Soot-based Implementation of a Demand-Driven Reaching Definitions Analysis

Longwen Lu[1], Cheng Zhang[2], Jianjun Zhao[1]
[1]School of Software, Shanghai Jiao Tong University
[2]Department of Computer Science and Engineering, Shanghai Jiao Tong University
longwenlu@gmail.com, {cheng.zhang.stap, zhao-jj}@sjtu.edu.cn

ABSTRACT

As a classical data-flow analysis, reaching definitions analysis is the corner stone of various techniques, such as code optimization and program slicing. The built-in data-flow analysis framework in Soot has been implemented in the traditional iterative style. While being able to meet general requirements for implementation of data flow analyses, the framework may be less efficient for a certain type of analyses in which the complete data-flow solution is unnecessary.

In this paper, we introduce our Soot-based implementation of an inter-procedural demand-driven reaching definitions analysis. For a demand for reaching definitions facts, the analysis only explores relevant program points and variables, saving a considerable amount of computation. Preliminary results show that the implementation can be much more efficient than its traditional counterpart in several scenarios. The Soot framework has greatly facilitated the implementation by providing abundant basic analysis results via well designed APIs.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Design,Performance

Keywords

Demand-driven, Reaching definitions, Inter-procedural static analysis

1. INTRODUCTION

As a classical data-flow analysis, reaching definition analysis has been extensively used in various techniques, including code optimization, bug finding, refactoring, etc. Traditional iterative algorithms of data-flow analysis compute complete data-flow solutions, that is, they compute for **every** program point the data-flow set in terms of **all** the variables in the program [1]. For example, in reaching definition analysis, the traditional iterative algorithm computes for every program point, p, the set of variable definitions that may reach p. Although being correct, such an algorithm may be unnecessarily costly, especially for large programs. Moreover, the complete data-flow solutions are also unneeded for several widely used techniques. For example, in program slicing, we are only interested in the reaching definitions of the variables that are relevant to the slicing criterion.

To improve the efficiency of data-flow analysis, Duesterwald et al. have proposed a demand-driven data-flow analysis framework [2] [3]. The basic idea of the framework is to formulate a demand for data-flow facts as a query and propagate it from a proper start point in the opposite direction to that of the corresponding traditional algorithms. The efficiency gain is achieved by early termination of the propagation. Although the demand-driven analysis has the same time complexity as the traditional analysis in the worst case, the empirical study [2] shows that it is generally more efficient in practice.

Our motivation of implementing the demand-driven reaching definitions analysis stems from our demand for a fast static slicer for Java programs. The data-flow analysis framework in Soot focuses on the traditional iterative algorithms and thus it provides complete reaching definition information. However, in static program slicing, analyzing data dependencies by computing reaching definitions in a demand-driven style is probably more efficient. Therefore, we have implemented the demand-driven reaching definitions analysis without using the generic data-flow analysis framework in Soot. Besides supporting our program slicer, the implementation can serve various client applications, which need partial information of reaching definitions, with higher efficiency. We believe our implementation can enrich Soot users' toolboxes by providing an alternative way of reaching definitions analysis.

The implementation is built based on the Jimple [4] intermediate representation of Soot. As the implemented analysis is inter-procedural, we use the built-in call graph provided by Soot. Using Spark [5], we are able to obtain more concise call graphs, making the analysis results more accurate. Although we have some minor technical problems unresolved, our use of Soot has greatly facilitated the implementation.

The rest of paper is organized as follows. Section 2 gives an overview of the demand-driven analysis. Section 3 describes the design and implementation of the analysis. Sec-

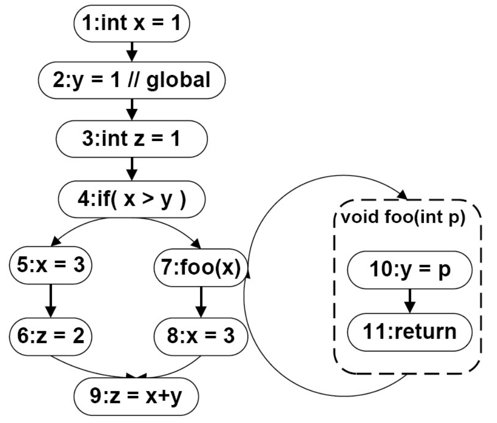

Figure 1: example for demand-driven reaching-definition analysis

tion 4 shows preliminary results on the comparison between the implemented analysis and the traditional reaching definitions analysis. Section 5 discusses our experience in using Soot and Section 6 concludes the paper.

2. DEMAND-DRIVEN REACHING DEFINITIONS ANALYSIS

The underlying assumption of the demand-driven data-flow framework is that it is often unnecessary to compute complete data-flow solutions (i.e., the data-flow sets at every program point). The basic idea is to generate queries for data-flow information on demand and propagate them along paths on the control flow graph (CFG), collecting data-flow information during the propagation. Sometimes, the answers to the queries can be determined by checking a small number of paths, leaving a large portion of the program unexplored and thus greatly saving computation. The worst case time complexity of the demand-driven framework is the same as the traditional iterative framework. Nevertheless, Duesterwald [2] has shown that the demand-driven framework is usually much more efficient.

The original demand-driven framework is designed for a family of data-flow analyses, including reaching definitions, available expressions, live variables, and very business expressions. Therefore, the framework is generic, in that, it is not limited to specific meet and join operations (e.g., set intersection and union) , analysis directions (i.e., forward or backward) and the definitions of KILL and GEN set. By specialized the framework, Duesterwald has demonstrated the demand-driven copy constant propagation and reaching definition analysis [2].

In our implementation, we focus on the reaching definitions analysis. The demand-driven reaching definition algorithm aims to answer the question, *"can a definition d can reach a program point p in a program?"*. To find the answer, the analysis launches a query at the point p and propagates this query along the control flow paths. A query is represented by a triple $\langle d, p, v\rangle$ where v indicates the variable whose value is defined by d. Besides the query, there are two key components in demand-driven analysis: 1) query propagation and 2) function summary.

Query propagation. After a query, $\langle d, p, v\rangle$, has been generated, it is propagated along paths in CFG in the reverse direction of the corresponding traditional data-flow analysis. During the propagation, the query is checked against each statement visited to see whether the semantics of the statement can determine the answer to the query.

For example, the traditional reaching definitions analysis is a forward union data-flow analysis, that is, the iterative algorithm processes each node forward along CFG edges and the data-flow set at the entry of a node is the union of the sets at the exits of all its predecessors. As a result, the definition, d, can reach the program point, p, if and only if there exists any path, π, between the program point of d and p and there is no other definition of v on π.

Correspondingly, in the demand-driven reaching definition analysis, the query is propagated backward along CFG edges. At a merge point of branches, multiple copies of the query will be generated and propagated along all directions. If a sub-query q_i is defined as *can d reach p along a path π_i?* (the answer is `true` or `false`), then the answer of the original query q, which is interpreted as *can d reach p in the program?*, is the **disjunction** of the answers of all the sub-queries. Therefore, if the answer of any sub-query is `true`, then it is unnecessary to do further query propagation, since the answer of q must be `true`. At each CFG node, p, being visited, the analysis performs the following actions :

1. If a query cannot be resolved at the node, the query is propagated to all the predecessors of the node. For a predecessor, p' (of p), the query $\langle d, p, v\rangle$ is transformed into $\langle d, p', v\rangle$.
2. If a query is resolved at the current node and the answer is `true`, the algorithm terminates and returns the answer `true`.
3. If the query is resolved at the current node and the answer is `false`, the algorithm just removes this query and goes on processing the remaining unresolved queries.

In the end, if no query remains, the algorithm terminates and returns `false` as the answer to the original query. For a node other than a method call, the answer of the current query is resolved to `true` (or `false`), if the definition d is generated (or killed) by the node. For a method call, the algorithm uses the function summary to determine whether the definition is generated, killed, or neither.

Consider the CFG in Figure 1, suppose that the information of interest is the reachability of the definition of variable `x` in node 1 to node 9 and thus the initial query is $\langle 1, 9, x\rangle$. Because this query cannot be resolved at node 9, the algorithm propagates the query to the predecessors of node 9, namely nodes 6 and 8. Meanwhile, the query is transformed into $\langle 1, 6, x\rangle$ and $\langle 1, 8, x\rangle$ for node 6 and node 8, respectively. Afterwards, $\langle 1, 8, x\rangle$ can be answered immediately and the answer is `false` because node 8 kills the definition in node 1. Meanwhile, $\langle 1, 6, x\rangle$ remains unsolved and is propagated to its predecessors node 5. Finally, the answer to $\langle 1, 5, x\rangle$ is `false`, and no more query remains. Hence, the final answer is `false`. We observe that only the information relevant to variable `x` (not `y`) is considered in this case. Furthermore, the algorithm finds the correct answer by visiting merely four statements in the program and it avoids analyzing the method call at node 7.

Function summary. The demand-driven framework is inter-procedural, so the method calls which occur during the query propagation are taken into account. The framework computes a summary for a method rather than propagating the query into the method body at every call site to that

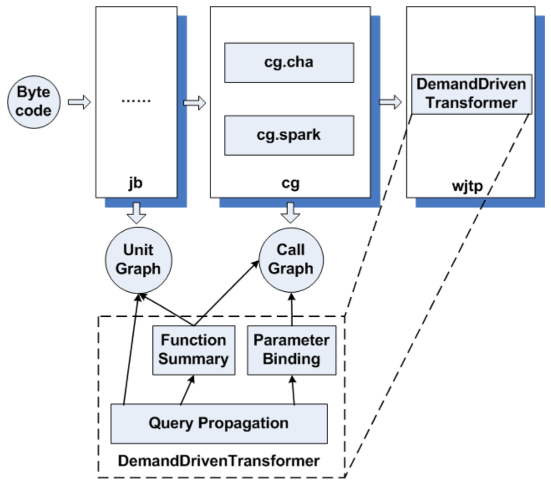

Figure 2: The work flow of the demand-driven reaching definitions analysis.

method. Note that the computation of a function summary is also demand-driven, that is, the framework only computes the summaries actually used in the query propagation. In this paper, we use the term "function summary", instead of "method summary", to keep in line with the terminology used in the original work [2] [3].

In the demand-driven reaching definition analysis, the function summary of a method, `m`, is computed by checking the CFG of `m` and the statements from exit to entry to see whether a definition is killed or generated in `m`. During the computation, summaries of the methods called by `m` (directly or transitively) are also computed if they are not yet available. A function summary is represented as two sets, namely KILL and GEN, which contain the definitions killed or generated in the method. The details of computing function summaries can be found in [2] [3].

Consider the example in Figure 1 again. This time we issue a query $\langle 2, 8, y\rangle$. Note that `y` is a global variable. When the query is propagated to node 7, the KILL and GEN information of 7 is computed. The algorithm summarizes the method `foo` invoked at node 7. Because the the definition of `y` at node 2 is killed by node 10 in method `foo`, the definition cannot flow through method `foo`. Hence, according to the function summary of `foo`, the definition of y at node 2 is killed and not generated at node 7. As a result, the answer to $\langle 2, 8, y\rangle$ is `false`.

3. DESIGN AND IMPLEMENTATION

In accordance with the framework, the implementation[1] of the demand-driven reaching definitions analysis consists of three components: basic entities, query propagation, and function summary computation. We directly use the Jimple intermediate representation [4], instead of extending the existing data-flow framework in Soot. Such a design decision gives us the flexibility to implement the analysis in a demand-driven style which is quite different from the traditional iterative data-flow analysis algorithms. However, due to the design decision, the current implementation suffers from loss of efficiency, because it does not reuse the highly efficient bit array data structure used in the original data-flow framework to represent data-flow set. In our future work, we will try to optimize the implementation.

Figure 2 shows the work flow of the implementation. The entry of the analysis is implemented by a scene transformer (a subclass of the class `SceneTransformer`), that is, the analysis will be triggered during the Whole-Jimple Transformation Pack (wjtp) in Soot. Meanwhile, Soot must be invoked with the whole program analysis mode enabled to use the analysis. The Jimple bodies of methods, the corresponding CFGs, and the call graph are the major input of the analysis. Points-to information provided by Spark [5] is optional in the analysis. The usage of these kinds of Soot-provided information is described in detail in the rest of this section.

Basic Entities. There are two basic concepts in reaching definitions analysis, namely *definition* and *program point*. In the implementation, we design two basic entity classes, `RdValue` and `RdProgramPoint` to represent the two concepts. A program point is represented by an instance of `RdProgramPoint` and a definition is represented by a pair of instances of `RdValue` and `RdProgramPoint`. In intra-procedural cases, it is sufficient to use a single instance of `RdProgramPoint` to represent a definition, since the instance fully contains the information of the definition (i.e., the defined variable and the location). However, in inter-procedural cases, the variable defined in the statement may be bound to other variables in different calling contexts. Thus, we use an extra instance of `RdValue` to keep track of this information.

The underlying Soot data structures of `RdValue` and `RdProgramPoint` are `Value` and `Unit`, respectively. The main difference between our entity classes and those in Soot is that the former are enhanced (based on the latter) with extra context information. For example, the class `RdProgramPoint` has a field (of type `SootMethod`) to indicate the method that contains the program point. The context information is retained to facilitate several steps in the analysis. We obtain our enhanced entities (`RdValue` and `RdProgramPoint`) by extracting their corresponding entities (`Value`, `Unit`) and contexts (`SootMethod`) created by soot and holding references to them. We use other major information, such as the CFGs and call graph, in their original forms in Soot. Specifically, the CFGs that we use are unit graphs (rather than basic block graphs), because the analysis is designed to be at statement level.

Query Propagation. The central concept in query propagation is *query*. A query is a triple, $\langle d, p, v\rangle$, which is implemented by three instances: one of `RdValue` and two of `RdProgramPoint`. Once a query is generated, it is propagated backward along CFG edges, starting from p (more exactly the unit of p). In essence, for each program point, p_i, the propagation checks whether d is killed or generated by p_i and then (if necessary) finds the predecessors of p_i and performs the check against each of them. This process continues until the answer of the query is determined.

In the wjtp pack of Soot, a CFG can be obtained easily by first getting the active body of the method and then passing it to the constructor of the specific unit graph class. By using the APIs in `UnitGraph`, we can easily identify the predecessors of a unit in the CFG. In addition, since the analysis is inter-procedural, when the propagation reaches the entry of the current CFG, it must continue at every call site to the current method. We use the method `edgesInto` of class `CallGraph` to get all the call edges into the method and then iterate the list of edges to find the relevant call

[1]Available on `http://code.google.com/p/dd-reach/`

sites. When going across a procedural boundary backward, the name of a variable may be transformed in the reverse direction of the parameter binding at the call site.

The checking at each program point determines whether the statement kills or generates the definition of interest, or neither. In the intra-procedural case, which is thoroughly discussed in the literature (i.e., [1]), a definition is killed by a statement if and only if the statement is an assignment and the left-hand-side variable is the same as the one of the definition. In contrast, there is exactly one statement that generates the definition. Because Jimple is a three-address code intermediate representation, there can be at most one assignment contained in a single Jimple statement. In other words, complex statements containing multiple assignments (i.e., `a = (b = 1);`) can never exist in Jimple. As a result, for a program point (i.e., a Jimple statement), we do the check in the following steps:

1. Check whether the statement is an assignment or not.
2. If the statement is not an assignment, it will neither generate nor kill the definition.
3. If the statement is an assignment, we get the sole left-hand-side variable of the statement and check whether it is the same variable as the one in the query. If it is the case, the definition is killed by the statement.
4. A special case is that the statement is exactly the one representing the definition and the left-hand-side variable is consistent with the variable in query. In this case, the definition is generated by the statement.

In the inter-procedural case (i.e., when the program point is a method call), the analysis has to determine whether the method call kills or generates definition, by looking up the function summary of the callee.

Function Summary Computation. The function summary of method m tells which definitions are killed or generated in m. Therefore, the summary of m can be viewed as two functions, $KILL_m : D \to B$ and $GEN_m : D \to B$, where D and B stand for the set of all definitions and boolean values (i.e., {true, false}), respectively. If a definition d is killed in m, it is killed on every path from the exit of m to the entry of m. If a definition d is generated in m, it is generated by a statement that is reachable from the exit of m.

The function summary computation is demand-driven. Since there is only one variable in each query, when the query reaches a method call, the algorithm computes a partial function summary which merely tells whether the variable in the query is killed or generated. In other words, the method's Kill-Gen information of other variables is not computed. Similar to query propagation, function summary computation starts from the method exit and checks backward along CFG edges. It is worth noting that for a method call encountered during the function summary computation of m, the function summaries of the callees of m are computed on demand. The formal definition of function summaries and the detailed algorithm to compute them can be found in the original publications [2] [3].

In the implementation, we use two static fields (of type `Map<Pair<RdValue, RdProgramPoint>, Boolean>` to store the function summaries for all the analyzed methods. One map is used to store KILL and the other is used to store GEN. The `Pair` is a utility class to represent the combination of two classes. In a `Pair` as a key in the `Map`, the `RdValue` represents the variable whose Kill-Gen information is stored, while the `RdProgramPoint` is the entry point of the method of interest. The Boolean value in the map indicates whether the `RdValue` is killed or generated in the method whose entry is the `RdProgramPoint`. Here we use the entry point to represent a method, in order to keep in line with the original algorithm.

In function summary computation, we use statement-level CFGs built from the active bodies of methods to do the backward exploration. When dealing with method calls, the implementation has to take into account all of the virtual call targets (if the call is a virtual call). This is an extension by our implementation, because the original algorithm does not target object-oriented programs. As the reaching definition analysis is a union data-flow problem, the KILL value of a virtual call site is the conjunction of the KILL values of all the targets. Correspondingly, the GEN value is the disjunction of those of the targets. Using the method `edgesOutOf(Unit)` of class `CallGraph`, we can easily get the list of virtual call targets and then compute the Kill-Gen information based on their function summaries.

Parameter Binding. At call sites, definitions of variables may go across the procedural boundaries. There are two kinds of variables, global and local. In our implementation, the universe of global variables consists of all static fields in the entire program. For a global variable, which is visible in both the caller and callee, whether its definitions can reach a program point in the callee generally depends on redefinitions on the CFG paths with respect to its original variable name. Moreover, definitions of either local or global variables may reach program points in the callee via parameter binding. At a call site, since actual parameters (in the caller) are bound to their corresponding formal parameters (in the callee), their definitions that reach the call site are passed into the caller. Therefore, when the query propagation reaches a method entry, if the variable v in the query $\langle d, p, v\rangle$ is a formal parameter of the callee, the analysis has to trace back into the caller to check whether the definition is killed or generated in the caller. Two complications are caused by parameter binding:

1. The names of the actual parameters are different from those of the corresponding formal parameters.
2. One variable can be used as multiple actual parameters and thus bound to multiple formal parameters.

Two relations, namely *bind* and $bind^{-1}$, are built to enable mapping between variables in different method contexts during inter-procedural query propagation. The two relations are defined as $bind : V \times P \to 2^V$ and $bind^{-1} : V \times P \to V$, where V is the set of variables and P is the set of program points. Given a pair of a variable (actual parameter) and a call site, *bind* maps it to the corresponding set of formal parameters. In contrast, given a pair of a variable (formal parameter) and a call site, $bind^{-1}$ maps it to the corresponding actual parameter.

In the implementation, we analyze every call site in the active body of every method, in order to gather the binding information and build the two relations. For each call site, we extract and store, in *bind*, the mapping between each actual parameter and its corresponding argument indices. It is sufficient to use an argument index to uniquely identify the formal parameter in the callee. It is worth noting that we do not actually store the relation $bind^{-1}$. When reverse binding is needed during query propagation, the argument index of the variable is determined using the method `getParameterLocal(int)` of interface `Body` and then the corresponding actual parameter is found by looking up *bind* using the call site and the argument index as the key.

In the query propagation, for a query, $\langle d, p, v\rangle$, the definition d must be consistent with v, that is, if a program point is a statement assigning value to v, then d must be killed by the program point. When d and v are in the same method, they are consistent if and only if the left-hand-side variable of d (noted as d_{lhs}) is the same as v. When d and v are in different methods, v must be the variable that d_{lhs} is bound to (via parameter binding) or they are the same variable (if they are globals). We have to ensure such consistency at the beginning of the query propagation. When the propagation goes on, the consistency will be maintained as long as the reverse binding is used correctly.

A query essentially asks the question: *can a definition d reach a program point p?* Therefore, at the beginning of the query propagation, only d and p are specified, whereas v is derived from d and p. The program point p is the start point of the query propagation. If d and p are not in the same method, we use *bind* to determine the variable v (at p) which is consistent with d. To this end, we first find all the call chains between m_d (the enclosing method of d) and m_p (the enclosing method of p) and then do a sequence of parameter binding (along each call chain) using *bind* to obtain v based on the variable of d. As there can be multiple call chains between m_d and m_p, we may get a set of variables to represent v (noted as $\{v_1, v_2, ..., v_n\}$). Since the analysis is a union problem, we use all the variables to form a set of queries, $\{\langle d, p, v_1\rangle, \langle d, p, v_2\rangle, ..., \langle d, p, v_n\rangle\}$, and use the disjunction of their results as the result of the original query.

Using Alias Information. As a static analysis, the demand-driven reaching definitions analysis can have its accuracy improved by using alias information. In our implementation, the improvement can be achieved in two aspects.

The first one is call graph pruning. As described above, for a virtual call site, the analysis takes into account all the virtual call targets and computes the overall result based on the result of every target. Consequently, the analysis result may be overly conservative, because some targets can never be reached at runtime. Thanks to the points-to information provided by Spark, some superfluous call edges can be pruned from the call graph. As a result, we can obtain more accurate results for the inter-procedural part of the analysis by using Spark.

Moreover, there are complex data structures, array and class with fields, in Java. Unlike the primitive types, these structures can be read and written partially. As this problem, we make a conservative approximation. An assignment to a single element of an array (or a field) is treated as an assignment to the entire array (or to the base of the field). Similarly, a use of an array element (or a field) is treated as a use of the entire array(or the base of the field). The references to complex structures act like pointers under our approximation. Consequently, the reaching definitions results can be affected by alias. In absence of alias, as described in Section 3, the answer to a query $\langle d, p, v\rangle$ is true if and only if d equals to p and d is a definition to variable v. However, d is not necessarily a definition to variable v when alias is considered. It is sufficient that d is a definition to a may-alias of v to make the answer true. We extracts the may-alias results from the points-to sets computed by Spark to refine the analysis results.

4. PRELIMINARY EVALUATION

Experiment Design. To evaluate the efficiency improvement gained by the demand-driven analysis, we compared the implementation with Soot's existing reaching definitions analysis. In Soot, `LocalDefs` defines the interface of reaching definitions. Out of its four implementations, we choose `SimpleLocalDefs` and `SmartLocalDefs` for our comparison, because other implementations are either composed of several analyses or analysis based on Shimple (an SSA variation of Jimple). `SimpleLocalDefs` and `SmartLocalDefs` implement intra-procedural reaching definitions analyses on the top of the traditional iterative data-flow analysis framework in Soot. As these analyses compute the solution of all program points, in the experiment, we obtain the complete solutions with demand-driven analysis by issuing adequate number of intra-procedural queries. We carried out the experiment on a desktop with 2.4GHz CPU and 4GB memory running 64-bit Windows 7 and Java 1.6. The benchmark is Apache Velocity[2], an open source project for web application development. After filtering out trivial methods, such as setters and getters, we applied the three algorithms to 844 methods in the benchmark. The sizes of methods vary from 30 to 1020 Jimple statements.

Table 1: Experimental results. (Analysis time is measured by nanosecond.)

#U	#D	#Q	Simple	Smart	DD	DD/Q
69	43	89	303359	538548	5233170	17103

Results. Table 1 shows the experimental results. Among the columns, '#U' and '#D' show the average number of statements and definitions per method, respectively; '#Q' shows the average number of queries needed for demand-driven analysis to obtain complete solutions per method; 'Simple', 'Smart' and 'DD' show the average analysis time for complete solutions of `SimpleLocalDefs`, `SmartLocalDefs` and the demand-driven analysis, respectively; 'DD/Q' shows the average analysis time per query in the demand-driven analysis.

As shown in Table 1, in the single-query case (answer to one query is needed), the demand-driven reaching definition analysis is much more efficient than the traditional ones (with about ×17 and ×31 speedups respectively). This result is reasonable, because the demand-driven analysis only computes a partial solution, while the traditional analyses need to compute the complete solution. However, the demand-driven analysis is less efficient than the traditional ones when the full solution is computed. This may be caused by that the demand-driven algorithm recomputes some information inevitably because the intermediate results are not shared between different queries.

Figure 3 shows the speedups of demand-driven algorithm to the traditional iterative ones in single-query case more detailedly. The x-axis is the intervals of speedup and the y-axis is the number of instances in the corresponding interval. The speedups vary from 0.4 to 228.8 times, but most of them distribute in a relatively small interval around 20-30

[2]`http://velocity.apache.org/`

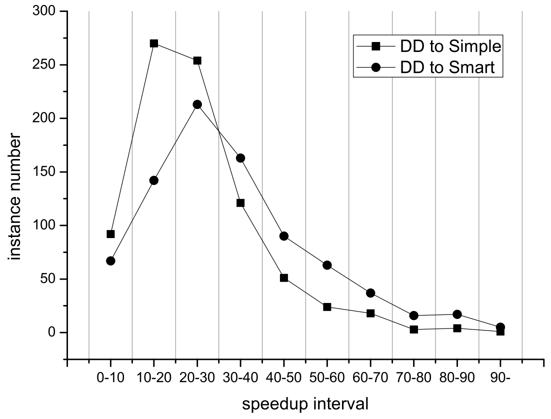

Figure 3: Speed up gained with demand-driven in single-query case

times. The figure indicates that we can expect an improvement of efficiency using demand-driven algorithm when the query number is under 25 (with respect to the benchmark used in our experiment). It is also worth noting that the speedup may be higher since our implementation can be further optimized. In brief, although the demand-driven reaching definitions analysis cannot replace the traditional analysis, we can have significant efficiency improvement by using the demand-driven analysis when only partial dataflow facts are needed.

5. DISCUSSION

Experience of Using Soot. As described in Section 3, using Soot has greatly facilitated our implementation of the demand-driven analysis, even though we have not used the existing data-flow framework in Soot. The implementation is built on the Jimple IR of Soot. On the one hand, as Jimple is the major IR of Soot, most of the built-in analyses are available for Jimple. On the other hand, Jimple, which is in the form of three-address code, is quite suitable for our implementation in three aspects:

1. It is easy to identify the variable of a definition, since there is exactly one left-hand-side variable.

2. It is easy to identify and exact information from a method call statement, because there is at most one method invocation expression.

3. Jimple programs are generally easier to understand than Java bytecode, because Jimple is stackless and it retains clear type information. Moreover, Jimple statements look more similar to Java statements.

The query propagation and function summary computation heavily rely on the control flow graphs and the call graph. Fortunately, Soot provides APIs to build these graphs straightforwardly. Additionally, Spark provides readily available points-to information and the pruned call graph, making it easy to experiment on the analysis with or without considering aliases.

In general, the APIs of Jimple statements, the control flow graph, and the call graph are well designed. For example, method `getLeftOp()` of interface `DefinitionStmt` directly gets the sole left-hand-side variable in an assignment statement; method `getPredsOf(Unit)` of class `UnitGraph` gets all the predecessors of the specified unit in the control flow graph; method `edgesInto(MethodOrMethodContext)` of class `CallGraph` can be used to find all the call edges that have the specified method as the target.

Potential Applications. Because our motivation for implementing the demand-driven analysis is to develop a static slicer, a straightforward application of the implementation is a demand-driven data dependence analyzer. Focusing on a slicing criterion, the analyzer can use the implementation to find out the relevant definitions that may reach and be used in the statements under investigation. Such a demand-driven data dependence analyzer is supposed to be more efficient than its counterpart using traditional reaching definitions analysis, because it may avoid computing irrelevant reaching definitions.

6. CONCLUSION

Based on Soot, we have implemented a demand-driven reaching definitions analysis. The implementation has been well supported by the built-in analyses and APIs in Soot. Preliminary evaluation shows that, in its target scenarios, the implementation is much more efficient than the existing reaching definitions analysis in Soot. We believe various applications can benefit from the implementation.

In our future work, we will implement the static slicer based on a demand-driven data dependence analysis using the implementation. The reaching definition analysis is just one specialized instance of the original demand-driven dataflow framework. We are planning to use Soot to implement the generic framework and then several data-flow analyses can be implemented by extending the framework.

7. ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China (NSFC) grants 60673120 and 91118004, and Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), China.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
[2] E. Duesterwald. *A demand-driven approach for efficient interprocedural data flow analysis*. PhD thesis, Pittsburgh, PA, USA, 1996. AAI9727844.
[3] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, Nov. 1997.
[4] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.
[5] O. Lhotak. Spark: A flexible points-to analysis framework for java. Master's thesis, McGill University, December 2002.