` package.

6. **System Initialization methods:** these are methods that are called during the bootstrapping and initialization stages in the life cycle of the virtual machine. These entry points are the hardest to detect, since they are usually VM-dependent and are transparent to to the user.

In our current implementation of VTA, we are confident that entry points of categories 1 – 4 are properly dealt with by the analysis. However, since none of our benchmarks uses the `awt` package heavily, it is possible that entry points involved with `awt` may not be properly detected. As for category 6, we have tested the results of our analysis using the VM and library code in Blackdown JDK1.1.8, so we believe that entry points specific to this VM are dealt with properly. However, because of VM-dependence, we will not draw the same conclusions for other virtual machines.

5.2 Native Methods

Native methods are methods which are implemented in languages other than Java, but which can be called like any other method by a Java program. Native methods are usually platform/VM dependent, and their effects cannot be detected by static analysis. In the current implementation, as well as the thesis implementation, the effects of native methods are summarized manually by examining the source code for these native methods. Hence, the current native adjustor is only tested on the VM in Blackdown JDK1.1.8, and may not be valid in other VMs. We hope that in the future there will be a way for the user to inform Soot about the effects of native methods. We should at least have several native adjustors that will work for the most popular virtual machines on a few platforms.

5.3 Dynamically loaded classes

As mentioned at the end of Section 2.3, the user must explicitly specify any dynamically loaded classes on the command line, since Soot cannot statically detect which classes will be instantiated by the `newInstance` method. The command-line option `--dynamic-path` can be used to specify a path under which all classes are assumed to be dynamically loaded. Unfortunately, this option does not handle packages correctly. For this reason, we have added the command-line `--dynamic-packages`, which designate all classes in the specified packages as dynamic classes. It would be convenient if the user could specify the names of each dynamic class in a file and pass it as an input to Soot. While this feature is absent in the current implementation, this should be done in the near future.

6 Implementation Details

The Variable Type Analysis algorithm resides in the `soot.jimple.toolkits.invoke` package. A brief description of the classes follows; the presence of an asterisk indicates that the class has been created or modified since the last release.

`AccessManager` methods for checking Java visibility and scope requirements. Helps determine whether it is safe to inline a particular call site.

`CallGraphStats(*)` a data structure that holds information about the call graph.

`ClassHierarchyAnalysis(*)` builds an invoke graph based on Class Hierarchy Analysis.

`ClosedInvokeGraph` an invoke graph that thinks it knows about the whole universe (not used in VTA).

`InlinerSafetyManager` methods for checking safety requirements for inlining, as defined in Vijay's thesis.

`InvokeGraph(*)` a strictly bipartite graph that links call sites to target methods.

`InvokeGraphBuilder` a scene transformer that calls ClassHierarchyAnalysis to obtain an active invoke graph.

`MethodCallGraph(*)` a directed graph whose nodes are methods and whose edges are call edges. It also knows about default entry points, and one can add more entry points using the api `addEntryPoint(SootMethod m)`.

`NodeTypePair(*)` a simple data structure that makes it easier to remember cast edges.

`SiteInliner` methods for inlining sites (used by StaticInliner).

`StaticInliner(*)` a scene transformer that inlines monomorphic call sites.

`StaticMethodBinder(*)` a scene transformer that finds monomorphic targets, creates static methods out of them and make the monomorphic call site call the static method instead.

`SynchronizerManager` utility methods for dealing with synchronization.

`ThrowManager` utility methods for dealing with traps.

`TypeGraph` an interface with no methods and which is implemented by `VTATypeGraph`. (Why is this useful?)

`TypeSet(*)` a bit vector implementation for flow sets with types as its elements. In retrospect, I should have used ArrayPackedSet instead.

`VTANativeAdjustor(*)` adjusts the VTATypeGraph to take into account the effects of native methods.

`VTATestingFramework(*)` a transformer that runs VTA and inject profiling code into the bytecode. The resulting class files will print out error messages if the runtime type of an object is not detected by VTA. Useful for debugging only. Not registered under the `wjop` pack.

`VTATypeGraph(*)` the VTA constraint graph. The constructor of this class builds the constraint graph based on the invoke graph it gets as an argument.

`VariableTypeAnalysis(*)` performs VTA: builds the constraint graph, solves it and trims the active invoke graph based on the results.

The following class lie outside the `soot.jimple.toolkits.invoke` package, but were modified to facilitate VTA execution. A brief description of the modifications follows.

`Main` (`soot` package) added the command-line option `--dynamic-packages`.

The scene transformers `StaticMethodBinder` and `StaticInliner` have been modified to accommodate the new integer option `VTA-passes`, which allows the user to specify how many passes of VTA to apply. If set to 0, VTA will not run, and the invoke graph will be built using CHA by default. The default values for `VTA-passes` are 0 for both `StaticMethodBinder` and `StaticInliner`.

7 Conclusions and Future Work

In this paper, we have described the algorithm used in the current implementation of VTA, and we have compared it with the one presented in the thesis. The algorithm itself is quite similar to the one presented in the thesis, except for the treatment of array variables, exception edges and classes instantiated via the `newInstance` method. We saw that the two implementations produce comparable results when applied to a set of benchmarks, and the time and space complexity is linear in the size of the program, as expected. We have also discussed some limitations of the current implementation, which can be solved if we have a complete set of entry points available. Finally, we have given a list of classes which play a role in the analysis.

Several improvements can be carried out on the analysis in order to obtain better results, and to lift some of the restrictions imposed by this implementation:

1. Improve on the conservative call graph. The current algorithm, as well as the one presented in the thesis, is a pessimistic algorithm; it is based on an invoke graph built using pessimistic CHA. An optimistic invoke graph builder, using either CHA and RTA, can cause VTA to give better results because the original invoke graph is more precise. oCHA and oRTA algorithms are currently being developed by Charles Godin; however, those algorithms still need to be adjusted to take into account of dynamic classes.

2. Reduce memory usage. This can be done by (a) implementing special-purpose data structures to replace standard data structures provided by JDK1.2; (b) consolidate the use of data structures to reduce duplication to a minimum; and (c) use pointers to call sites rather than string representations in the VTA constraint graph.

   Alternatively, one can modify the VTA algorithm so that it builds only parts of the constraint graph at a time. The algorithm can build the part of the constraint graph that belongs to a particular method, and then summarizes that subgraph by eliminating local variables. These subgraphs can then be merged at a later time to obtain the complete VTA graph. This idea has been explored in some detail by Guillaume Marceau.

3. Provide a way for the user to inform Soot of the effects of native methods. One way is to make Soot accept text files that summarizes the effects of a particular native method. We can then create, for each VM, a library of summary files for native methods specific to that VM.

4. Provide a way for the user to inform Soot of VM-specific entry points. Again, a text file approach would be appropriate here.

5. Provide a way for the user to inform Soot of dynamically loaded classes. The user should be able to list those classes in a text file and pass it as an input to Soot.

References

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallee-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, MN, June 2000.

[Sun99] Vijay Sundaresan. Practical techniques for virtual call resulution in Java. Master's thesis, McGill University, June 1999.