Generating Run-Time Progress Reports for a Points-to Analysis in Eclipse*

Jason Sawin
Ohio State University

Mariana Sharp
Ohio State University

Atanas Rountev
Ohio State University

ABSTRACT

Eclipse plug-ins have access to a rich collection of GUI components. One such component is the *progress bar*, which allows a long-running task to provide Eclipse users with feedback about its progress. This paper considers the problem of providing precise progress bar reports for plug-ins that perform static code analysis. Since static analyses often take a long time to execute, progress indicators can inform the user whether the analysis is actually making progress, and how long it will take to complete. This type of responsiveness is essential for providing positive user experience.

In this paper we consider points-to analysis, which is a popular static analysis for object-oriented software. Reporting the run-time progress of a points-to analysis requires heuristics for *a priori* estimates of the total running time of the analysis. We define several such heuristics for a whole-program subset-based points-to analysis for Java, implemented as part of the Soot Eclipse plug-in. We also present an experimental evaluation of the heuristics on a large set of Java programs. These results provide useful insights for the creators of points-to analyses and other static analyses that will be built and distributed in Eclipse.

1. INTRODUCTION

Research on static code analysis aims to increase the productivity of software developers and to facilitate the creation of reliable software. However, without an intuitive, meaningful, and responsive user interface, it is doubtful that an analysis will achieve this goal on a large scale. That is why more and more static analysis researchers are choosing to implement their analyses as extensions to existing integrated development environments (IDEs) and to provide access to the analysis results using the GUI facilities of the IDE.

For static analysis of Java, Eclipse is the obvious IDE to extend — after all, one of Eclipse's defining features is the ease with which it can be extended. The Eclipse tool framework provides many low-level services on which static analysis can rely, such as parsers, intermediate representations of code, editors, etc. In addition, Eclipse also provides sophisticated GUI components that plug-in developers can use to create production-level user interface. One such GUI component that is often neglected or poorly implemented is the *progress bar*. These unostentatious components are an integral part of any well-designed user interface. They are especially apt for the UI of most static analyses. Often static analyses run behind the scenes and can take a considerable amount of time to complete. Without the use of a progress indicator, there is no way for the end user to determine (1) *if the analysis is actually making progress* and (2) *how long will the analysis take to complete*. This type of user responsiveness has been shown to be paramount for providing a pleasant user experience that creates product loyalty [2].

Even though Eclipse provides a rich API, conveying to the user the progress of most static analyses in a meaningful way is difficult. Precise tracking of a task's progress during execution is dependent upon prior knowledge of some measure of the total amount of work that will be performed by that task. This prior knowledge is generally not available for the first execution of a static analysis on a particular body of code. Thus, designers of static analyses must rely on heuristics to estimate the total amount of work that will be performed by their product. To ensure a successful user interface, these heuristics must be both lightweight (in terms of time and resources) and accurate. Such heuristics are also necessary for repeated executions of the analysis on successive versions of the same program; in this case, information from previous runs can be used to achieve greater accuracy.

This paper present our work on creating and evaluating several progress estimation heuristics for a points-to analysis available in Eclipse. *Points-to analysis* determines the objects pointed to by a reference variable or a reference object field. Such information plays a fundamental role as a prerequisite for many other static analyses. Our work is based on the open-source Soot plug-in [7, 6] and its Spark points-to analysis engine [3] which implements several state-of-the-art analysis algorithms. We considered the standard configuration for Spark's engine, which implements a subset-based whole-program points-to analysis. The goal of this paper is to define and evaluate several heuristics for producing (1) a priori estimates of analysis running time and (2) run-time estimates of analysis progress.

In earlier work [4] we considered similar questions for the run-time progress of the Rapid Type Analysis (RTA) [1] call graph construction algorithm implemented in our TACLE Eclipse plug-in [5]. This RTA implementation and the Spark points-to analysis differ significantly. The RTA implementation uses Eclipse ASTs for application classes together with pre-computed library summary information for the Java libraries. In contrast, Spark uses Soot's JIMPLE intermediate representation for all application and library classes. More importantly, the complexity of the two algorithms differs radically: RTA is essentially linear in the size of the program call graph, while the points-to analysis is potentially cubic in the number of statements in the intermediate

*This work was supported by an IBM Eclipse Innovation Grant.

representation. Thus, the techniques and results from [4] cannot be directly applied to the problem of estimating the run-time progress of the points-to analysis in Spark.

The contributions of this work are as follows:

- We define several heuristics for Eclipse progress indicators for a whole-program subset-based points-to analysis for Java. The approach considers both the case of a fledgling analysis (executed for the first time on a given program) and a repeated analysis (executed repeatedly on different versions of the same program).
- We present an experimental evaluation of the heuristics on a large set of Java programs. These results provide useful understanding for the creators of points-to analyses and other static analyses that will be built and distributed in Eclipse.

2. BACKGROUND

Our work extends the points-to analysis in Spark to utilize the Eclipse progress monitoring API. This section provides brief descriptions of the points-to analysis and the progress monitor interface in Eclipse.

2.1 Points-to Analysis

The Spark analysis engine [3] in Soot [7] implements a generic framework for points-to analysis of Java programs. Our focus is on Spark's default subset-based flow- and context-insensitive points-to analysis. The analysis takes as input an entire program, starting from a class containing a main method, and builds points-to sets which represent potential run-time points-to relationships.

The algorithm is based on a pointer assignment graph (PAG). PAG nodes represent the memory locations accessed by the program (e.g., local variables and fields of heap objects), and PAG edges represent the flow of pointer values to these locations due to assignments and parameter passing. Spark maintains a list of reachable methods, initialized with the main method and with the methods executed at JVM startup.

Spark offers a variety of options. The *worklist-based* algorithm with *on-the-fly call graph construction* is the default analysis used by Spark, and it is the one considered in this paper. (Our techniques can also be easily applied to the other variations of Spark's analysis.) The analysis maintains a worklist of PAG nodes whose points-to sets need to be propagated. In the main loop of the algorithm, nodes are removed from the worklist and the PAG edges associated with those nodes are processed. Whenever points-to relationships are added to the points-to set of a PAG node, the node is added to the worklist. With the on-the-fly option, methods are added to the list of reachable methods whenever the points-to set propagation produces new potential receiver objects at instance calls. Every time a new reachable method is discovered, the nodes and edges for its statements are added to the PAG and the node points-to sets are initialized appropriately.

2.2 Eclipse Progress API

The *Eclipse Jobs API* allows plug-in developers to partition their applications into tasks that can be executed in separate threads. The Jobs API also provides plug-ins with access to Eclipse's progress monitoring components. For brevity we will only discuss those methods in the Jobs API that are relevant to tracking the progress of a task. A task wrapped in a subclass of class `Job` is executed when its `run` method is called by the scheduler. This method takes as its only parameter an `IProgressMonitor` object. The scheduler passes to `run` an implementation of a progress monitor.

An implementation of interface `IProgressMonitor` defines the following key methods:

- `void beginTask(String taskName, int totalWork)`: this method notifies the progress monitor that a task has started. Parameter `totalWork` is the total number of work units the task is expected to complete.
- `void worked(int work)`: this method notifies the monitor that a task has completed the number of work units indicated by `work`.
- `void done()`: alerts the monitor that the task has completed its work.

These methods are used to drive the GUI progress bar provided by Eclipse. With each invocation of method `worked`, the monitor updates the progress bar. The progress bar will indicate 100% of the work completed when either the accumulative installments of `work` add up to `totalWork`, or method `done` is invoked. The plug-in designer must ensure that the progress monitor is initialized with the correct `totalWork` value, and that methods `worked` and `done` are invoked correctly.

3. ESTIMATION HEURISTICS

This section describes the heuristics that we developed to estimate the progress of the points-to analysis in Spark. There are two tasks that such heuristics must complete: (1) estimating the total amount of work to be passed to `beginTask`, and (2) where in the analysis and with what values should `worked` be invoked.

We propose several heuristics for two different circumstances: *fledgling analysis* and *repeated analysis*. A fledgling analysis is an execution of the analysis in the absence of historical information being saved from a previous run. Conversely, a repeated analysis is the case when the analysis has been previously executed on a particular application. This earlier execution saves relevant information which is utilized by subsequent executions of the analysis on modified versions of the same applications.

3.1 Fledgling Analysis

The techniques outlined in this section attempt to predict the total amount of work needed to be conducted by the points-to analysis. These estimates are made without the benefit of historical information. This case occurs when executing the analysis for the first time on a particular program.

Least Effort Estimate: During an execution of the points-to analysis, the bulk of the running time is spent in the loop in which every iteration removes and processes a single PAG node from the worklist. One possible estimation technique would be to predict the number of loop iterations needed to completely analyze an application, and then to call `worked(1)` for each iteration. However, the average number of such iterations for our benchmarks applications approaches 200,000. Monitoring progress at such a fine grain would be detrimental to analysis running time.

The approach we propose monitors progress at a coarser grain. Since the analysis performs work for only methods that are reachable from the main method, the total number of reachable methods can be used to determine a rough measure of progress. Our approach produces an estimate of the total number of reachable methods. This value is then passed to `beginTask` upon invocation of the points-to analysis. Every time the analysis discovers a new reachable method, `worked(1)` is invoked, indicating 1 unit of work has been completed.

The key issue for this technique is determining the estimated value of the `totalWork`. Since there is no way to precisely predict the total number of reachable methods that will be discovered by the points-to analysis, the simplest approach would be to hardcode some estimate. The hardcoded value used in our experiments was 5667. This value was attained by taking the average number of reachable methods across 17 sample Java applications. These sample applications *did not* include any benchmark applications used in the experiments presented in this paper.

Number of User-Defined Methods: This technique tailors its estimations to each unique application under analysis. To do this, it introduces a light pre-processing phases. This pre-processing phase utilizes the Eclipse API to determine the number of user-defined methods (i.e., non-library methods) in the application under analysis. Our previous work has shown this task to be very quick and unperceivable to the client [4]. This number is then used as the estimate of total work passed to `beginTask`. During the analysis, `worked(1)` is called immediately after a new user-defined method is discovered.

Total Derived from User Methods: This technique combines the approaches from above. It uses a hardcoded ratio to predict the total number of reachable methods as a percentage of user-defined methods. The ratio of user-defined methods to library methods was calculated for the 17 sample applications. These results indicate that on average 9.81% of the reachable methods in a call graph created by the points-to analysis are methods defined by the user. Using this result and the same preprocessing phase described above, the total amount of work can be estimated as `totalWork` $= \eta/.0981$, where η equals the total number of user-defined methods. During the analysis, `worked(1)` is called when any new method is discovered.

3.2 Repeated Analysis

Many static analyses may be invoked on successive versions of the same application. For example, during the development process, many programmers will use some form of *change impact analysis*. Such analyses are used for a variety of reasons — for example, determining the scope of a particular change, or deciding which unit tests should be rerun as a result of the change. Thus, if a fledgling analysis saves certain historical information and stores it in the Eclipse workspace structure, a subsequent repeated analysis could use that information to greatly improve the progress estimates. The following heuristics are dependent upon such historical information being available.

Number of Reachable Methods: Similarly to the **least effort** heuristic for the fledgling analysis, this technique uses the number of reachable methods to estimate the total amount of work. However, rather than hardcoding the prediction, this approach uses the number of reachable methods discovered during the last execution of the analysis on this program — more precisely, on the earlier version of the program at the time of that last run. During the worklist algorithm, every processed method (in user code or in library code) corresponds to a unit of work.

Methods Weighted by Number of Statements: In the points-to analysis, each statement in a method needs to be processed. Thus, a method which contains more statements will require more work then one with fewer statements. This technique attempts to account for this uneven distribution of work. When a new method is discovered, `worked` is invoked with the number of statements contained in that method. The total estimated amount of work passed to `beginTask` is the total number of statements processed during the analysis of an earlier version of the application.

4. EVALUATION OF PROGRESS REPORTS

This section presents the metrics we used to evaluate progress estimation techniques. One crucial measure of a progress monitor is how *accurately* it portraits the progress of the running application. For any one estimation point (i.e., an invocation of `worked`) accuracy can be measured by taking the accumulated estimated progress up to that point and calculating the difference between that value and the actual amount of progress made. To calculate these results we recorded (1) the heuristic's current estimate of progress at each call to `worked`, and (2) the actual system time for which the analysis has been running up to this point. Using this information we are able to calculate

$$p_i = (\Sigma_{k \leq i} \texttt{worked}_k)/\texttt{totalWork}$$

where $\texttt{worked}_k$ is the amount of work reported by the call to `worked` at time t_k. We also calculate $perfect_i = t_i/totalTime$ where t_i is the actual system time recorded at the ith estimation point and $totalTime$ is the total system time it took the analysis to complete. Using these results we compute

$$\Delta_i = |p_i - perfect_i|$$

We use the average value of Δ_i across all estimation points to measure the accuracy of a heuristic. The closer the average Δ_i is to zero the more accurate the heuristic.

Another important attribute of a progress monitor is how smoothly it reports progress. A progress bar that only indicates progress at 23%, 25%, and 85% would be very frustrating to the clients. Smoothness is a function of: the number of times `worked` is called, the estimated amount of work being completed with each call and the length of time between consecutive calls. Since all our heuristics invoke `worked` frequently, we concern ourselves with the last 2 factors.

We evaluate the *smoothness* of a progress monitor by comparing the slope of the "real" function to that of the "perfect" function, which has slope of 1. For each estimation point we calculate

$$\epsilon_i = |1 - (p_i - p_{i-1})/(t_i - t_{i-1})|$$

We use the average value for ϵ_i across all estimation points to provide the measure of smoothness achieved by a monitor. Ideally this value would be close to zero.

5. EXPERIMENTAL STUDY

This section presents an experimental study of the estimation techniques defined above. For the experiments we

Comp	.java	.class	Meths	ReachMeths
javacup_h	39	41	382	4030
javacup_i	39	41	382	4010
javacup_j	40	42	391	4035
jflex1.3.3	48	60	446	6148
jflex1.3.4	48	60	447	6149
jflex1.3.5	48	60	447	6149
jgraph5.7.4.6	58	137	1467	9330
jgraph5.7.4.7	58	137	1467	9330
jgraph5.8	59	137	1475	9342
jpws.2.0	89	141	1130	10108
jpws.3.0	103	187	1459	10611
jpws.3.1	104	193	1491	10665
sablecc.2.8	210	249	2108	5237
sablecc.3.1	198	267	2138	5572
sablecc.3.2	198	267	2137	5571
verbos.1.4	50	57	479	8571
verbos.1.5	52	58	513	8616
verbos.1.7	54	60	545	8546
vietpad.1.2	79	197	577	9231
vietpad.1.2.1	79	197	578	9236
vietpad.1.3	97	215	596	9269

Table 1: Subject Programs

used three different versions of open-source Java applications, as shown in Table 1. Columns ".java" and ".class" show the number of java files and class files present in the application respectively. Column "Meths" shows the number of user-defined methods in the program, while column "ReachMeth" shows the number of reachable methods reported by the points-to analysis.

5.1 Fledgling Analysis

Columns under (1) (2) and (3) in Table 2 contains the results for the three fledgling analysis (Section 3.1) estimation techniques. Columns "Avg Δ_i" show the average difference between p_i and $perfect_i$ for all estimation points, as described in Section 4. Recall that the closer the average values for Δ_i is to zero the more accurate the result. Columns "Avg ϵ_i" shows the average difference between the slopes of the estimation function and that of the perfect function (Section 4). Smaller values indicate a smoother result, with ideal values being close to 0.

Least effort estimate. The results of hardcoding the estimated `totalWork` varied widely from application to application. Applications with a total number of reachable methods close to the estimated total of 5667 produced a fairly accurate result. For example, `sablecc.3.1`, which has 5572 reachable methods, produced an average Δ_i of .04. This means, that on average, each estimation point for `sablecc.3.1` was only off the perfect estimation by 4%. Unfortunately the accuracy of this technique diminishes if the application under analysis has either a significantly greater or less number of total reachable methods then the hardcoded prediction. The most extreme cases being `jpws` and `vietpad`, both of which have a total number of reachable methods near 10,000. For these applications this technique produced a progress monitor which indicated 100% of the work being completed when in actuality only about 50% of the work had been finished. This result is extremely undesirable as it may lead the user into believing that the analysis has either malfunctioned or completed, causing them to terminate the process prematurely. The average value for ϵ_i across all applications was fairly low, indicating that reporting progress at the reachable method level produces a satisfactory result.

Number of user-defined methods. The columns under the heading (2) in Table 2 display the results for the estimation technique based on the number of user-defined methods. On average this technique did not produce a more accurate result than the **least effort** method. In some cases it produced a significantly worse result. There are several variables which can cause imprecision in this technique's estimation of progress. The first being that not all user methods will necessarily be reachable from main. For example, methods that are only used for testing or are defined in dead coded will not be included in the call graph but will be counted in the preprocessing phase of this technique. The other major contributing factor to this technique's lack of accuracy is the fact that not all calls to user-defined methods are evenly dispersed throughout an application. The amount of time spent processing library methods could vary widely between the processing of any two user-defined methods. This last factor also explains why the smoothness of this technique is significantly worse than that of (1). These bursts of reporting progress followed by long periods of silence creates a very erratic progress monitor and consequently an estimation function with very irregular slope. This technique has one distinct advantage over its predecessor in that it did not stall at 100%. For all our benchmark applications the preprocessing phase of this technique discovered slightly more user-defined methods then were actually reachable. This resulted in the progress bar never quite reaching 100% before the application called `done`.

Number of total reachable methods calculated as a percentage of user-defined methods. Technique (3) corresponds to the approach which attempts to predict the total number of reachable methods as a percentage of the user-defined methods. This technique produced the least accurate result of all the techniques considered for the fledgling analysis. This is due to the fact that not all the applications conform to the estimate that user-defined methods comprise 9.81% of the total number of reachable methods. For applications which vary significantly from the estimate the results were disastrous. For example, roughly 40% of the reachable methods discovered by the points-to analysis for `sablecc.2.8` come from user-defined methods. This difference leads to an average Δ_i value of .45, an unacceptable result. The technique did produce a fairly smooth result, because it was reporting progress for all reachable methods and not just a subset as in technique (2).

5.2 Repeated Analysis

The columns under (4) and (5) in Table 2 display the results from the *repeated analysis* techniques. These techniques rely on historical information being saved during a previous execution of the analysis on the earlier version of the application. In our experiments we considered consecutive releases of the same applications. The historical information saved by the earlier version was used in the analysis of the later version. Since there is no historical information available for the first version of the subject applications, no results were recorded for them.

Total number of reachable methods. Technique (4) uses the total number of reachable methods discovered during the analysis of the preceding version as its estimate for `totalWork`. It monitors progress in the same manner as tech-

	(1)		(2)		(3)		(4)		(5)	
Program	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i
javacup_h	.17	1.26	.10	21.52	.04	1.98	–	–	–	–
javacup_i	.17	1.15	.12	20.46	.04	1.78	.03	1.71	.05	1.54
javacup_j	.15	1.21	.09	19.79	.03	1.82	.03	1.81	.03	1.53
jflex1.3.3	.06	1.75	.08	23.43	.12	2.33	–	–	–	–
jflex1.3.4	.06	1.75	.09	23.57	.13	2.33	.04	1.57	.02	1.36
jflex1.3.5	.07	1.79	.10	21.41	.14	2.36	.04	1.60	.03	1.21
jgraph5.7.4.6	.21	2.54	.22	7.53	.17	1.81	–	–	–	–
jgraph5.7.4.7	.21	2.57	.22	7.59	.17	1.83	.03	1.38	.03	1.18
jgraph5.8	.21	2.50	.21	7.35	.16	1.80	.03	1.32	.03	1.21
jpws.2.0	.26	3.25	.07	13.86	.03	1.34	–	–	–	–
jpws.3.0	.25	3.25	.10	11.12	.12	1.01	.06	1.56	.05	1.36
jpws.3.1	.25	3.25	.10	10.44	.13	.99	.05	1.46	.03	1.26
sablecc.2.8	.11	1.51	.36	5.79	.45	1.60	–	–	–	–
sablecc.3.1	.04	1.88	.23	7.51	.40	1.65	.05	2.07	.06	1.78
sablecc.3.2	.05	1.82	.24	7.21	.40	1.64	.05	1.86	.03	1.52
verbos.1.4	.17	2.28	.24	16.80	.22	2.75	–	–	–	–
verbos.1.5	.18	2.27	.24	16.84	.18	2.31	.05	1.36	.03	1.18
verbos.1.7	.18	2.27	.26	15.82	.20	2.51	.04	1.33	.02	1.14
vietpad.1.2	.26	3.18	.22	24.30	.25	3.04	–	–	–	–
vietpad.1.2.1	.26	3.10	.22	9.11	.25	2.95	.08	1.68	.06	1.44
vietpad.1.3	.27	3.11	.21	20.00	.25	2.85	.08	1.68	.06	1.41
AVERAGE	.17	2.27	.17	17.47	.18	2.01	.04	1.59	.03	1.37

Table 2: Initial analysis: (1) least effort, (2) number of user-defined methods, (3) derived number of total method – Repeated Analysis: (4) total reachable methods (5) methods weighted by number of statements

nique (1) by calling `worked(1)` every time a new method is discovered. Even with this very simple historical information the results show a *marked improvement* over all the heuristics investigated for the fledgling analysis. Imprecision in the technique has two main sources: (A) not accounting for the difference of cost to analyze distinct methods and (B) the total number of reachable methods can change between versions of the application. Since this approach records progress for every method discovered, it produces a fairly smooth result.

Methods weighted by the number of statements. Technique (5) is very similar to technique (4) in that it relies on a single integer value (the number of statements evaluated) being saved from a previous execution of the analysis. However, varying the amount of work being reported with each call to `worked` as measured by the number of statements in each method does produce a more accurate result. Of course, this is only a rough estimation of the total amount of work needed to be performed by the analysis for each method. If in a particular application the ratio of analysis time to the number of statements varies widely from method to method, this technique will become less accurate. Accuracy will again decrease as the difference between the total number of statements processed between the previous version of the application and the current one increases.

6. CONCLUSIONS AND FUTURE WORK

This paper presents work which proposed and evaluated several techniques for estimating the run-time progress of a points-to analysis. For the fledgling analysis, which does not utilize historical information, the accuracy of the techniques were dependent upon the composition of the application under analysis. The experimental results for the fledgling analysis failed to establish one heuristic as being clearly better than the others. The results for the repeated analysis shows that by saving even simple historical information between consecutive runs, a points-to analysis can greatly increase the accuracy of its progress estimates.

In the future we intend to focus on improving the techniques used to estimate progress of static analysis in the absence of historical information. We plan to investigate more sophisticated classification techniques which will better tailor progress estimates to individual applications.

7. REFERENCES

[1] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[2] J. Johnson, editor. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[3] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[4] J. Sawin and A. Rountev. Estimating run-time progress of a call graph construction algorithm. In *IEEE International Workshop on Source Code Analysis and Manipulation*, 2006.

[5] M. Sharp, J. Sawin, and A. Rountev. Building a whole-program type analysis in Eclipse. In *Eclipse Technology Exchange Workshop*, pages 6–10, 2005.

[6] `www.sable.mcgill.ca/soot/eclipse`.

[7] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.