Fast and Precise Application Code Analysis using a Partial Library

Akshay Utture
University of California, Los Angeles
U.S.A.
akshayutture@ucla.edu

Jens Palsberg
University of California, Los Angeles
U.S.A.
palsberg@ucla.edu

ABSTRACT

Long analysis times are a key bottleneck for the widespread adoption of whole-program static analysis tools. Fortunately, however, a user is often only interested in finding errors in the application code, which constitutes a small fraction of the whole program. Current application-focused analysis tools overapproximate the effect of the library and hence reduce the precision of the analysis results. However, empirical studies have shown that users have high expectations on precision and will ignore tool results that don't meet these expectations.

In this paper, we introduce the first tool *QueryMax* that significantly speeds up an application code analysis without dropping any precision. *QueryMax* acts as a pre-processor to an existing analysis tool to select a partial library that is most relevant to the analysis queries in the application code. The selected partial library plus the application is given as input to the existing static analysis tool, with the remaining library pointers treated as the bottom element in the abstract domain. This achieves a significant speedup over a whole-program analysis, at the cost of a few lost errors, and with no loss in precision. We instantiate and run experiments on *QueryMax* for a cast-check analysis and a null-pointer analysis. For a particular configuration, *QueryMax* enables these two analyses to achieve, relative to a whole-program analysis, an average recall of 87%, a precision of 100% and a geometric mean speedup of 10x.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**.

ACM Reference Format:
Akshay Utture and Jens Palsberg. 2022. Fast and Precise Application Code Analysis using a Partial Library. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510046

1 INTRODUCTION

Motivation. Long analysis times are a key bottleneck for the widespread adoption of whole-program static analysis tools. Several recent papers for both Java [3, 10, 15] and C/C++ [8, 22, 23] report that a whole-program analysis on their largest benchmarks can take several hours. Analyzing a large collection of benchmarks like an app-store takes even longer, with a total compute time of many years for the largest app-stores. Hence, a speedup in analysis time can save significant compute time and energy, and enable us to use more precise and expensive algorithms.

Whole-program analyses may be slow, but a user is often only interested in finding errors in the application code [34], which constitutes a small fraction of the whole program. In the NJR-1 dataset [31], application code (excluding third-party libraries) constitutes less than 1% of the whole program on average. Hence, an application-focused analysis has the potential for a large speedup.

Ideally, an application-focused analysis should compute the same set of errors for the application-code as a whole-program analysis. However, this is hard to achieve because errors can both originate in or propagate through the library. We use the singular *library* to refer to the aggregate of the third-party libraries and the standard library. The quality of an application-focused analysis tool's results can be quantified using *precision* and *recall. Precision* is the ratio of true-positives in the tool's results, with the whole-program analysis results serving as the ground-truth. *Recall* is the ratio of whole-program analysis errors caught by the tool. Thus, any application-focused analysis tool can be judged by its performance on the three metrics of precision, recall and speedup.

The current best tool for an application-focused analysis is Averroes [1]. Averroes overapproximates the effect of the library with a compact summary. The overapproximation ensures high recall and the small size of the summary compared to the whole library gives a large speedup. However, this summary is created by merging the analysis information from all the library pointers into a single set, resulting in significantly worse precision than the whole program analysis. In our experiments, Averroes gets an average precision of 59% relative to the whole-program analysis. This precision drop is problematic because empirical studies show that users have a very high bar for precision.

For example, Christakis and Bird [6] find that, in practice, static analysis users care much more about precision than recall. They conclude that practical analysis tools must aim for a minimum of 80% *user-perceived precision.* Failing to meet this value results in users ignoring the tool output entirely. Other empirical studies [4, 13] also arrive at similar conclusions. Whole-program analyses themselves often get much less than 80% *user-perceived precision* [3, 5, 18]. Hence, an application-focused analysis that gets less than 100% precision relative to a whole-program analysis will almost certainly fail to meet the 80% *user-perceived precision* target. This defines the goal of our paper.

Our goal in this paper is to capture the speedup potential of an application-focused analysis, while maintaining 100% precision relative to the whole-program analysis.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510046

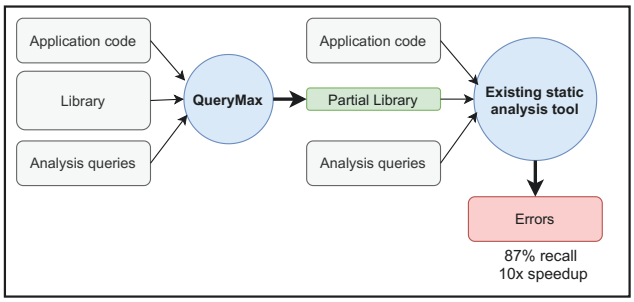

Figure 1: Overview of the QueryMax workflow

Our technique. In this paper, we introduce a new application-focused analysis tool called *QueryMax*, that achieves our goal of 100% precision and gets both good speedup and good recall. Figure 1 gives an overview of the workflow. *QueryMax* acts as a preprocessor to an existing static analysis by selecting a small subset of the library (i.e. partial library) which is relevant to the set of analysis queries in the application. To decide which part of the library is most relevant, *QueryMax* uses a new static analysis called the *external source analysis*. Once *QueryMax* picks the partial library, the existing static analysis tool is run on the application code plus the partial library, with all external library pointers treated as the bottom element in the abstract domain.

The analysis queries used in Figure 1 are exactly like the queries in a demand-driven analysis [28] and they represent all the instructions of interest in the application code. For example, in a cast-check analysis, the analysis queries would be all the down-cast instructions in the application code.

The complexity of *QueryMax* is $O(a^3 + p^2)$ where a is the size of the application-code and p is the size of the (application-code + partial-library). This is much less than the complexity of a whole-program analysis like 0CFA, which has complexity $O(n^3)$ where n is the size of the whole program. Here we assume $(n > p)$ and $(n >> a)$, both of which are true for our benchmarks.

Our experiments focus on Java bytecode programs from the NJR-1 dataset [31], but our approach applies to other object-oriented languages as well. We implemented *QueryMax* in Wala [33] and ran experiments on it with an existing cast-check analysis and null pointer analysis.

Our contributions.

- We introduce a new static analysis, the *external source analysis*, which computes the set of external library pointers affecting each pointer in the application code.
- We describe the *QueryMax* tool which uses the external source analysis and picks a partial library which is small yet sufficient to yield a good recall.
- We show experimentally that *QueryMax* successfully speeds up two different analyses. In a particular configuration, *QueryMax* achieves a 97% recall (on average, relative to a whole-program) and an 8.7x geometric-mean speedup for a cast-check analysis, and a (79% recall, 11.2x speedup) for a null pointer analysis. Both analyses get 100% precision.

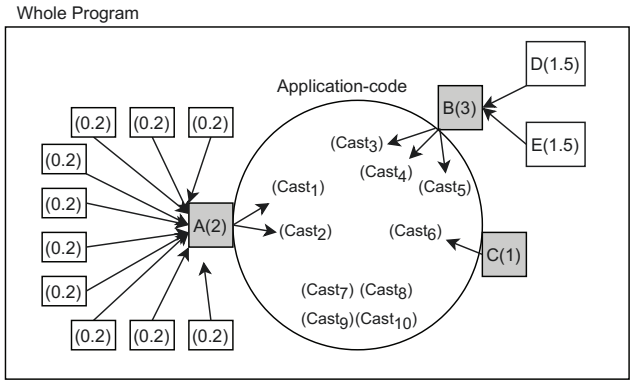

Figure 2: Schematic of a cast-check analysis on application-code

Significance. The impact of this research contribution is that the 10x analysis speedup without any loss in precision will help us meet user expectations on both speedup and precision. Further, the speedup will enable us to use expensive and precise analysis algorithms as well as analyze large programs or large collections of programs (like an app-store) that previously couldn't be analyzed in a reasonable amount of time.

2 EXAMPLE

In this section, we show an example of how *QueryMax* picks a partial library to analyze, and compare this with Averroes' approach. We also discuss two other baselines which can be adapted to provide a speedup over a whole-program analysis: a demand-driven analysis [24, 28] and an application-only analysis.

Figure 2 shows the schematic of a program we wish to analyze for cast-errors. The application code, represented by the circle, is the part in which we wish to catch the cast errors, and everything outside is the library. The grey boxes (labeled A, B, C) on the edge of the circle show library methods with pointers that influence the value of cast instructions in the application code. The accompanying number in the grey box tells us how many cast instructions are affected by that method. The application code has a total of 10 cast instructions and each cast instruction is considered an analysis query. We say that an application-focused analysis *covers* a cast-query if it overapproximates the result of that query. In other words, a query *covered* by a tool is guaranteed to mark it as a cast-error if the whole-program analysis does.

The first baseline technique is to run a demand-driven analysis for every analysis query in the application. The demand-driven analysis exhaustively traces the backward slice of all 10 cast instructions. Casts numbered 7-10 at the bottom of the application circle get their value from inside the application, and hence are answered quickly. The casts affected by B and C (casts numbered 3-6) are also answered quickly because the backward slices have only 2 and 0 caller-methods respectively. However, the demand-driven analysis faces a significant slowdown when answering the two cast queries influenced by A ($Cast_1$ and $Cast_2$). Their backward trace involves the 10 callers of A, each of which could result in a long trail, making

this approach expensive because of these two queries. In total, the demand-driven analysis analyzes all the 15 library methods in the figure. It gets 100% precision and covers all 10 cast instructions since its output is identical to the whole-program analysis. Note that the demand-driven analysis is the only one which requires a new demand-driven design of an existing inter-procedural analysis; the others use the existing interprocedural analysis as is.

The second baseline is an application-only analysis. Such an analysis analyzes the code inside the application circle in isolation and assumes the bottom element of the abstract domain for all library pointers outside. Hence it analyzes zero library methods and only *covers* the 4 casts that get their values from inside the application (that is, the casts numbered 7-10). The application-only analysis gets 100% precision because its errors are the subset of the whole-program errors that do not involve the library.

Averroes [1] improves upon the application-only analysis by modeling the whole library with a small summary. In Figure 2, everything outside the application circle is represented using this summary. The summary primarily consists of a single summary-pointer to represent all library pointers, and a single summary-node to perform all the object initializations and application call-backs. A usual inter-procedural cast-analysis is performed on the application-code plus this summary. Averroes's summary is sound for some analyses, the cast check being one them. Hence, it covers all 10 cast instructions while only analyzing the summary. However, the analysis information merged in the common summary-pointer and summary-node drops precision relative to the whole-program analysis.

QueryMax's approach differs from Averroes primarily in that it selects a small part of the library to fully analyze instead of modeling the library using a summary. *QueryMax* keeps expanding the partial library to be used until it reaches some stopping criterion. Let us assume that we use *QueryMax* with a stopping criterion of 80% *query coverage*. This means that we will have to pick a *fragment* consisting of the application-code plus a partial library, such that at least 8 of the 10 queries (i.e. casts) are *covered* within this fragment.

QueryMax starts out by performing an *external source analysis* on the application code to find out which library pointers affect the 10 cast instructions. This information is marked by the arrows inside the application circle. *QueryMax* then assigns priorities to each external library method based on the number of casts it affects. In Figure 2, this is denoted by the numbers in the grey boxes. Next, *QueryMax* expands on the method with the highest priority (method *B*) to look at its callers, callees and field-reads. Method *B* has 2 callers, *D* and *E*. We estimate that each of *D* and *E* affects half as many casts as B, and hence each of them get half its priority (i.e. 1.5 each). Now, the method with the highest priority is *A*, which on expansion leads to 10 different caller methods, and we assign a priority of (2 / 10) to each of them. The next methods with the highest priority are *D* and *E*, followed by method *C*. Each of these methods are expanded in turn.

At this point, our *fragment* consists of the application code plus a partial library consisting of methods (*A*, *B*, *C*, *D*, *E*). Performing another *external source analysis* on this fragment shows that now 8 of the casts (casts numbered 3-10) are covered within this fragment. Recall that we started *QueryMax* with a stopping criterion of 80% query coverage, or in other words, we would like to terminate when 8 of the 10 casts (i.e. queries) are covered. Hence, *QueryMax* stops expanding at this point, and an existing inter-procedural cast-check analysis is now performed on this *fragment*. By terminating the expansion early, *QueryMax* avoided exploring the 10 callers of method *A*, and their subsequent callers which could potentially expand large sections of the program, while only answering the queries for $Cast_1$ and $Cast_2$. In total, by using *QueryMax*, we analyzed only 5 library methods and covered 8 casts. *QueryMax*, just like an application-only analysis, reports a subset of the whole-program errors, thereby getting 100% precision.

Analysis Tool	Casts covered	Lib Methods analyzed	Precision
Application-only	4	0	100%
QueryMax	8	5	100%
Demand-driven	10	15	100%
Averroes	10	Summary	Low

Figure 3: Number of casts covered, library methods analyzed, and Precision (relative to the whole program analysis) for each of the competing tools

Figure 3 summarizes the number of library methods analyzed (less is better), the cast-instructions covered (more is better), and precision (more is better) for each of the four techniques. *QueryMax*, the demand-driven analysis and the application-only analysis each get 100% precision. For the other two metrics, *QueryMax* obtains a useful trade-off point in between the application-only analysis and the demand-driven analysis. Note that the differences in library methods analyzed is rather small for this example, but the differences are much larger in real programs. Averroes covers all casts and analyzes just the small summary, but gets low precision, thereby falling short of our 100% precision goal.

This example illustrates the core insight underlying *QueryMax's* speedup: few queries in the application code require large sections of the library for their analysis (like $Cast_1$ and $Cast_2$), whereas the remaining queries need a much smaller subset of the library. By identifying these expensive queries and assigning them a low priority, *QueryMax* can pick a small partial library that is sufficient to cover all the remaining queries. The downstream client can now use this partial library in its analysis, which is a fraction of the size of the whole library. The trade-off is that the few expensive queries (like $Cast_1$ and $Cast_2$ in the example) are not fully covered by the partial library, resulting in a few missed errors.

3 APPROACH

In this section, we describe in detail how *QueryMax* works to pick the partial library to analyze.

3.1 Overview

QueryMax picks its partial library by finding the library classes mostly likely relevant to the queries in the application code. *QueryMax* accomplishes this by using a new static analysis called an *external source analysis*. *QueryMax* expands its partial library in a greedy fashion to maximize the number of queries answered in the application code until some stopping criterion is reached. We

No	Stmt	Condition	Constraint
1	x = y	x is not an array	ext(y) ⊆ ext(x)
2	x = y	x is an array	ext(y) ⊆ ext(x) and ext(x) ⊆ ext(y)
3	x = y.f	field f is internal	ext(f) ⊆ ext(x)
4	y.f = x	field f is internal	ext(x) ⊆ ext(f)
5	x = foo(z)	target foo(p){.. ret q} is internal	ext(q) ⊆ ext(x) and ext(z) ⊆ set(p)
6	x = y.f	field f is external	{f} ⊆ ext(x)
7	y.f = x	field f is external	(No constraint)
8	x = foo(z)	target foo(p){.. ret q} is external	{q} ⊆ ext(x)
9	N/A	foo(x) has an external caller y.foo(z)	{z} ⊆ ext(x)

Figure 4: Constraints for the External Source Analysis

discuss two stopping criteria: a *class-budget* if the user wants to set a limit on the number of classes analyzed (proxy for analysis time), and a *query-coverage* if the user wants to set a goal for the number of queries covered (proxy for recall).

3.2 External Source Analysis (ESA)

The external source analysis, or *ESA* for short, takes a program and a subset of its classes called the *fragment*, and computes, for every pointer in the fragment, the set of external pointers that pass values to it. For example, defining the application code as the *fragment* would make the library pointers the external pointers, and an ESA would tell us which library pointers directly pass values to each pointer in the application code. An example of applying the ESA was illustrated in the example in Figure 2, where we computed the library methods affecting cast-instructions in the application code.

The ESA is designed to be context-, flow- and field-insensitive because it's primary application is partial-program analysis, which is time-sensitive. Any overhead of performing an ESA during partial program analysis eats into the speedup that we may get over a whole-program analysis.

Figure 4 outlines the core constraints used for ESA. The second column lists a statement, the third column lists an accompanying condition, and the fourth column gives the corresponding constraint. The third column in the figure uses the words *internal* and *external*. A pointer is considered internal if it is within the fragment, and external otherwise. The abstract domain for the ESA consists of all possible subsets of external pointers. Hence, the notation $ext(y)$ in the fourth column represents the set of external pointers passing values to the fragment pointer y. This is different from the notation $\{z\}$ which is a singleton set consisting of the external pointer z.

Rows 1-5 in Figure 4 are identical to a standard context, flow and field-insensitive pointer analysis such as [29], and we assume that the reader understands them well. Rows 6-9 deal with the different types of external pointers: external fields, external return values, and external function-arguments. The constraints for these rows are similar to what one would expect for a *new* statement in a pointer analysis. Row 6 says that for the read of an external field f, the external field f should be added to the *ext* set of the assigned variable x. Row 7 says that writes to external fields produce no constraint. Row 8 says that for every external target of a method call, the return pointer of the target should be added to the *ext* set of the assigned variable x. There are no constraints for the arguments in this case. Row 9 says that if a method in the fragment has a caller outside the fragment, then the external caller's argument should be added to the *ext* set of the method's parameter.

The generated constraints can be solved using standard static-analysis constraint solving techniques. The complexity of solving the ESA constraints on a fragment of size p is $O(p^3)$. The complexity calculations are very similar to that of a context-insensitive pointer analysis.

In addition to the ESA, we define a faster version of it called the *fast-ESA*, with the primary change being to the abstract domain. Instead of maintaining the set of external sources for every fragment pointer, *fast-ESA* only maintains whether or not the set is non-empty. Hence there are only two elements in the *fast-ESA* abstract-domain: the top element is used when the fragment pointer may be passed a value by an external source, and the bottom element is used when the pointer is guaranteed to not get any values from external sources. The constraints are the same as in Figure 4, except for Rows 6-9 using the Top element instead of the external pointer names. Due to the smaller size of the abstract domain, the complexity of *fast-ESA* on a fragment of size p is $O(p^2)$, which is lesser than the cubic complexity of ESA. Hence, *fast-ESA* allows us to compute whether a fragment pointer is affected by external sources much quicker than an ESA.

3.3 QueryMax Algorithm

The *QueryMax* algorithm is used to pick a *fragment* to analyze, consisting of the application and the partial library, with a best effort to catch as many of the whole-program errors as possible. The example in Section 2 showed how *QueryMax* runs for one particular case. Here, we describe the algorithm (given in Figure. 5) in detail. The figure has three main procedures: the main algorithm, the class-budget stopping criterion and the query-coverage stopping criterion.

The main algorithm (line 1) takes as input the application classes, set of all classes, and the queries to be answered. For internal book-keeping, *QueryMax* uses the set $fragment$ to mark the classes that are to be analyzed finally, a $visited$ set for the methods, and a priority-queue *pQueue* to keep track of the priorities of the external (library) methods to be explored. The intuition behind the priority values is that they represent the estimated number of queries answered by that method, and *QueryMax* will explore methods with a higher priority earlier.

The main algorithm starts off by performing an ESA (line 5), with the application classes as the $fragment$. The ESA computes the set of external library pointers affecting each pointer in the application classes. Using the ESA result, we compute its inverse information: the number of queries affected by each of the external library pointers (line 6). Now, the method of each of the external library pointers is added to *pQueue* with a priority equal to the number of queries it affects. For external field pointers, we add the methods which write to that field. Each of the external library pointers' methods are added to the $visited$ set. After this initialization phase, we move into the main algorithm loop.

```
1:  procedure QueryMax(appClasses, allClasses, queries)
2:      fragment ← appClasses
3:      visited ← new Set()
4:      pQueue ← new PriorityQueue()
5:      esa ← ESA(allClasses, appClasses)
6:      extLibPtrs ← computeAffectedQueries(esa, queries)
7:      for ExternalLibraryPointer e in extLibPtrs do
8:          pQueue.setPriority(e.method, e.affectedQueries)
9:          visited.add(e.method)
10:     end for
11:     while not (pQueue.empty() ∨ Criterion) do
12:         Method m ← pQueue.poll()
13:         analysisFragment.add(m.declaringClass)
14:         methodSlice ← getmethodSlice(m)
15:         newPriority ← m.priority / methodSlice.size
16:         for Method n in methodSlice do
17:             if visited.contains(n) then
18:                 pQueue.addToOldPriority(n, newPriority)
19:             else
20:                 pQueue.setPriority(n, newPriority)
21:                 visited.add(e)
22:             end if
23:         end for
24:     end while
25:     return fragment
26: end procedure
27:
28: procedure BudgetCriterion(fragment)
29:     percentAnalyzed ← (fragment.size / allClasses.size)
30:     return (percentAnalyzed ≥ budget)
31: end procedure
32:
33: procedure CoverageCriterion(fragment, queries)
34:     coveredQueries ← fastESA(allClasses, fragment, queries)
35:     coverageRatio ← coveredQueries / fragment.totalQueries
36:     return (coverageRatio ≥ goal)
37: end procedure
```

Figure 5: QueryMax algorithm

The main algorithm loop starts at line 11. It keeps looping until either *pQueue* is empty or we satisfy the stopping criterion (described below). Inside the loop, we remove the method m with the maximum priority in *pQueue*, and add its class to the $fragment$. This step is a greedy move to expand the class that is expected to affect the largest number of queries. The next step is to find the *method-slice* of m (line 14). This is similar to computing one step in the backward slice of a pointer, but is performed at the granularity of methods instead of pointers to reduce the overhead. The *method-slice* consists of callers and callees of m, as well as methods which write to fields that are read in m. Each method in the method-slice gets a new priority which is the priority of m divided by the size of its method-slice. The intuition behind this priority assignment is that if m affects k queries and has t callers/callees, then each caller/callee is expected to affect k/t queries. If a method from the method-slice is already in *pQueue* we add the new priority to its old priority, else we add the method to *pQueue* with the new priority. Finally, once the loop has terminated, the $fragment$, which has the set of classes to be analyzed, is returned. An existing inter-procedural static analysis is performed on the set of classes returned, with all external pointers assumed to be the bottom element.

QueryMax uses a stopping criterion to know when to stop expanding the fragment and return, and we experiment with two such criteria: *class-budget* and *query-coverage goal*.

Class budget. The class budget stopping criterion (line 28) is used when the user wants a handle on the analysis time. The class budget is a proxy for a time budget, and we prefer to use the number of classes instead of analysis time because it can be accurately computed in advance without running the actual analysis. This criterion simply checks if the percentage of classes used in the fragment is greater than a certain budget. The budget is assumed to be specified as a global variable for readability. For this paper, we experiment with a 3%, 10% and 30% class-budget. A budget of under 2% will have no space for library methods in some programs, and a budget of over 40% will analyze a large partial library, resulting in only a small speedup.

Query-coverage goal. The query-coverage criterion (line 33) is used when the user wants a handle on the recall. Query-coverage is a proxy for recall, because the number of errors found is expected to be proportional to the number of queries covered. The query-coverage criterion uses a *fast-ESA* (line 34) to find the number of queries covered by the fragment classes, and computes a *coverage-Ratio* which is the percentage of queries covered. Finally, if the *coverage-Ratio* exceeds the query-coverage goal, then we return true. The goal is assumed to be specified as a global variable for readability. The coverage criterion is not used at every iteration of the main loop because the *fast-ESA* adds significant overhead. Instead, we only evaluate this criterion at some set checkpoints. For this paper, we experiment with 70% and 90% query-coverage goals. A goal of less than 60% gives recall close to that of a application-only analysis, and a goal of greater than 95% requires too many classes to be added to the partial library, thereby resulting in too small a speedup.

The overall complexity for *QueryMax* is $O(a^3 + p^2)$ where a is the size of the application-code and p is the size of the (application-code + partial-library). The $O(a^3)$ term comes from the ESA performed on the application-code on line 5, and the $O(p^2)$ term comes from the *fast-ESA* performed for the coverage-criterion on line 34.

3.4 Applicability of QueryMax to Client Static Analyses

Now that we understand how *QueryMax* works as a preprocessor to select a partial library, we can discuss what kind of client analyses *QueryMax* can be applied to.

Firstly, since *QueryMax* trades off recall for analysis speedup, its client analysis should be able to afford to lose some recall. For example, compiler optimization clients that prefer the static analysis be sound (or soundy [16]), will not use *QueryMax*. Secondly, *QueryMax* is restricted to client analyses that only care about errors

Client Analysis	Analysis Queries
Cast-check analysis [29]	Cast instructions
Null-pointer analysis [12]	Method calls and field accesses
Taint Analysis [17]	Taint sink instructions
Type-state analysis [9]	State-change instructions
Pointer analysis [14]	Client analysis queries

Figure 6: Analaysis Queries for different Client Analyses

manifesting in the application code. It cannot speed up a client analysis that aims to catch errors manifesting in both the application code and the library.

On the plus side, *QueryMax* makes no assumptions about the flow-, context- and field-sensitivity of the client analysis that it is preprocessing for. Hence it can be applied regardless of the client analysis' sensitivities. Further, unlike [1], it makes no assumptions about the demarkation between application and library code. Hence, the user can choose any subset of classes as the application code to focus on and get everything outside the subset treated as the library.

Figure 6 lists some analysis clients that *QueryMax* could be applied to and shows the corresponding analysis queries for such a client analysis. This is not an exhaustive list of client analyses, and its main purpose is to give examples of what the analysis queries would be for different kinds of client analyses. Typically, an analysis query would be any instruction in the application code where a particular kind of error could potentially manifest. For example, for a cast-check analysis the queries are cast instructions. For a null-pointer analysis they are all dereference instructions, including method calls and field accesses. For a taint-analysis which is defined in terms of vulnerable source-sink pairs, the analysis queries would be all the sinks. For a type-state analysis, like one that checks for the correctness of file-operations, all the state-change operations (like file-open, file-close, etc.) will be the analysis queries. A pointer analysis itself does not have any statements or variables of interest, and hence cannot define analysis queries for itself. However, if the pointer analysis is used by a particular client (like cast-check or taint analysis), we can define its analysis queries as the queries of that client.

4 IMPLEMENTATION

The WALA [33] framework for Java bytecode analysis is used to implement *QueryMax* and the ESA analysis. The actual analysis is performed on the WALA IR, which is in SSA form and hence automatically grants partial flow-sensitivity. We use the CHA-callgraph for all the analyses, since computing a whole-program 0-CFA callgraph would defeat the purpose of doing a partial library analysis. We ignore call-graph edges involving a single call-site with more than 10 targets, since the likely root cause of this is severe imprecision, and it results in mostly false-positives. We also exclude the *java/util* package since it is well known for introducing too many false-positives unless one uses high context-sensitivity [30].

Client Analyses. *QueryMax* accepts any inter-procedural analysis to run with as long as the analysis can be run on a subset of the classes in the program. We experiment with two such analyses: a cast-check analysis and a null-pointer analysis. The cast-check analysis is based on the VTA algorithm [29] for pointer analysis. The null-pointer analysis (based on [12]), focuses on catching null-pointer exceptions resulting from uninitialized instance fields. The two analyses vary significantly in their constraints, abstract domains, design decisions, number of analysis queries, and number of errors per program. Hence, the two analyses offer considerable diversity for experimentation. We leave to future work to experiment with other client analysis, including other implementations of cast-check and null-pointer analysis, such as NullAway [2].

For the analysis sensitivities, we choose to be context-, flow- and field-insensitive as far as possible. The cast-check analysis is insensitive on all three axes. The null-pointer analysis is context- and field-insensitive but flow-sensitive because a flow-insensitive version of the analysis trivially marks all fields as null. Our choice of sensitivities are different from other papers such as [24–26], because their task is to improve precision, whereas ours is to improve analysis speed. For the task of improving precision, a flow-, context- and field-sensitive analysis is the hardest baseline because it is the most precise. In contrast, for our task of improving analysis speed, a context-, field- and flow-insensitive analysis is the hardest baseline because it is the fastest.

Demand-driven analysis. We choose to write our own demand-driven cast-check instead of using an existing tool like [24] or [28]. This ensures that the whole-program analysis and demand-driven analysis are identical in their various sensitivities, analysis design decisions, constraint solvers and errors generated. This normalization helps to make a fair timing comparison between the demand-driven analysis, and other techniques like *QueryMax*, Averroes and the application-only analysis. For the demand-driven cast check, we implement caching across queries to reuse computations done for a previous query.

Most prior research on demand-driven analysis deals with pointer analysis which can be used to implement the cast-check. However, a design of the demand-driven version of the null-pointer analysis [12] is not publicly available and is non-trivial to design from scratch. Hence, for the demand-driven analysis, we only report experiments for the cast-check analysis.

Averroes. Averroes takes as input the original Jar file and the set of application classes, and produces modified Jar files consisting of the application classes and the library summary. We do not count the time taken to produce the modified Jar files since it is a one-time cost which is amortized across all client analyses. The Averroes library summary also has the *java/util* package excluded from it. Finally, the same null-pointer and cast-check analyses described above are run on the modified Jar files, thereby making a fair comparison between Averroes and the other techniques.

Reflection. We do not use WALA's inbuilt reflection support for the client analyses because this would worsen the analysis time of the baseline, thereby making *QueryMax* look better. Further, we also do not use reflection support for the ESA. While reflection support may help the ESA find external sources reachable through reflection, its overhead is too high and this reduces the effective speedup provided by *QueryMax*.

Statistic	Mean	Std-dev
Lines of application code	9911	12689
Number of application classes	97	91
Number of 3rd party library classes	2608	5220
Percentage of application classes	0.33%	0.33%

Figure 7: Statistics about the benchmark programs

Statistic	Cast-check	Null-pointer
Total number of programs	221	221
Mean Errors per program	4.4	37
Std-dev Errors per program	27	56
Programs with non-zero errors	58	177
Mean Analysis time	27 sec	293 sec
Std-dev Analysis time	41 sec	142 sec

Figure 8: Statistics about the whole-program cast-check and null-pointer analysis on the benchmark set

Precision, Recall and Speedup. To measure the quality of an analysis using *QueryMax* or any of the baseline techniques like Averroes, demand-driven analysis, etc., we evaluate it on the three axes of speedup, precision, and recall. Here are the standard formulae for computing these metrics:

$$Speedup = \frac{Whole\text{-}program\ analysis\ time}{Application\text{-}focused\ analysis\ time}$$

$$Precision = \frac{|A \cap W|}{|A|} \qquad Recall = \frac{|A \cap W|}{|W|}$$

where A is the set of errors given by *QueryMax* and W is the set of errors given by the whole-program analysis (which we consider as the ground-truth).

5 DATASET DESCRIPTION

We use the NJR-1 dataset (available here [31]), as our benchmark-set. We chose NJR-1 because its 293 Java bytecode programs run successfully with WALA, and each program explicitly lists its set of application and third-party library classes. Out of the 293 programs we remove 68 programs that crash the Averroes tool. The crash reports have been filed with the developers. Another 4 programs which run out of memory for the whole-program null-pointer analysis are removed, leaving us with a total of 221 programs.

Figure 7 lists some statistics about the benchmark programs. On average, each benchmark program has almost 10k lines of Java source code in the application, with an average of almost 100 classes each. The third-party library classes are much larger, with an average of 2608 classes per benchmark, and these correspond to an estimated 250,000 lines of Java source code. The application classes constitute just 0.33% of the program, with the remaining being the Java standard library and third party library classes. The large standard deviation for all these metrics implies that they vary significantly across benchmarks. Among the 221 benchmarks, 63 use reflection in the application code and 130 use reflection in the third-party libraries.

Figure 8 lists some statistics about the benchmarks when analyzed with a whole-program null-pointer analysis and the cast-check analysis. The cast check analysis gets 4.4 errors per program on average, whereas the null pointer analysis gets 37. This large difference is expected, since down-casting is rare, whereas method calls and field accesses are common.

The table also shows that only 58 of the 221 programs have non-zero cast errors and only 177 of them have non-zero null-pointer errors. The programs with zero errors in the whole program analysis are a problem for the evaluation because their recall is undefined for all of the techniques. Hence, the experimental results are reported in two parts: those with zero errors and those with non-zero errors. We report the recall and speedup for the non-zero error cases and only speedup for the zero error cases.

The analysis times for the two analyses also vary widely, with the cast-check taking 27 seconds per program and the null-pointer analysis taking 293 seconds per program. The standard deviation for analysis times is large, especially for the cast-check analysis, implying that a few outliers have large analysis times.

6 EXPERIMENTAL RESULTS

In this section, we discuss our experimental results which validate the following claims.

(1) **C1:** *QueryMax* gets a significant speedup, full precision and reasonable recall as compared to the whole-program analysis, with trade-off points that none of the existing techniques can achieve.
(2) **C2:** The distribution of speedups and recall-scores are uniform across the benchmarks.

The experiments were carried out on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 GB RAM. For the JVM, the default heap size of 32GB, and default stack size of 1MB, was used. The artifact for the paper is available here [32].

The first two sub-sections validate the claims made, and these experiments focus on the programs with non-zero errors. The third subsection evaluates the programs with zero errors, the fourth examines the *QueryMax* analysis time split-up, the fifth compares the correlation between class-budget and analysis time, and the sixth subsection outlines the threats to validity.

6.1 C1: Main Result

Figures 9 and 10 show the various recall and speedup trade-off points for the cast-check analysis and null-pointer analysis respectively. The X-axis gives the recall plotted on a linear scale and the Y-axis gives the speedup plotted on a logarithmic scale. There is actually a third axis for precision, but we do not show it because all the techniques except for Averroes, get a 100% precision. We mark Averroes' precision directly in the figure.

Whole-program analysis. The whole-program analysis (marked by the black circle) is considered as the ground-truth and the reference for all speedup calculations. Hence it trivially gets 100% recall and 1x speedup.

Demand-driven analysis. The demand-driven analysis (marked by the green triangle) computes the same result as a whole-program analysis and hence gets 100% recall, but it manages a 5.1x geometric

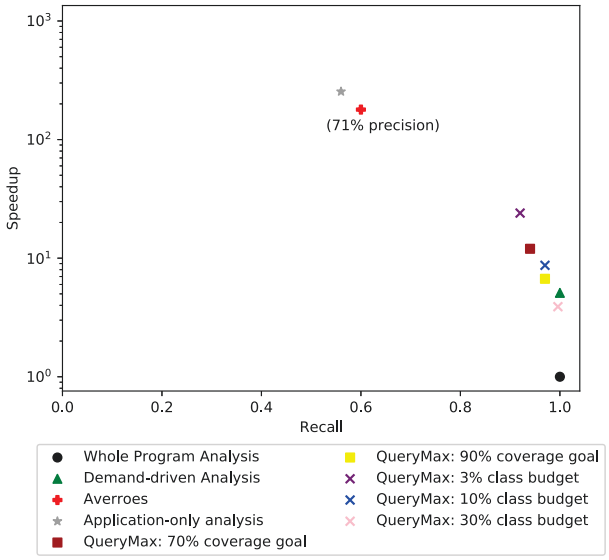

Figure 9: Recall and Speedup for the various techniques for the cast-check analysis

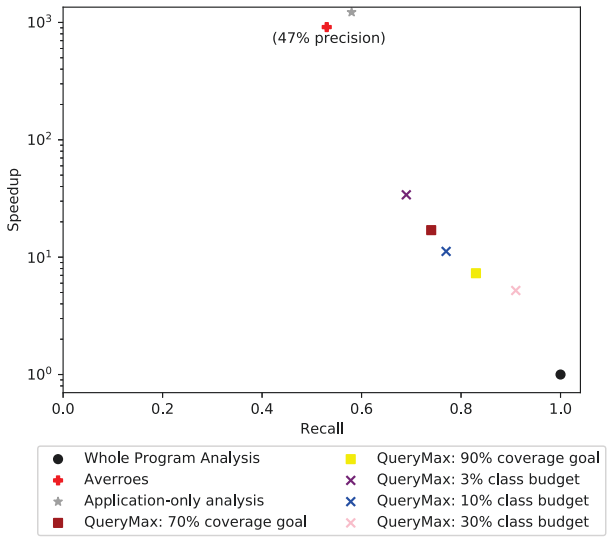

Figure 10: Recall and Speedup for the various techniques for the null-pointer analysis

mean speedup for the cast-check analysis because it avoids analyzing the whole program. This mean speedup is not representative of the average benchmark. One portion of the benchmarks get a large speedup because they analyze a small part of the program, while others experience a slowdown because they analyze a large section of the program and the demand-driven analysis adds some overhead. The reason for this difference in speedups is that some programs either have expensive queries like the example in Section 2, or a larger number of queries, and others don't. This observation is in line with previous experiments on demand-driven analyses [11]. A demand-driven version of the null-pointer analysis does not exist (see why in Section 4), but we expect it to perform worse than in the cast-check analysis because there are significantly more queries in the null-pointer analysis and the demand-driven analysis works on a per-query basis.

Application-only analysis. At the other end of the spectrum is the application-only analysis (marked by a grey star), which is orders of magnitude faster, but gets a significantly lower recall. For the cast-check analysis it gets a 254x speedup and a 56% recall, whereas for the null-pointer analysis it gets 1222x speedup and 58% recall. The large speed-up is attributed to the fact that the application constitutes only 0.33% of the whole program on average (Figure 7). An application-only analysis is a good option for use-cases where analysis speed is significantly more important than recall, but when both are important, it doesn't strike as good of a balance between the two.

Averroes. The point closest to this is Averroes (marked by a red plus), which gets a (179x speedup, 60% recall, 71% precision) for the cast check analysis, and a (913x speedup, 53% recall, 47% precision) for the null-pointer analysis. This is the only tool for which we report the precision because the other tools get 100% precision.

The massive speedup of Averroes is attributed to the fact that its summary is tiny compared to the size of the library. However, the tiny size is also what causes analysis information to be merged and precision to drop. The 47% and 71% precision values are significantly lower than our target of 100% precision.

Averroes should theoretically get 100% recall for the cast-check, but not for the null-pointer analysis because its library summary includes information about object-initialization but not about field-initialization. The observed recall is lower than expected because of a bug in its dealing of inner-classes which causes any error propagating through a Java inner-class to be dropped. The bug has been reported to the developers.

QueryMax. Finally, *QueryMax* gives some points in between these two extremes. The points marked with crosses are for the class-budgets and the points marked with with squares are for the query-coverage goals.

For the cast-check analysis (Figure 9) *QueryMax* performs very well. The 3% budget (purple cross) gets a 24x speedup and 92% recall, and this strikes a really useful balance between the two metrics. The 10% budget (blue cross) gets an 8.7x speedup and a 97% recall, thereby favoring the recall a little more than the speedup, but still a great trade-off between the two metrics. The 30% budget (pink cross) gets 3.9x speedup and a 99.6% recall.

The query-coverage stopping criterion (represented by the squares) for the cast-check analysis gets similarly good results. The 70% goal (brown square) gets (12x speedup, 94% recall) and the 90% goal (yellow square) gets (6.7x speedup, 97% recall). The speedups for the coverage goals are slightly lower than the class budgets. For example, the yellow square in Figure 9 is directly below the blue cross. This happens because calculating the query-coverage involves the overhead of at least one *fast-ESA*, which the class-budget version avoids. However, the coverage-goal gives a guarantee on

the number of queries covered, which could be more valuable than a guarantee on the number of classes analyzed.

For the null-pointer analysis (Figure 10), we see a similar speedup vs recall trade-off for *QueryMax*. The 3% class-budget, marked by the purple cross gets (34x speedup, 69% recall), the 10% class budget marked by the blue cross gets (11x speedup, 77% recall), and the 30% class-budget, marked by the pink cross gets (5.2x speedup, 91% recall). The query-coverage points (marked by squares) lie in between these three points. Unlike the cast-check analysis, the coverage-goal variants are not much worse than the class-budget variants for the null pointer analysis. We discuss the reason for this observation in Section 6.4

Comparing figures 9 and 10 shows that *QueryMax* gets much better recall for the cast-check than the null-pointer analysis. The main reason for this is that some dereference instructions get a high-priority from *QueryMax*, but are often never null-pointer exceptions. For example, in any given program, the *println()* call occurs many times, and in all cases gets its value from the field *java/lang/System.out*. Since this field affects several dereference instructions, it ends up getting a high-priority and that part of the library gets added to our partial library first, even though the *println()* calls never cause null-pointer exceptions. A similar case happens to some other common dereference instructions.

To sum up, *QueryMax* with either stopping criterion provides a useful analysis design point in-between the application-only analysis and the demand-driven analysis, just like in the example from Section 2. Further, unlike Averroes, it achieves this speedup without sacrificing precision, and thus continues to meet the high-precision expectation of its users.

6.2 C2: Distribution of Recall and Speedup

We now understand the recall and speedup trade-off points for *QueryMax*, but we would also like to know their distribution across the benchmark programs. Figures 11 and 12 use a histogram to show the distribution of the recall and speedup for *QueryMax* with a 70% query coverage. The X-axis gives the speedup or recall, with the values split into bins, and the Y-axis gives the number of programs in each bin. Just like figures 9 and 10, we use a logarithmic scaling for speedup here. The recall is still plotted on a linear scale.

The recall for *QueryMax* with the cast-check analysis (Figure 11) is close to 100% for most of the programs, with only a couple of programs getting lower scores. Two programs get a 0 recall. These programs had just 1 and 2 errors each and missing those errors meant a recall of 0. The null-pointer analysis (Figure 12) has a similar story for recall, but it has a larger number of programs with 0 recall. In most of these cases, the null-errors are very few and highly related, and hence missing one library method could cause all the null-errors to be missed.

The speedups for both analyses are consistent, with most programs getting close to the mean speedup value. The cast-check has 2 programs that get less than a 1x speedup. This happens because if *QueryMax* cannot guarantee that 70% coverage has been reached by the time its chosen fragment expands to 30% of the program, it simply falls back to picking the whole program, thereby resulting in no speedup.

6.3 Zero-Error Benchmarks

The results so far focused on the programs with non-zero errors. Figure 13 lists the speedup for programs with zero errors in the whole-program analysis. The speedups for *QueryMax* are on average twice as much as the non-zero error benchmarks. The demand-driven cast-check however, gets a 42x speedup here as compared to the 5.1x speedup on the non-zero error benchmarks. This high speedup for the demand-driven analysis on these benchmarks stems from the fact that these programs have much fewer down-cast instructions than the non-zero error benchmarks. Thus, when there are very few analysis queries, a demand-driven analysis gets a higher speedup.

6.4 Split-up of Analysis Time

Recall the workflow of *QueryMax* from Figure 1. We first run *QueryMax* with either a query-coverage goal or a class-budget. For query-coverage, *QueryMax* includes the additional overhead of the *fast-ESA*. Finally, we run the existing analysis. Figure 14 gives a split-up of the time between *QueryMax* (minus the *fast-ESA*), the *fast-ESA*, and the existing static analysis, for the query coverage goal.

For the cast-check, the *fast-ESA* takes 51% of the time, whereas the other *QueryMax* part takes just 4%. This explains why the query-coverage criterion from Figure. 9 is slower than the class-budget one; computing the query-coverage needs the *fast-ESA*, but computing the class-budget does not.

For the null-pointer analysis, both the *fast-ESA* and the other part of *QueryMax* take up a small percentage of the time (8% totally). The contribution of *QueryMax* and *fast-ESA* to the total analysis time is larger for the cast-check than the null-pointer analysis. The reason for this is that existing null-pointer analysis has a longer absolute analysis time than the cast-check, but the absolute *fast-ESA* time is similar in both cases.

6.5 Analysis-time vs Number of Classes

As a minor result, we show the relationship between the class-budget and the analysis time, to justify our use of the former as a proxy for the latter. Figure 15 compares the number of classes analyzed on the X-axis with the analysis time on the Y-axis for both analyses. Each point represents one analysis of *QueryMax* with a class-budget. For both analyses, the analysis time is almost linear, but the cast check has more outliers, which explains the high-standard deviation for its analysis time (see Figure 8). The figure also plots a regression line, and the equation of this line can be used to convert time-budgets into class-budgets.

6.6 Threats to Validity

There are two main threats to validity. The first is that out of the application, third party libraries and standard library, the standard library forms the largest part. Even though different programs interact with different parts of the standard library, it still means that the benchmarks are not perfectly independent for a static analysis. However, this issue occurs with any static-analysis benchmark-set where the programs access the standard library.

The second is that analysis time measurements for all the programs were performed using a single run, even though execution times can vary across runs. However, since the speedups are large

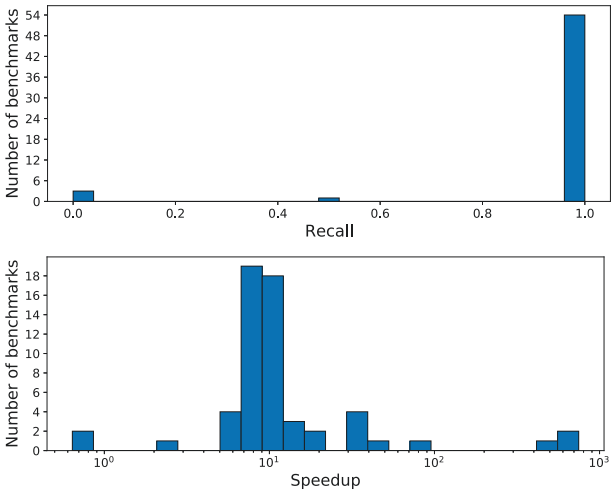

Figure 11: Speedup and Recall histograms for *QueryMax* (70% query coverage) on the cast-check analysis

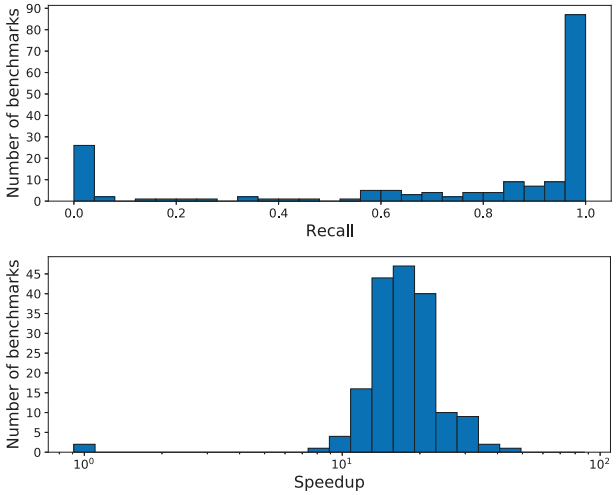

Figure 12: Speedup and Recall histograms for *QueryMax* (70% query coverage) on the null-pointer analysis

Analysis	Cast-check	Null-pointer
Application-only	395x	2196x
Averroes	230x	1744x
QueryMax 3% class-budget	30x	84x
QueryMax 10% class-budget	13x	33x
QueryMax 30% class-budget	6.4x	18x
QueryMax 70% query coverage	16x	20x
QueryMax 90% query coverage	12x	10x
Demand-driven	42x	N/A

Figure 13: Speedup for the various analysis techniques for the Zero-error benchmarks

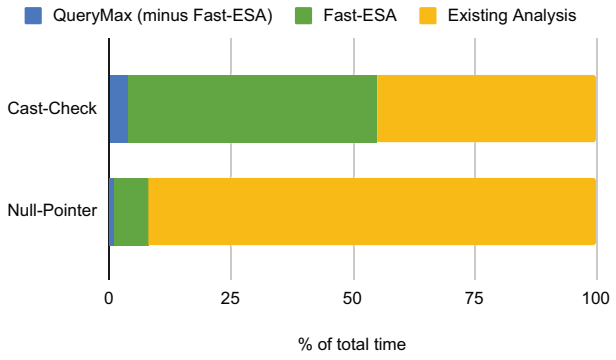

Figure 14: Split up of the time taken by each component for an analysis using *QueryMax* with the query-coverage goal

(an order of magnitude) and the benchmarks are numerous, these variations matter less. Further, since the total experiment-time is already ten days, performing multiple runs is infeasible.

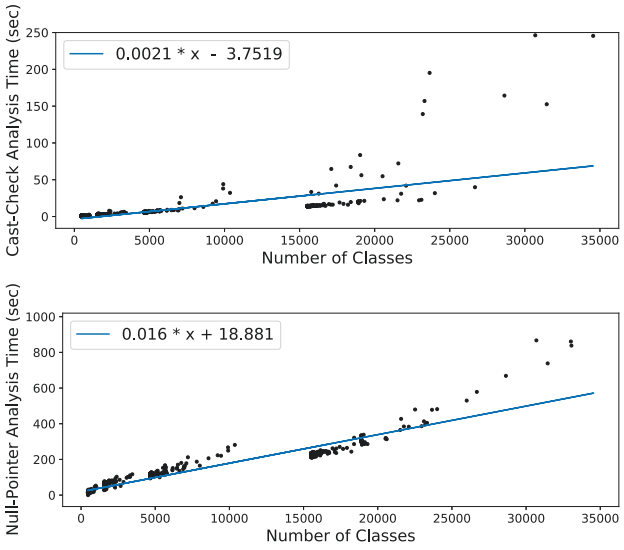

Figure 15: Class-budget and analysis time relationship.

7 RELATED WORK

The three research directions that focus on speeding up static analysis by avoiding the analysis of the entire program are library-summary based analysis, demand-driven analysis, and the analysis of program fragments. We discuss each of these in turn.

Library-summary based analysis. The main idea behind the research in this area is to create an analysis summary for the library and use this library summary instead of the actual library code to analyze the application.

Averroes [1] heavily compresses the library into a small summary. This summary consists of a single summary-pointer to represent all library pointers, stubs for methods called directly from the application, and a single summary-method to perform all the object initializations and application call-backs. Since this summary is quite small compared to the library, using it in place of the library results in a massive speedup. However, the small size of the summary has two downsides: precision drops because information is merged in the single summary-pointer, and some kinds of information (like field initialization information for the null-pointer analysis) get left out out of the summary. *QueryMax*, in contrast, leaves out no information in the partial library that it chooses, and more importantly, preserves the precision.

The *component-level analysis* by Rountev et. al [19, 21] differs from Averroes in that its library summary contains all the information necessary to get the same result as a whole program analysis. The first time an analysis is run, the library is separately analyzed and summarized, and the summary is integrated with the application analysis. This saves no time in the initial run (the overhead causes a slowdown). However, it saves time in subsequent runs when the same library summary is reused across different programs or future versions of the program. *QueryMax* on the other hand never uses the whole library and it speeds up the analysis of each program independently. Further, unlike the component-level analysis which needs a separate design for each type of analysis, *QueryMax* can be used off-the-shelf with any analysis.

Demand-driven analysis. Demand-driven analyses [11, 24, 27, 28] are well-accepted as the most efficient option for single analysis queries, and work best for resource-constrained environments like IDEs and JIT compilers. They also perform well when the number of queries is small [28]. However, when analyzing entire applications in which the number of queries is large, the demand-driven analysis could end up analyzing large parts of the program and cause a slowdown because of their overhead [11]. We also see this observation in our benchmarks, where some programs get huge speedups over a whole-program analysis, but some experience slowdowns.

Unlike the demand-driven approach, *QueryMax* avoids expensive queries by assigning them a low priority, like in the example from Section 2. It also avoids the demand-driven overhead since it still runs a batch analysis, thereby performing better when there are many queries to be answered. Further, since *QueryMax* is only a preprocessor to an existing whole-program analysis, it can be used with an existing analysis, without requiring a design of a demand-driven version of it.

Analysis of Program Fragments. There has been past research on analyzing program fragments in isolation. In our use-case, the program fragment is the application-code. Cousot and Cousot [7] describe four techniques for this general approach. The first is a simplification-based separate analysis, which analyzes the various fragments of a program separately and then combines their information. This idea is similar to the library-summary based analysis by [19], and has the drawbacks as discussed above. The second technique is a worst-case analysis, which means running an application-only analysis, but using the top element of the abstract domain for library pointers. This introduces additional false-positives. Our experiments on this technique show that it gets a precision (averaged over both analyses) of 22% which is far below our 100% target precision. The third technique is to ask a user to provide stubs for the library (i.e. information about the library interface) and then perform an application-only analysis that incorporates these stubs instead of the library. This can give high recall, precision and speedup, but it requires a static-analysis expert to manually write and update the stubs for each library. The fourth technique uses a relational abstract domain and analyzes a program fragment by giving symbolic names to external pointers and lazily evaluating the values they pass. To the best of our knowledge, there are no recent implementations or experimental results to compare the effectiveness of this technique in practice.

Rountev et. al [20] introduce a technique to improve the performance of a whole-program flow-sensitive analysis. They perform a flow-sensitive analysis for the application code and then use a whole-program flow-insensitive analysis to overapproximate the effect of the library pointers. The two limitations of this technique are that it drops precision as compared to the original flow-sensitive analysis, and it cannot be used to speed up a flow-insensitive analysis. *QueryMax* on the other hand maintains the same precision as the original analysis tool and works with any level of context-, flow- or field-sensitivity.

8 CONCLUSION AND FUTURE WORK

In this paper, we introduce a new application-focused analysis tool *QueryMax*, which achieves a large speedup over a whole-program analysis, without losing any precision. *QueryMax* acts as a preprocessor to an existing static analysis to select a partial library that is small but sufficient to answer most of the analysis queries. *QueryMax* provides the user with two stopping criteria: a class-budget or a query-coverage goal, depending on whether the user wants a handle on the analysis time or the recall. Our experiments on the NJR-1 dataset show that *QueryMax* provides a significant speedup at the cost of a small and controlled drop in recall, and with no loss in precision.

A possible future research direction could be to evaluate *QueryMax* and the other baseline techniques with other client analyses such as taint analysis or type-state analysis. Additionally, one could also extend the approach to the Android platform, with the help of frameworks such as WALA that support Android analysis. Finally, a third direction could be to study how the *QueryMax* approach translates to benchmarks in other popular languages such as C/C++ and Javascript.

ACKNOWLEDGMENTS

This work was supported by the U.S. NSF Award 1823360, and the ONR Award N00014-18-1-2037. We also thank the ICSE'22 reviewers and Aishwarya Sivaraman for their constructive comments that helped improve the paper.

REFERENCES

[1] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-Program Analysis without the Whole Program. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 378–400. https://doi.org/10.1007/978-3-642-39038-8_16

[2] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-Based Null Safety for Java. In *Proceedings of the 2019 27th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Tallinn, Estonia) *(ESEC/FSE 2019)* Association for Computing Machinery, New York, NY, USA, 740–750. https://doi.org/10.1145/3338906.3338919

[3] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges, Eric Bodden, and Andreas Zeller. 2020. Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1061–1072. https://doi.org/10.1145/3377811.3380438

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[5] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2491956.2462186

[6] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[7] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)* Springer-Verlag, Berlin, Heidelberg, 159–178.

[8] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 72–82. https://doi.org/10.1109/ICSE.2019.00025

[9] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (may 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

[10] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/3213846.3213860

[11] Nevin Heintze and Olivier Tardieu. 2001. Demand-Driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

[12] Laurent Hubert, Thomas Jensen, and David Pichardie. 2008. Semantic Foundations and Inference of Non-Null Annotations. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems* (Oslo, Norway) *(FMOODS '08)*. Springer-Verlag, Berlin, Heidelberg, 132–149. https://doi.org/10.1007/978-3-540-68863-1_9

[13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.

[14] Ondrej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using SPARK. *International Conference on Compiler Construction* 2622, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[15] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[16] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (jan 2015), 44–46. https://doi.org/10.1145/2644805

[17] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD) *(SSYM'05)*. USENIX Association, USA, 18.

[18] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-Guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 722–735. https://doi.org/10.1145/3192366.3192417

[19] Atanas Rountev, Scott Kagan, and Thomas Marlowe. 2006. Interprocedural Dataflow Analysis in the Presence of Large Libraries. In *Proceedings of the 15th International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. Springer-Verlag, Berlin, Heidelberg, 2–16. https://doi.org/10.1007/11688839_2

[20] Atanas Rountev, Barbara G. Ryder, and William Landi. 1999. Data-Flow Analysis of Program Fragments. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 235–252.

[21] Atanas Rountev, Mariana Sharp, and Guoqing Xu. 2008. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4959)*, Laurie J. Hendren (Ed.). Springer, 53–68. https://doi.org/10.1007/978-3-540-78791-4_4

[22] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 812–823. https://doi.org/10.1145/3377811.3380346

[23] Qingkai Shi and Charles Zhang. 2020. Pipelining Bottom-up Data Flow Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 835–847. https://doi.org/10.1145/3377811.3380425

[24] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[25] Fausto Spoto. 2011. Precise Null-Pointer Analysis. *Softw. Syst. Model.* 10, 2 (may 2011), 219–252. https://doi.org/10.1007/s10270-009-0132-5

[26] Manu Sridharan and Rastislav Bodík. 2006. Refinement-Based Context-Sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

[27] Manu Sridharan and Rastislav Bodík. 2006. Refinement-Based Context-Sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

[28] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

[29] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) *(OOPSLA '00)*. Association for Computing Machinery, New York, NY, USA, 264–280. https://doi.org/10.1145/353171.353189

[30] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

[31] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. 2020. *NJR-1 Dataset.* https://doi.org/10.5281/zenodo.4839913

[32] Akshay Utture and Jens Palsberg. 2021. *Artifact for ICSE-22 submission "Fast and Precise Application Code Analysis using a Partial Library".* https://doi.org/10.5281/zenodo.5551128

[33] WALA. 2015. IBM, "T.J. Watson Libraries for Analysis (WALA),". http://wala.sourceforge.net.

[34] Weilei Zhang and Barbara G. Ryder. 2007. Automatic Construction of Accurate Application Call Graph with Library Call Abstraction for Java: Research Articles. *J. Softw. Maint. Evol.* 19, 4 (July 2007), 231–252.