Evaluating the Precision of Static Reference Analysis Using Profiling

Donglin Liang, Maikel Pennings, and Mary Jean Harrold
College of Computing,
Georgia Institute of Technology
Atlanta, GA 30332, USA
{dliang,pennings,harrold}@cc.gatech.edu

Abstract

Program analyses and optimization of Java programs require reference information that determines the instances that may be accessed through dereferences. Reference information can be computed using reference analysis. This paper presents a set of studies that evaluate the precision of some existing approaches for identifying instances and for computing reference information in a reference analysis. The studies use dynamic reference information collected during run-time as a lower bound approximation to the precise reference information. The studies measure the precision of an existing approach by comparing the information computed using the approach with the lower bound approximation. The paper also presents case studies that attempt to identify the cases under which an existing approach is not effective. The presented studies provide information that may guide the usage of existing reference analysis techniques and the development of new reference analysis techniques.

1. INTRODUCTION

Program analyses and optimization of Java programs require reference information that specifies the instances of classes[1] that may be accessed through dereferences. Reference variables in Java programs and pointer variables in C programs have many similarities. Thus, researchers have suggested the use of points-to analysis algorithms, originally developed for use in analyzing pointers in C programs, for the computation of reference information for Java programs. Recent work has presented several approaches for extending flow-insensitive and context-insensitive points-to analysis algorithms for analyzing Java programs [9, 11, 15]. This work has also evaluated the performance of various extension approaches. In previous work, we showed that, by taking advantage of the simplicity of the common ways in which Java programs are written, the efficiency and the precision of a flow- and context-insensitive points-to analysis algorithm can be improved significantly for Java programs [9].

Despite these results, many issues regarding static reference analysis remain unresolved. One important issue is determining the precision of the existing algorithms. This information will help determine whether there is a need to develop more precise static reference analyses. Another important issue is identifying the cases in which existing algorithms are imprecise. This information can be used to develop more effective techniques for handling such cases in static reference analysis to achieve a better trade-off between precision and efficiency.

To address these issues, we must measure the precision of a static reference-analysis algorithm. Measuring the absolute precision of such an algorithm is difficult, however, because the problem of computing precise reference information is undecidable. Mock et al. [10] used dynamic points-to information, collected by executing C programs, to evaluate the precision of the static points-to information computed by several existing points-to analysis algorithms [4, 13, 14]. They reported that, for the programs they studied, static points-to information may contain a significant amount of spurious information. Although these studies give insight into the precision of static points-to information for several existing algorithms, they are limited to evaluation of analyses on C programs.

To measure the precision of reference-analysis algorithms for Java, we use a similar approach. We use dynamic

[1] In the remainder of the paper, we refer to instances of classes as instances.

reference information, collected during run-time, as a lower-bound approximation to the precise reference information. We then measure the precision of a reference-analysis algorithm by comparing the information computed by the algorithm with this lower-bound approximation. If the comparison always yields little difference, we can conclude that the information computed by the algorithm has high precision. If, in some cases, the comparison yields a significant difference, we can study these cases and develop more effective techniques for handling them.

This paper presents a set of empirical studies, using this approach, that evaluate the precision of static reference-analysis algorithms using dynamic reference information. The studies evaluate two aspects of static reference analysis: the approach for identifying instances and the mechanism for computing the reference information. A static reference analysis can be imprecise because it must use a fixed number of names to identify an unbounded number of instances created during the execution of a program. A static reference analysis can also be imprecise because it cannot precisely determine whether a program path is executable. For efficiency, an algorithm may use a small number of names to identify the instances, and use a flow-insensitive approach that assumes any sequence of the program statements is executable to compute reference information. Such an algorithm may compute very imprecise information. Our studies evaluate the precision of some existing approaches for identifying instances and computing reference information. Guided by the results of the studies, we performed two case studies that attempt to identify the cases under which an approach is not effective. Identifying and investigating these cases may guide the development of new reference analyses.

One major contribution of this paper is that it presents the results of the first set of empirical studies that evaluate various approaches for identifying instances in Java programs. One of these approaches—identifying heap-allocated memory (e.g., instances) using the allocation site—has been widely used in points-to analysis for C without any empirical evaluation. Our studies show that these approaches may be sufficiently precise for instances allocated at most allocation sites in a program. Our studies also show that using call strings to distinguish instances allocated at the same allocation site may improve the precision of static reference analysis.

Another major contribution of this paper is that it presents the results of the first set of empirical studies that compare reference information computed by Andersen's algorithm, a flow-insensitive and context-insensitive algorithm [1], with the reference information recorded during the execution of the program. Our studies show that Andersen's algorithm may compute very precise information for many allocation sites in some subjects, but may compute very imprecise information for many allocation sites in other subjects. New static reference analysis techniques may need to be developed to compute more precise information.

```
1. main() {                8. int *getint()
2.    int *p, *q;          9. {
3.    p = getint();        10.   int *t;
4.    q = getint();        11.   t = malloc(4);
5.    *p++;                12.   *t=0;
6.    *q++;                13.   return t;
7.  }                      14.  }
```

Figure 1: Example Program 1.

A third major contribution of this paper is that it presents the results of two case studies in which we investigate the cause of the imprecision in identifying instances and in computing points-to information using some existing approaches. We found that these approaches may compute very imprecise information for instances that are used to construct sophisticated data structures, such as graphs. Therefore, to compute more precise information, new static analysis techniques must provide more effective mechanism for handling such data structures.

2. STATIC REFERENCE ANALYSIS

A static reference analysis must perform two tasks. The first task is to identify the instances of various classes that may be created in a program. We refer to an approach for performing this task as a *naming scheme*. A static reference analysis must use a finite number of names to identify the instances created during the execution of a program. Because the number of instances created by a program may be unbounded, a name may be used to identify more than one instance. Because a static reference analysis computes information based on these names, it computes the same information for the instances identified using the same name. If different instances identified by the same name are accessed by the program in different ways, the points-to information computed by the reference analysis will be imprecise.

A static reference analysis can identify instances using an approach for identifying memory blocks allocated from the heap in C programs. Many of the points-to analyses for C programs, including those that have been extended for Java programs, use an allocation site to identify the memory blocks allocated from the heap at this allocation site (e.g., [1, 2, 8, 7, 14]). However, because different blocks allocated at the same allocation site may be used in different ways, this approach may result in very imprecise information. For example, for the program in Figure 1, a points-to analysis identifies the memory blocks allocated at statement 11 using h_11. Thus, it reports that statements 5 and 6 may access the same memory location. Such information is spurious because these two statements do not access the same memory location during any execution.

Some points-to analyses for C programs [3] identify the memory blocks allocated from the heap using the allocation site plus a call string that encodes a set of n procedure calls that are on the top of the stack when the allocation occurs. In this way, the points-to analysis can distinguish the memory blocks allocated at an allocation site by the call strings, and thus, can compute more precise information. For example, for the program in Figure 1, a points-to analysis uses h_11_3 to identify the memory blocks allocated at statement 11 when `getint()` is called at statement 3, and uses h_11_4 to identify the memory blocks allocated at statement 11 when `getint()` is called at statement 4. Thus, it reports that statements 5 and 6 do not access the same memory location. This example shows that this naming scheme allows the points-to analysis to compute more precise information than using the previous naming scheme.

Identifying memory allocated from the heap using the allocation site plus a call string may increase the number of names that identify memory locations, and thus, may increase the cost of a points-to analysis. In addition, this approach may still result in imprecise points-to information because memory blocks allocated from the same allocation site may be used differently even when the call strings are the same.

The second task that a reference analysis must perform is to determine the potential instances that may be accessed through dereferencing a reference variable at a statement. Given a naming scheme, a reference analysis reports a set of names that identify the memory locations whose addresses may be stored in a reference variable p at a statement s when a program is executed. This set of names is typically referred to as the *points-to* set of p at s. Let I be the set of all instances that may be identified by name N in the program. A reference analysis must include N in the points-to set of p at s if the address of any instance in I is present in p immediately before s is executed. Because the problem of tracking the exact values for a variable is undecidable, a reference analysis may compute imprecise information.

A reference analysis can compute the reference information using approaches, which are used in points-to algorithms, that determine the potential memory locations that may be accessed through each pointer dereference in C programs. Many points-to analysis algorithms have been developed. These algorithms can be classified according to their approach for modeling control-flow within a procedure and among procedures. A *flow-sensitive* algorithm (e.g., [7, 2]) considers the execution order of program statements, and computes a set of points-to relations at each program point. A *flow-insensitive* algorithm (e.g., [1, 8, 14]) ignores the execution order of program statements, and computes one set of points-to relations for all program points in a procedure or in the entire program. A *context-sensitive* algorithm (e.g., [7, 8, 2]) matches procedure calls with returns when it propagates information across procedure boundaries. A *context-insensitive* algorithm (e.g., [1, 14]) treats procedure calls and returns as if they were goto statements, and does not match procedure calls with returns when it propagates information across procedure boundaries.

Typically, a flow-insensitive algorithm is more efficient but less precise than a flow-sensitive algorithm, and a context-insensitive algorithm is more efficient but less precise than a context-sensitive algorithm. However, studies [8] show that a flow-insensitive and context-insensitive algorithm may compute points-to information that is close in precision to that computed by a more expensive flow-sensitive and context-sensitive algorithm.

3. RECORDING DYNAMIC REFERENCE INFORMATION

We use the Java Virtual Machine Profiler Interface (JVMPI)[2] to record the dynamic information from the execution of the subject programs. JVMPI provides a communication channel between the Java Virtual Machine (JVM) and a profiler. Through JVMPI, the profiler can register the events in which it is interested to JVM. As JVM executes a subject program, it notifies the profiler when a registered event occurs. The profiler can then retrieve a variety of information related to the event through interface functions provided by JVMPI.

Our profiler records information for two kinds of events. The first kind of event is an *instance-creation event*. For instance-creation events, when an instance is created, our profiler records the instance ID assigned by JVM, the type of the instance, the source line number for the corresponding "new" statement, and the method that contains the "new" statement. Such information is used to map the instance to its allocation site. The profiler can also record the n (n is a parameter for the profiler) top-most callsites on the stack when the instance-creation event occurs. Each callsite can be identified using the signature of the invoked method, the source line number for the corresponding "call" statement, and the signature of the method that contains the "call" statement. These n callsites can be used to construct the call string that distinguishes this instance from the instances created at the same allocation site under other calling contexts.

The second kind of event that our profiler records is a *method-call event*. When a non-static method is called, the profiler records the signature of the invoked method, the source line number for the corresponding "call" statement, and the signature of the method that contains the "call" statement, for identifying the callsite. The profiler also records the ID of the receiver instance on

[2] http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/, Sun Microsystems, Inc.

which this method call occurs. In this way, for each execution of a subject program, the profiler records the set of instances that may be accessed through dereference at the callsites. We use this information to measure the precision of a static reference analysis.

4. EMPIRICAL STUDIES

This section presents the results of several empirical studies. The studies evaluate the precision of two approaches, discussed in Section 2, that identify instances in reference analysis. The studies also evaluate the precision of computing reference information using an extension of Andersen's algorithm that we presented in previous work [9]. The extension identifies each instance using the instance's allocation site, and computes reference information using a flow-insensitive and context-insensitive approach. We implemented the extension using our Java Architecture for Bytecode Analysis (JABA)[3].

4.1 Subject Programs

We performed the studies on a set of real Java programs. For each subject program, we created, according to the description of the program, a set of test cases that exercise different functionalities of the program. We executed the program with each test case and collected the dynamic information using the profiler. Because our reference analysis does not analyze library classes, our profiler does not record information about the events that occur in library-provided code. In addition, because many subject programs create a large number of instances for class "java.lang.String" and class "java.lang.StringBuffer", recording information about these instances may cause the profiler to run very slowly and to consume a large amount of disk space. Therefore, our profiler does not record information about these instances.

Table 1 shows the subject programs that we used in our studies. For each subject program, the table shows the number of lines of code (*Locs*), the number of classes (*Cls*), the number of methods (*Meth*), the number of methods analyzed by our implementation of Andersen's algorithm (*And*), the number of methods executed by all test cases (*Dyn*), and the value of *Dyn/And* in percentage (%). The size and coverage information shown in the table exclude the library classes. Because some methods in each program are not reachable from `main()`, our implementation of Andersen's algorithm may not analyze those methods. Therefore, for each subject, the number shown in column *And* is smaller than the number shown in column *Meth*. To better understand the coverage, we compared the number of methods analyzed by Andersen's algorithm to the number of methods covered by the test cases. The table shows that, for many subject programs, our test cases cover more than 70% of the methods that have been analyzed by Andersen's algorithm.

[3]http://www.cc.gatech.edu/aristotle/, Georgia Tech.

	Subject Size			*Coverage*		
program	*Locs*†	*Cls*	*Meth*	*And*	*Dyn*	*%*
JavaSim	4078	37	242	72	56	78%
antlr	32030	148	1858	950	682	72%
jar	1839	8	89	36	24	67%
jas	4620	116	413	295	264	89%
javac	28191	151	1404	1266	779	62%
java_cup	10155	35	372	269	206	77%
jbf	6025	45	548	184	100	54%
jess	19317	207	1132	974	757	78%
jfe	31636	310	1837	830	726	87%
jlex	7351	20	134	105	92	88%
jtar	11904	40	202	147	97	66%
kawa	27394	319	1989	1792	1331	74%
raja	6369	65	391	37	32	86%
sablecc	44521	295	2025	1598	1376	86%
toba	6417	26	196	150	105	70%

†*Locs* for each subject is calculated using unix command "wc -l" on the java files.

Table 1: Sizes of the subject programs and number of methods covered by Andersen's algorithm and the test cases.

Note that for `raja` and `JavaSim`, only a small fraction of methods are analyzed by Andersen's algorithm because both programs contain a library and a simple driver that uses a small subset of the library classes. Also note that for `jbf`, `jar`, and `antlr`, less than 50% of the methods are analyzed by Andersen's algorithm. The coverage for `jbf` is low because `jbf` contains classes that are used for both `jb` and `jf`, but `main()` in the program is a driver that exercises the functionalities for `jb` only. The coverage for `jar` is low because `jar` contains a set of classes that implement a verifying feature, but are not used by the program. For `antlr`, the percentage is low because `antlr` contains a set of classes that are used for debugging purposes.

4.2 Study 1

This study evaluates the precision of identifying instances in reference analysis using the existing naming schemes presented in Section 2. Let I be the set of instances identified by name N according to a naming scheme. For any instance $i \in I$, let $C[i]$ be the set of callsites at which i is recorded as the receiver by the profiler; a reference analysis must report that N may be the receiver at the callsites in $C[i]$. Therefore, a reference analysis must report that N may be the receiver at the callsites in C that is defined as the following

$$C = \bigcup_{i \in I} C[i] \qquad (1)$$

However, if N is reported to be the receiver at a callsite c, we expect that any instance $i \in I$ can be observed as the receiver at c by the profiler. Thus, if C is not

```
1. A createA(int tag) {
2.    A p = new A();
3.    if( tag>=0 )
4.      p.setPositive();
5.    else
6.      p.setNegative();
7.    return p;
8.  }
```

Figure 2: Example Program 2.

the same as $C[i]$, then the reference analysis computes imprecise information for i.

For example, let h_2 be the name that identifies the instances created at statement 2 in the program in Figure 2. Suppose that instance i_1 is created at statement 2 when `createA()` is called with a value greater than 0, and i_2 is created at statement 2 when `createA()` is called with a value less than 0. i_1 will be the receiver for the method call at statement 4, and i_2 will be the receiver for the method call at statement 6. Thus, $C[i_1] = \{4\}$ and $C[i_2] = \{6\}$. A reference analysis must report that h_2 may be the receiver at the set of callsites $C = \{4, 6\}$. Because h_2 identifies both i_1 and i_2, we may conclude that both instances can be receivers at statements 4 and 6. Such information is imprecise.

Our studies compare the difference between $C[i]$ and C for each instance i that has been recorded for all test cases. If $C[i]$ is the same as C, then we say that the naming scheme is *empirically precise* for i. Otherwise, we say that the naming scheme is *empirically imprecise* for i. If a naming scheme is empirically precise for i, then a reference analysis using this naming scheme may compute precise information for i. However, if the naming scheme is empirically imprecise for i, then no reference analysis using this naming scheme can compute precise information for i.

In this study, for each instance i created during the execution of each subject program, we computed a precision reference value $r[i] = |C[i]|/|C|$, where $C[i]$ and C are defined in the previous paragraph according to a particular naming scheme, and $|C[i]|$ is the size of $C[i]$ and $|C|$ is the size of C. $r[i]$ shows the difference between $C[i]$ and C. If $r[i]$ is 1, then the naming scheme is empirically precise for i. In the study, we consider four naming schemes: level-0 naming scheme identifies instances using the allocation site only; level-x $(0 < x < 4)$ naming scheme identifies instances using the allocation site plus a call string whose length is x.

Figures 3 and 4 show the distribution of the precision reference value for each instance created during the execution of a subject. Each graph in the figure represents the data for a subject program.[4] In each graph, there are four bars in each group. From left to right, each bar represents the data computed for level-0 through level-3 naming schemes, respectively. A bar in the group marked with $[v, v + 0.2)$ ($v = 0.0, 0.2$,0.4, 0.6, 0.8) represents the percentage of instances whose precision reference value $r[i]$ is in the interval $[v, v + 0.2)$ (i.e. $v \leq r[i] < v + 0.2$); a bar in the group marked with 1 represents the percentage of instances for which the naming scheme is empirically precise.

From the figures, we can see that, for subjects such as `JavaSim` and `jar`, each naming scheme is empirically precise for a large percentage ($> 65\%$) of instances created during execution. From Table 1, we can see that these subjects are relatively small compared to the others. By inspecting the description and the source code of these programs, we found that they are relatively simple, and they do not construct complicated data structures. For example, `JavaSim` contains a driver that sets up an simulation environment and then simulates the traffic of a network model. It creates only a small number of instances during the execution.

From the figures, we can also see that, for the other subjects, each naming scheme is empirically precise for a much lower percentage of instances created during the execution of the programs. For example, for subject `jess`, all the naming schemes are empirically precise for less than 15% of instances.

To identify the factors that may cause such imprecision in these programs, we investigated the precision for identifying instances that are allocated at each allocation site in these subjects. For each allocation site, we computed the average of the precision reference values for all the instances allocated at the site. Table 2 shows the distribution of such an average for the allocation sites. In the table, each row represents the data for a specific naming scheme and a specific subject; each number, except the last one, in each row indicates the number of allocation sites at which the average of the precision reference values for the instances allocated at each site is in $[v, v + 0.1)$ ($[v, v + 0.1)$ is marked at the head of the corresponding column). The last number in each row indicates the number of allocation sites that create instances whose precision reference value is one.

From the table, we can see that, for many subject programs, the naming schemes may be very precise for instances created at a large percentage of allocation sites; the naming schemes may be very imprecise for instances created at a very small number of allocation sites. For example, for subject `kawa`, the level-3 naming scheme is empirically precise for instances created at about 70% of allocation sites in the program; for the same naming scheme, only 13 out of 495 allocation sites create instances whose average of precision reference values is

[4]Data for `jtar`, `raja`, and `toba` are not shown in the Figures because in these programs, all the naming schemes are empirically precise for more than 95% of instances.

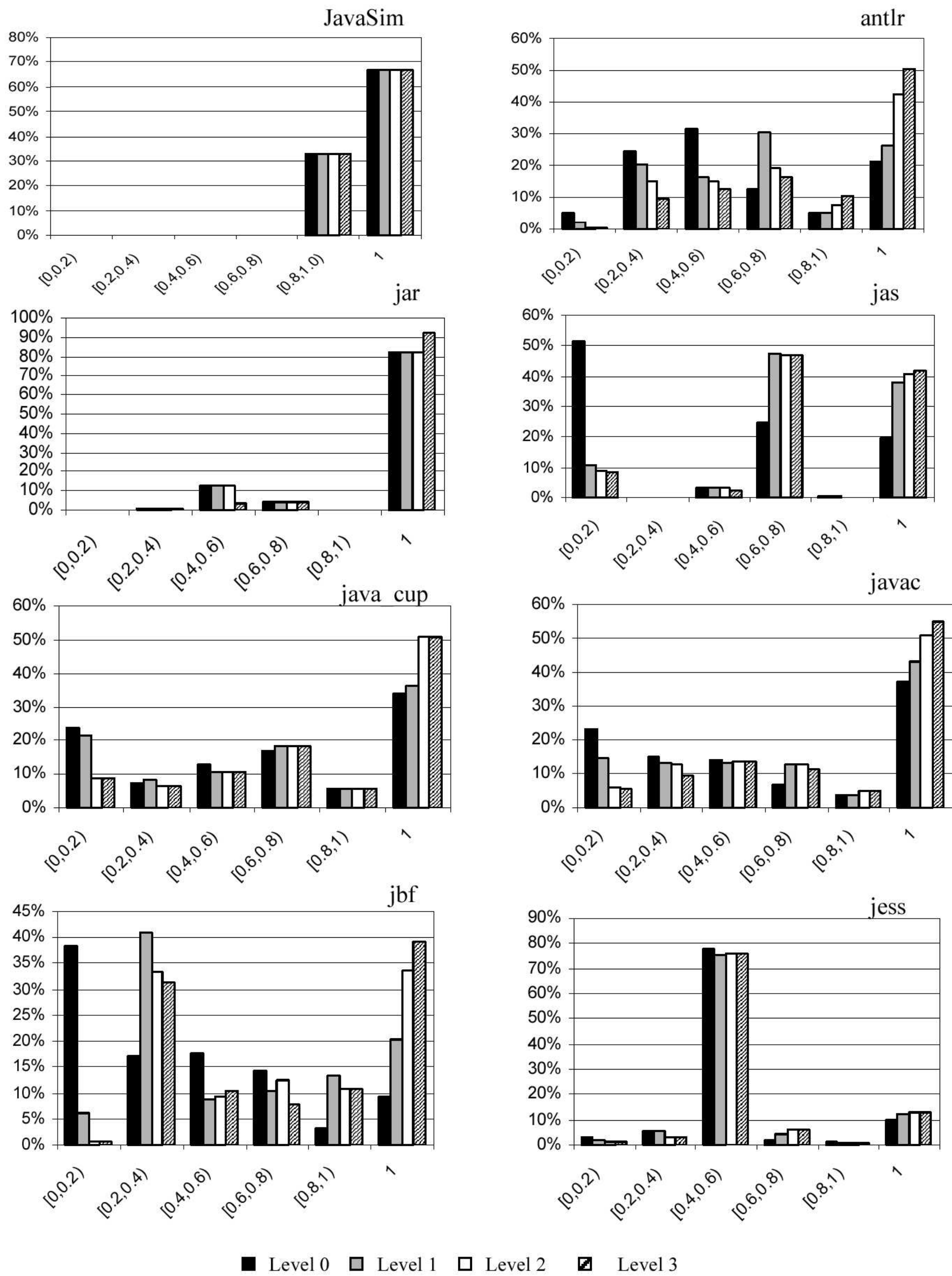

Figure 3: Distribution of the precision reference values for all instances .

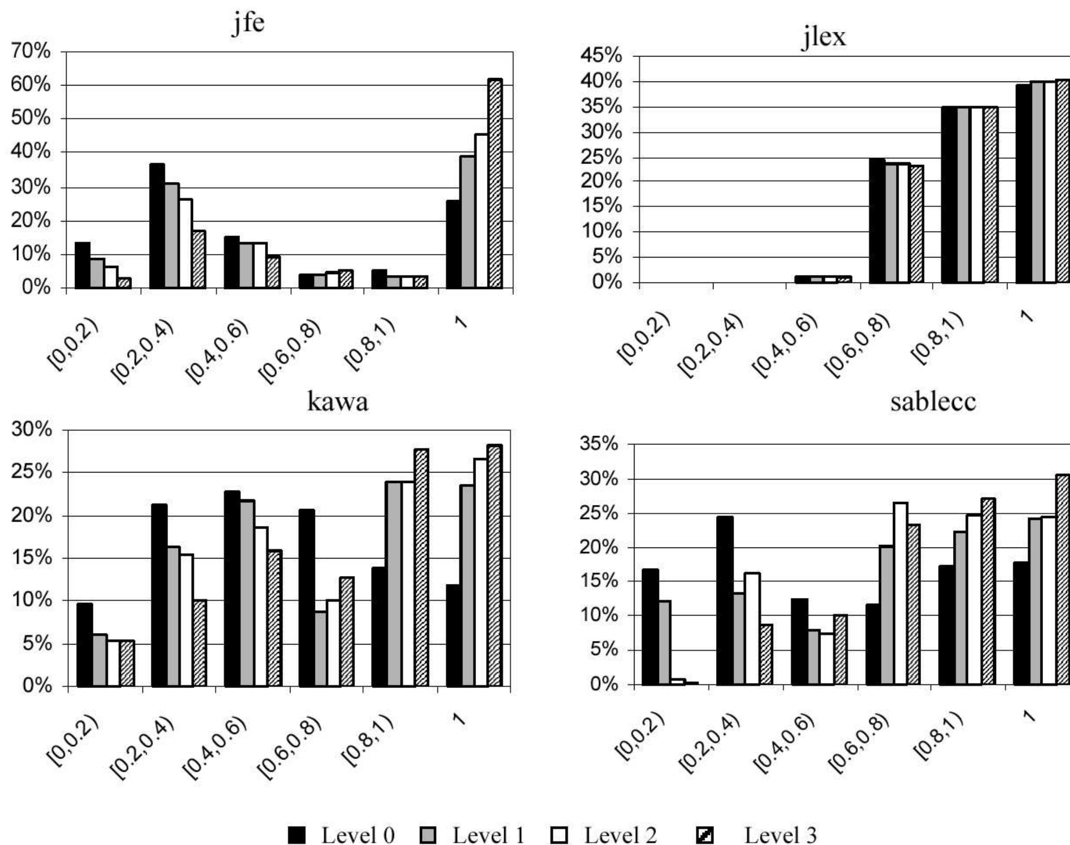

Figure 4: Distribution of the precision reference values for all instances (continued).

from 0.0 up to 0.4.

To investigate these problematic allocation sites that create instances whose precision reference values are extremely low, we inspected the descriptions and the source code of these programs. We found that most of these programs are language processors. These language processors typically process some text or binary files, build an internal data structure to represent the input, and then apply operations on this internal data structure. Such a data structure is typically a tree or a graph that contains many nodes. These nodes are created at a few allocation sites and may be used in different ways according to the information associated with individual nodes or their relations with other nodes.

For example, in `java_cup`, when the level-3 naming scheme is used, 99% of the instances that have a precision reference value between 0.0 and 0.6 are created at allocation sites that create instances for the following classes

java_cup.lalr_item
java_cup.lalr_item_set
java_cup.terminal_set
java.lang.Vector

The instances created for java.lang.Vector are used by instances of java_cup.lalr_item. The instances created for the other classes are used for the construction of the data structures for the parser to perform various actions according to the production rules specified by the program's inputs. Depending on the content, these instances may be accessed by the program at different statements. Therefore, identifying these instances using the allocation sites or the allocation sites plus call strings may result in very imprecise information. To compute more precise information for such instances, new naming schemes need to be developed to account for the content of such instances.

From the data presented in Figures 3 and 4, we can also see that, for some subject programs, using call strings to distinguish the instances allocated at the same allocation site may improve the precision of the information computed by static reference analysis. For example, for subject `kawa`, the level-0 naming scheme is empirically precise for only 12% of the instances created during the execution; in contrast, the level-3 naming scheme is empirically precise for 28%. However, for other subjects, such as `JavaSim`, `jess`, `jlex`, and `toba`, the improvement in the precision is insignificant.

program	*Naming scheme*	(0.0, 0.1)	[0.1, 0.2)	[0.2, 0.3)	[0.3, 0.4)	[0.4, 0.5)	[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0)	1
JavaSim	Level 0	0	0	0	0	0	0	0	2	0	1	17
	Level 1	0	0	0	0	0	0	0	1	0	1	18
	Level 2	0	0	0	0	0	0	0	0	1	1	18
	Level 3	0	0	0	0	0	0	0	0	1	1	18
antlr	Level 0	0	1	3	11	9	26	14	20	24	21	125
	Level 1	0	1	0	6	9	17	16	20	23	23	139
	Level 2	0	0	0	5	6	17	12	23	23	28	140
	Level 3	0	0	0	4	5	16	13	21	23	32	140
jar	Level 0	0	0	0	0	2	1	1	0	1	0	33
	Level 1	0	0	0	0	2	1	1	0	1	0	33
	Level 2	0	0	0	0	2	1	1	0	1	0	33
	Level 3	0	0	0	0	2	0	1	0	1	0	34
jas	Level 0	2	3	0	0	0	4	5	1	40	19	115
	Level 1	0	2	0	0	0	3	5	3	40	20	116
	Level 2	0	2	0	0	0	2	4	3	40	16	122
	Level 3	0	1	1	0	0	1	4	4	38	17	123
java_cup	Level 0	1	0	2	0	5	6	9	13	12	9	130
	Level 1	0	1	2	0	4	5	9	15	12	9	130
	Level 2	0	0	2	0	4	5	9	15	12	10	130
	Level 3	0	0	2	0	4	5	9	15	12	10	130
javac	Level 0	2	7	10	14	14	15	31	17	33	36	272
	Level 1	1	1	8	9	14	15	25	23	36	38	281
	Level 2	1	1	6	4	11	10	25	18	37	49	289
	Level 3	0	2	4	5	10	9	22	17	33	55	294
jbf	Level 0	0	2	0	2	3	1	2	9	4	5	18
	Level 1	0	1	0	0	3	3	2	8	5	6	18
	Level 2	0	0	0	0	2	2	2	9	6	6	19
	Level 3	0	0	0	0	1	3	1	9	5	6	21
jess	Level 0	2	5	6	19	23	23	25	29	33	12	200
	Level 1	0	2	7	16	19	19	24	30	38	18	204
	Level 2	0	1	4	13	24	20	21	26	39	24	205
	Level 3	0	0	3	12	22	22	21	28	37	25	207
jfe	Level 0	0	7	21	16	9	20	20	18	20	23	189
	Level 1	0	0	15	9	8	18	18	18	24	39	194
	Level 2	0	0	10	9	6	16	20	20	21	43	198
	Level 3	0	0	6	4	5	14	19	21	21	54	199
jlex	Level 0	0	0	0	1	0	2	1	5	9	6	54
	Level 1	0	0	0	1	0	2	1	3	10	7	54
	Level 2	0	0	0	1	0	2	1	3	10	6	55
	Level 3	0	0	0	1	0	1	2	3	9	7	55
kawa	Level 0	2	5	10	7	16	17	33	30	35	26	314
	Level 1	1	3	10	6	14	12	29	26	35	36	323
	Level 2	0	3	9	3	16	10	23	20	30	42	339
	Level 3	0	3	7	3	13	10	22	19	28	44	346
sablecc	Level 0	1	2	5	4	4	14	11	31	31	40	318
	Level 1	0	1	2	3	3	11	12	32	31	43	323
	Level 2	0	0	2	2	2	11	12	34	32	43	323
	Level 3	0	0	1	1	3	10	12	27	37	44	326

Table 2: Distribution of the average precision reference values for instances created at individual allocation sites. Note that data for `jtar`, `raja`, and `toba` are not presented because almost all instances in these programs have a precision reference value of one.

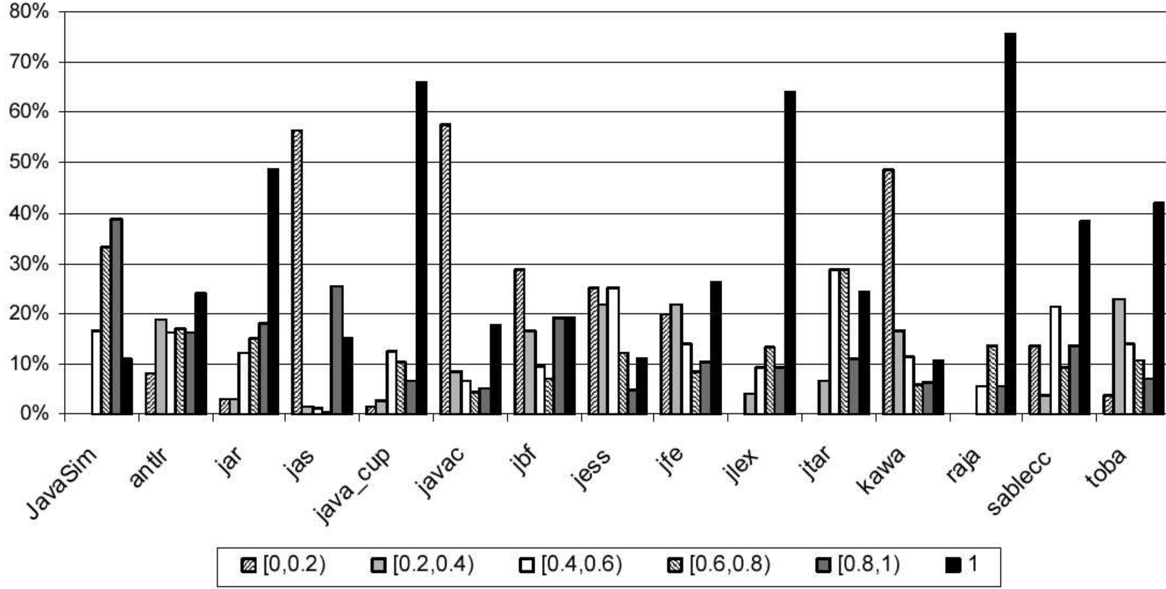

Figure 5: Distribution of the precision reference values for allocation sites.

In summary, naming schemes that identify instances using allocation sites or allocation sites plus a call string may be very imprecise for a large percentage of instances created during the execution of a program. However, such imprecision may occur for instances allocated at a small percentage of allocation sites in each program. Therefore, information computed by static reference analysis using these naming schemes may be useful for reasoning about certain properties of the instances created at a large percentage of allocation sites. Our case studies indicate that the instances for which the naming schemes are highly imprecise are typically created at a small number of allocation sites, and are often used to construct a large complicated data structure. New naming schemes must be developed to compute more precise reference information for such a data structure. Because such a data structure involves only the instances created at a small portion of the allocation sites, a hybrid naming scheme may be the best approach: at the large percentage of allocation sites where identifying instances using existing naming schemes is sufficient, we use the existing naming scheme; for the rest of the allocation sites, we use a new naming scheme. Such a hybrid naming scheme may yield a good trade-off between precision and efficiency. Our study also shows that using call strings to distinguish the instances created at the same allocation site may enable a static reference analysis to compute more precise information for instances allocated at some allocation sites in many programs.

4.3 Study 2

This study evaluates the precision of the mechanism that is used to compute points-to information in Andersen's algorithm. Andersen's algorithm uses the allocation sites to identify instances. Let I be the set of instances that are identified by name N. For any instance $i \in I$, let $C[i]$ be the set of callsites at which i is recorded as the receiver. Andersen's algorithm must report that N may be a receiver at the set of callsites

$$C = \bigcup_{i \in I} C[i] \qquad (2)$$

However, because Andersen's algorithm is not precise, it may report that N may be the receiver at a larger set of callsites C'. We want to measure the difference between C and C'. The difference may serve as an indicator of the precision of Andersen's algorithm. C' may contain callsites that are not executed by any of our test cases. We do not consider those callsites because they may never be reached by any execution of the program.

In this study, for each name N that identifies the instances created at an allocation site, we compute a precision reference value $R[N] = |C|/|C' \cap Cover|$, where C and C' are defined in the previous paragraph and $Cover$ is the set of callsites that have been exercised by at least one test case. If $R[N]$ is close to 1, then using this naming scheme, any other static reference analysis may not compute significantly more precise information for N than Andersen's algorithm. However, if $R[N]$ is not close to 1, then some other static reference analysis may compute more precise information for N than Andersen's algorithm.

Figure 5 shows the distribution of the precision reference values computed for the allocation sites in each subject.

In the figure, from left to right, each of the first five bars within a group represents the percentage of allocation sites whose precision reference values are in $[v, v + 0.2)$ ($v = 0.0, 0.2, 0.4, 0.6, 0.8$); the sixth bar within a group represents the percentage of allocation sites whose precision reference values are 1. As mentioned in Subsection 4.1, we do not consider allocation sites for "java.lang.String" and "java.lang.StringBuffer". To compare the information computed by Andersen's algorithm with that collected by the profiler, we do not consider the allocation sites not exercised by test cases.

The graph shows that, for `jar`, `java_cup`, `jlex`, and `raja`, information computed by Andersen's algorithm is very close to that recorded by the profiler for a large percentage of allocation sites. However, the graph also shows that, for other subjects (e.g., `jas`, `javac`, and `kawa`), information computed by Andersen's algorithm are quite different from that recorded by the profiler for a large percentage of allocation sites. The result indicates that information computed by Andersen's algorithm is quite imprecise for these subjects. New reference-analysis algorithms could be developed for computing more precise information for these programs.

To understand the reasons why Andersen's algorithm is so imprecise for some subject programs, we performed a case study on one of our subjects `jess`. `jess` is a rule engine and scripting environment for building an expert system.[5] `jess` processes programs written in a language similar to CLIPS. In the case study, we identified, in `jess`, the allocation sites whose precision reference values are between 0.0 to 0.1. We then checked the source codes of `jess` to see how the instances created at these sites are used by the program. In this program, there are 74 allocation sites whose precision reference values are between 0.0 and 0.1. By checking the source code, we found that these allocation sites create instances for one of the following seven classes

jess.Context	3
jess.Funcall	1
jess.FuncallValue	6
jess.IntArrayValue	1
jess.Value	51
jess.ValueVector	6
jess.Variable	6

The number following each class name indicates the number of allocation sites that create instances for this class. Each number includes only those allocation sites whose precision reference values are between $(0.0, 0.1)$.

We further found that most of the instances of these classes (except jess.Context) created in the program are used to construct a parse tree for the program inputs. The parse tree is a recursive data structure. Instances of the same type (e.g., jess.Value) in the data structure are created at various allocation sites and are used for different purposes. For example, an instance of jess.Value has a field "m_type" that determines how the other fields of this instance should be used. In many places, the program checks the value of "m_type" in an instance and accesses this instance according to the value of "m_type". Typically, an instance of jess.Value created at an allocation site has a specific value for "m_type". Therefore, instances of jess.Value created at different allocation sites are accessed differently by the program. Andersen's algorithm cannot distinguish different parts of a recursive data structure, and thus, will compute the same information for different allocation sites that create instances for jess.Value. Therefore, Andersen's algorithm will compute very imprecise information for these allocation sites. For similar reasons, Andersen's algorithm also computes very imprecise information for allocation sites of classes such as jess.ValueVector and jess.FuncallValue.

In summary, Andersen's algorithm may compute imprecise information for a large percentage of allocation sites in many programs. Therefore, there is a need to develop more precise static reference analyses. Our case study reveals that, instances allocated at the allocation sites for which Andersen's algorithm computes highly imprecise information, may often be used to compose sophisticated data structures. Because many existing points-to algorithms (e.g., [1, 2, 3, 8, 7, 14]) developed for C have very limited power in distinguishing different parts in such data structures, we expect these algorithms, when being extended for Java programs, to compute very imprecise information for these callsites. Shape analysis techniques (e.g., [5, 6, 12]) have been developed to compute more precise information for sophisticated data structures in C. The effectiveness of these techniques on Java programs must be evaluated. In addition, these techniques do not seem to scale to large C programs. New reference-analysis algorithms may need to be developed for handling such data structures in Java programs more effectively.

5. CONCLUSION

This paper presents empirical studies that evaluate the precision of existing approaches in identifying instances in static reference analysis and in computing points-to information in static reference analysis. Our studies show that

- Identifying instances using the allocation sites may be sufficiently precise for instances allocated at most of the allocation sites in a program.
- Using call strings to distinguish instances allocated at a allocation site may improve the precision of identifying instances in some programs.
- Andersen's algorithm may compute very precise information for most allocation sites in some subjects, but it may compute very imprecise information for most allocation sites in other subjects.

[5] http://herzberg.ca.sandia.gov/jess/.

Our studies suggest that hybrid approaches for identifying instances and computing points-to information may need to be developed: for those allocation sites at which existing approaches are precise enough, we use such approaches; for the remainder, we develop new approaches for effectively handling each case.

We also use the results of our studies to gain insights into possible ways in which we may develop new reference-analysis techniques. We perform case studies to investigate the characteristics of the program constructs that cause the imprecision in existing approaches for identifying instances in static reference analysis and in computing points-to information. We found that these existing approaches may be very imprecise for sophisticated data structures. To compute more precise information, new static reference analysis must employ more effective techniques for handling such data structures.

In future work, we will collect additional subject programs and create more test cases to continue our empirical studies. New subjects and test cases will give us more information that may enhance the confidence of our conclusion, and may lead to new discoveries. We will also perform additional case studies to gain more insights into the common usage patterns of references in Java programs. Such insights may guide us to develop effective techniques for handling such usage patterns in static analysis. Such insights may also help us to identify the limitations of static reference analysis, and may lead us to discover other alternatives, such as combining static analysis with dynamic analysis, for computing more precise reference information.

6. ACKNOWLEDGEMENTS

This work was supported in part by a grant from Boeing Aerospace Corporation to Georgia Tech, by National Science Foundation awards CCR-9988294, CCR-0096321, and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. Huaxing Wu helped with the creation of the graphs, and Tongyu Li helped with the creation of the test cases for the subjects.

7. REFERENCES

[1] L. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.

[2] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages*, pages 133–146, Jan. 1999.

[3] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 232–245, 1993.

[4] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of 2000 Conference on Programming Language Design and Implementation*, June 2000.

[5] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[6] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis of imperative programs. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 249–260, San Francisco, California, 17–19 June 1992.

[7] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[8] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *7th ESEC/FSE*, pages 199–215, Sept. 1999.

[9] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2001.

[10] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analyses and potentials applications in program understanding and optimization. In *2001 SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, jun 2001.

[11] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java based on annotated constraints. In *Conference on Object-oriented programming, systems, languages, and applications*, Oct. 2001.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 16–31, Jan. 1996.

[13] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium*, pages 1–14, 1997.

[14] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

[15] M. Streckenbach and G. Snelting. Points-to for java: A general framework and an empirical comparison. Technical report, University Passau, Nov. 2000.