Approximating Flow-Sensitive Pointer Analysis Using Frequent Itemset Mining

Vaivaswatha Nagaraj
Indian Institute of Science
vaivaswatha@hpc.serc.iisc.in

R. Govindarajan
Indian Institute of Science
govind@serc.iisc.in

Abstract

Pointer alias analysis is a well researched problem in the area of compilers and program verification. Many recent works in this area have focused on flow-sensitivity due to the additional precision it offers. However, a flow-sensitive analysis is computationally expensive, thus, preventing its use in larger programs.

In this work, we observe that a number of object sets, consisting of tens to hundreds of objects appear *together* and *frequently* in many points-to sets. By approximating each of these object sets by a single object, we can speedup computation of points-to sets. Although the proposed approach incurs a slight loss in precision, it is shown to be *safe*. We use a well known data mining technique called frequent itemset mining to find these frequently occurring objects.

We compare our approximation to a fully flow-sensitive pointer analysis on a set of ten benchmarks. We measure precision loss using two common client analysis queries and report an average precision loss of 0.25% on one measure and 1.40% on the other. The proposed approach results in a speedup of upto 12.9x (and an average speedup of 6.2x) in computing the points-to sets.

1. Introduction

Pointer analysis plays a key role in many compiler optimizations and program verification techniques. The analysis aims at determining the possible memory locations that a pointer may point to during any run of the program[19]. This helps to uncover possible aliases in a program, thus enabling safe compiler code transformations[2] and useful hints from error detection and program verification tools[18]. For example, to infer that a variable is dead at a program point, a compiler must make sure that no alias of that variable is live at that point. Alias information is essential to many compiler optimizations such as loop invariant code motion, auto parallelization etc[25, 37]. Alias information is also useful for program verification and error detection tools to provide useful diagnostics about a program. Bug detection[12], race detection[11] and detecting vulnerabilities in web programs[21] are few examples of error detection methods that use alias information. In the absence of alias information, a compiler or an error detection tool is forced to make conservative assumptions, thus limiting its impact.

Pointer alias analysis has been shown to be undecidable [32]. This has led to a number of algorithms that achieve different balances between precision and efficiency. Production compilers often use a fast and imprecise pointer analysis method called the address-taken analysis[19]. More precise analyses have been proposed over the years. Depending on how pointer assignment statements (e.g., "$x = y$") are processed, analyses may be classified as inclusion based or unification based. In inclusion based analyses[3], the points-to set of x is updated to include (is a super-set of) the points-to set of y. On the other hand, in a unification based analysis[35], the points-to set of x and y are considered to be equal. The paper by Das[8] proposes an algorithm whose precision lies in between that of inclusion and unification based algorithms.

Another classification of pointer analysis algorithms is based on whether the analysis takes program control flow into account, in which case it is considered as flow-sensitive [14]. If the analysis ignores control flow and thus computes the same points-to sets at every program point, it is considered flow-insensitive[20]. Analyses can also be distinguished based on the calling contexts considered when analyzing a function. A context-insensitive analysis does not distinguish between the various contexts in which the function is called. On the other hand, context-sensitive analyses distinguish between some or all contexts in which the function is called[16]. A field-sensitive analysis tracks the individual members of an aggregate (such as *struct* in C) whereas a field-insensitive algorithm [22] flattens aggregates for the purposes of the analysis.

Flow-sensitive pointer analyses can compute points-to sets with higher precision[19], and thus, have been shown to be of importance to a variety of client program analyses such as analysis of multi-threaded code [34] and detection of security vulnerabilities[6]. This, along with the challenge of scaling it to large programs, has lead to the problem being an important research topic in the area of pointer analysis [14, 24, 25, 27, 29, 40]. In this work, we speed up flow-sensitive pointer analysis by approximating groups of

1-4799-8161-8/15/$31.00 ©2015 IEEE

memory locations (objects) with a single summary location. The approximation we propose is based on the observation that certain groups of objects appear *together* and *frequently* in the points-to sets of many pointers. Hence, minimizing the time spent in propagating these groups of objects during the analysis can significantly reduce the analysis time, at the cost of a small precision loss. Our key contributions in this work are:

- We analyze the flow-sensitive points-to sets of a few commonly used benchmarks and observe the frequent occurrence of a few sets of objects in the points-to sets of pointers.
- We propose a technique to identify and merge such frequently occurring object sets. We show that such a summarization of memory locations is *safe*, although it may lead to some precision loss.
- We experimentally evaluate our approximation technique on a set of ten previously used benchmarks. We observe an average speedup by a factor of 6.2x, while incurring an average precision loss of 0.25% and 1.40% on two precision measures, respectively.

The paper is organized as follows. Section 2 discusses some basics of pointer analysis. We also provide a brief introduction to frequent itemset mining here. Section 3 discusses the occurrence of frequent object sets and the motivation to approximate them. We discuss our approach to performing the actual merging of frequent objects in Section 4, followed by experimental evaluation in Section 5. In Section 6 we compare our work with other related work. Section 7 provides concluding remarks.

2. Background

In this section, we discuss some basics of pointer analysis and the idea behind the staged flow-sensitive analysis. We also summarize a few terms and definitions related to frequent itemset mining.

2.1 Pointer Analysis

Pointer (or points-to) analysis is the problem of determining at compile time, the possible values that a pointer variable may have at run time. Pointer analysis involves analyzing the following types of statements in a program, till a fixed point in the computed points-to set of each pointer variable is reached.

- $x = \&a$: (*address-of*) Pointer x is assigned the address of memory object (address-taken variable) a.
- $x = y$: (*copy*) The points-to set of y is included in the points-to set of x.
- $x = *y$: (*load*) For each object a currently in the points-to set of pointer y, the points-to set of a is included in the points-to set of x.
- $*x = y$: (*store*) For each object a currently in the points-to set of pointer x, the points-to set of y is included in the points-to set of a.

In addition to these statements, memory allocation statements (such as **malloc** or **new**) also need to be handled. These statements can be processed in different ways. The strategy used to handle allocation of heap memory (i.e., dynamically allocated memory) is sometimes referred to as a *heap model*[14]. Some analyses treat all memory locations (dynamically) allocated by an allocation statement to be a single abstract memory location. Such a strategy has often been used in different pointer analysis algorithms[18, 19] and is the one that we use in our analysis.

A common use of pointer analysis is to determine if two pointers alias. Two pointers are said to alias if the intersection of their points-to sets is not empty[18]. Alias analysis plays an important role in other program analyses and optimizations such as Mod/Ref analysis, unreachable code elimination etc[19].

2.1.1 Correctness

If there exists an execution sequence of the program in which a pointer p points to an object (address-taken variable) x, then any pointer analysis must include object x in the points-to set of p.

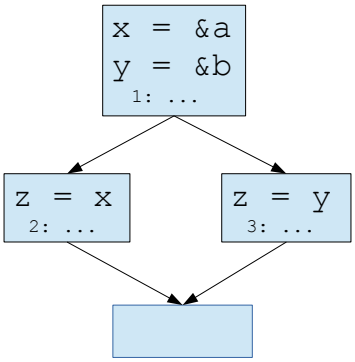
2.1.2 Precision

The definition of correctness above only specifies what objects must be present in the points-to set of a pointer p. However, determining the minimal set of objects satisfying the correctness condition is, in general, undecidable[32].

The size of points-to sets of pointers computed by a pointer analysis algorithm is a measure of the precision of the algorithm. The points-to set computed by a more-precise algorithm, for each pointer, will be a subset of the points-to set of that pointer as computed by a less precise algorithm. In the worst case (least precise), an analysis might report that every pointer may point to every memory location in the program.

2.2 Flow-Sensitive Pointer Analysis

Flow-sensitivity is a dimension of pointer analysis that determines the points-to sets for pointers at each program point[22]. A flow-sensitive algorithm respects program control flow, and hence computes points-to sets at each program point. A flow-insensitive algorithm on the other hand computes a single global points-to set for each pointer. A flow-sensitive analysis is, in general, more precise compared to a flow-insensitive analysis. Figure 1 illustrates the difference in precision between a flow-sensitive (more precise) and a flow-insensitive (less precise) algorithm. The reader may note that the points-to sets for any pointer are the same at all program points in a flow-insensitive analysis.



(a)

pointer	Flow-sensitive			Flow-insensitive		
program point →	1	2	3	1	2	3
x	a	a	a	a	a	a
y	b	b	b	b	b	b
z	-	a	b	a,b	a,b	a,b

(b)

Figure 1. Difference in points-to sets of a flow-insensitive and a flow-sensitive pointer analysis

In a flow-sensitive pointer analysis, there are two possibilities that can occur when processing a store statement "∗x = y"[22] (1) The pointer *x* points only to a single object *a*, in which case the points-to set of *a* will be replaced with the points-to set of *y*. This is called a *strong update*. (2) The pointer *x* points to more than one object, in which case the points-to sets of all of these objects will be updated to include the points-to set of *y* along with their existing values. This is called a *weak update*.

2.3 Staged Flow-Sensitive Pointer Analysis (SFS)

The staged flow-sensitive pointer analysis[14] was introduced to scale flow-sensitive pointer analysis to large programs by using memory def-use information obtained from a fast (but less precise) pointer analysis.

Prior to the introduction of the staged analysis, flow-sensitive pointer analyses needed to propagate data-flow information (points-to sets in this case) to all program points since def-use information for memory locations would not be available (a pointer analysis is needed to determine the memory locations that may be referenced in loads/stores). The staged flow-sensitive pointer analysis tackles this inefficiency by performing a fast (and less precise) pointer analysis (called AUX) prior to the main analysis. Typically, AUX is flow-insensitive. The technique then uses AUX to compute conservative def-use chains for memory locations. During the main flow-sensitive analysis, it propagates data-flow information only along these def-use chains, thus minimizing the amount of data-flow information propagated.

In this work, we use the staged flow-sensitive pointer analysis, both as a comparison, and also as the basis for our approach. AUX also enables us to extract useful information about the structure of the points-to sets and take advantage of it for the main analysis.

2.4 Frequent Itemset Mining

In this section, we give a brief introduction to the problem of frequent itemset mining, a well studied problem in the area of data mining[31]. We begin by introducing related terminology.

Let U be a set of items. A *transaction* is a subset of U. The term arises from a set of items bought in a transaction (sometimes the term *basket* is used instead of *transaction*). A *Frequent Itemset I* is a set of items that appears in (is a subset of) at least *s* transactions, where *s* is called the *support threshold*. The frequency of occurrence of *I* is called its *support*. An itemset is considered *maximal* if no proper superset of the itemset is frequent, i.e., appears more than the support threshold.

Typically, the number of transactions is high, and the size of a transaction (i.e., number of items in the transaction) is smaller. Given a set of *transactions*, the problem of finding all itemsets whose *support* is at least a specified *support threshold* is called frequent itemset mining[31].

Most algorithms for frequent itemset mining rely on the *monotonicity* property: If a set *I* of items is frequent, then so is every subset of *I*. A well known algorithm for frequent itemset mining is the *apriori* algorithm[1]. The algorithm performs multiple passes on the set of transactions,computing frequent itemsets of increasing sizes in each pass. It utilizes the set of frequent items from the previous pass to obtain a pruned set of candidates for the current pass. Although historically significant, the *apriori* algorithm is generally considered inefficient and several newer algorithms have been proposed. In this work, we use the ECLAT[41] algorithm to find frequent itemsets. This algorithm employs better heuristics (such as equivalence class clustering) to quickly determine maximal frequent itemsets. In our work, we are only interested in maximal frequent itemsets.

In our problem context, the set of all memory objects (address-taken variables) form the universe U of items. The points-to sets of pointers form the transactions. We aim to find those object sets that occur frequently in the points-to sets of many pointers.

3. Motivation

A key observation made by us in this paper is that there are object sets that occur *together* and *frequently* in the flow-

benchmark	num_freq_objects (%of total objects)	frequency
ex	34 (1.4%)	2.8%
176.gcc	157 (1.1%)	19.6%
nethack	412 (2.1%)	14.1%
254.gap	615 (7.2%)	26.2%
253.perlbmk	1353 (29.4%)	30.5%
vim	3368 (1.9%)	6.6%
python	5598 (25.5%)	20.3%
svn	6131 (25.3%)	18.1%
pine	22053 (5.4%)	16.5%
gdb	15164 (20.8%)	18.4%

Table 1. Details of frequently occurring objects

sensitive points-to sets of many pointers. [1]. We motivate this with the help of data obtained from the staged flow-sensitive pointer analysis [14], which is flow and field-sensitive, but context-insensitive. We analyze the results of this analysis for a few benchmarks to find frequently occurring object sets (i.e., sets of memory objects that are pointed-to by many pointers).

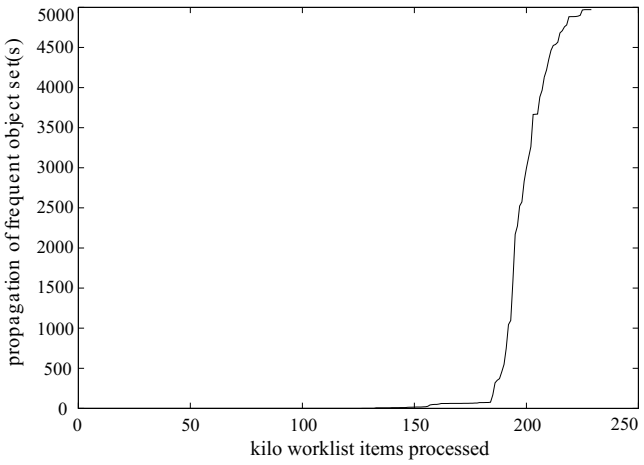
Table 1 details the number of frequent objects in the points-to sets of each benchmark along with their frequency (support) of occurrence. The object sets here are all maximal. Some benchmarks had more than one such maximal frequent object set. Except in the case of `ex` and `python`, the sizes of other frequently occurring sets were insignificant. Hence we report only the most frequently occurring set here (although we do take advantage of the others in our implementation).

The frequency of occurrence of the object sets reported here is computed as a percentage of the number of occurrences of the object set in the points-to sets of pointers. Additionally, these object sets may appear in the points-to sets of many memory objects (*address-taken* variables) at various program points too. Hence optimizing the time spent by the analysis in propagating points-to information related to frequent object sets is likely to speed up the analysis.

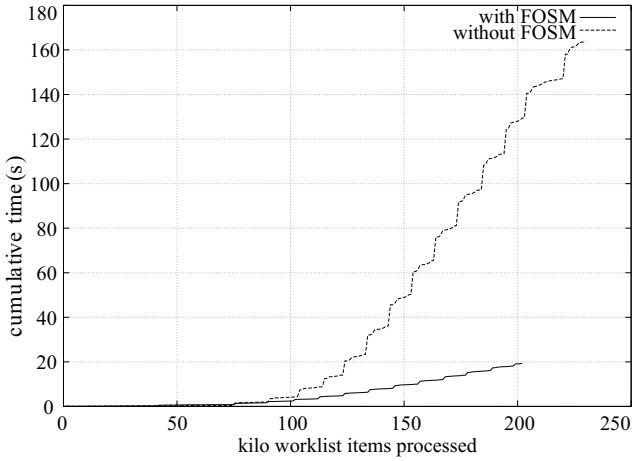
Figure 2(a) is a plot of the number of pointers that contain the frequent object sets in their points-to sets, as the analysis progresses. The progress of the analysis is measured (x-axis) in terms of the number of worklist items processed. The graph (plotted for staged flow-sensitive analysis (SFS)[14] on the benchmark `vim`), shows that after 150,000 worklist items are processed, the frequent object sets start to form and their growth and propagation is steep (happens between 185,000 to 225,000 worklist items).

What happens if the analysis is seeded with the frequent object set information, right at the beginning? We plot the cumulative execution time of the analysis against the same measure of progress (number of worklist items processed) in

[1] This has been observed before for less precise pointer analyses and was used in designing a persistent data structure for storing points-to sets [38]



(a)



(b)

Figure 2. Progress of the analysis vs (a) Growth of frequent object sets (b) Cumulative time: with and without optimizing frequent object sets.

Figure 2(b). Using the oracle information of frequent object sets in SFS, we plot the execution time for SFS without and with frequent object set optimization. Without optimization, as the number of pointers containing the frequent object sets (in their points-to sets) increase, the analysis time also starts to increase. This is mainly due to the large sizes of these frequent object sets, propagation of which is an expensive operation. SFS with frequent object set information abstracts large frequent object sets as single objects, and hence is able to reduce the time taken to propagate them. Another interesting point here is that, after optimizing, the analysis converges faster, i.e., there are fewer worklist items processed.

This motivates that identifying groups of objects that occur frequently and abstracting them as a single object can speedup the analysis. To identify these frequently occurring sets of objects in the points-to sets of pointers, we employ *frequent itemset mining*[31], a well-known problem in the

area of data mining. As already explained in Section 2.4, the set of all memory objects form the universe U of items, and the points-to sets of pointers form the transactions.

4. Frequent Object Set Merging (FOSM)

The frequent occurrence of some memory objects in the points-to sets of pointers indicates that modifying the analysis to handle these object sets better can make the analysis faster. A simple way to take advantage of these frequent object sets is to consider all objects in a frequently occurring set as a single *summary* memory location. The only requirement on such a *summary* location is that it may not be subjected to *strong* updates [10] (*strong* updates are discussed in Section 2). Other correctness aspects of merging memory objects into a single summary location are discussed in Section 4.1.

To merge sets of frequent memory objects into a single location, we need to know which object sets are frequent. One way to gather this information is to keep track of pointers that have *similar* points-to sets during the analysis. A similarity measure such as the Jaccard similarity[31] can be used to measure similarity between two sets. Jaccard similarity is measured as the size of intersection of sets divided by the size of their union. i.e., the similarity measure of two sets A and B is defined as $|A \cap B|/|A \cup B|$, where $|A|$ denotes the cardinality of set A. The similarity value can range from 0 (dissimilar) to 1 (identical). As the analysis progresses, objects pointed-to by similar pointers, satisfying certain properties (such as a minimum size for its points-to set) can be merged into a single object. Such an approximation has already been experimented with for Andersen's inclusion based (flow-insensitive) analysis[28]. We discuss some disadvantages of such an approach in Section 6.

Instead of incrementally determining objects that need to be merged, we determine candidate objects for merging and merge them even before the analysis begins. However, the data on frequent object sets (as mentioned in Section 3) is not available until the actual analysis itself finishes running. We therefore need an approximation for it in order to do the merging prior to the analysis. Since the flow-sensitive analysis we use is a staged analysis[14], the frequent object sets from the result of a less precise analysis (in this case flow-insensitive Andersen's analysis) can be used to approximate the frequent object sets of the flow-sensitive analysis.

We found out experimentally that, although the frequency (support) of the frequent object sets vary between the flow-insensitive analysis and the flow-sensitive analysis (this is expected as the two analyses compute results with different precisions), the frequent object sets themselves are almost the same. Table 2 shows the similarity between the most frequent object set of the flow-insensitive analysis and that of the flow-sensitive analysis for each benchmark. The similarity measure used here is the previously mentioned Jaccard similarity. As can be seen from the table, the similarity between maximal frequent object sets of a flow-sensitive and a flow-insensitive analysis is very high, greater than 90% in 9 out of our 10 benchmarks; in the other benchmark it is more than 75%

Benchmark	Jaccard Similarity
ex	1
176.gcc	1
nethack	0.99
254.gap	1
253.perlbmk	1
vim	0.98
python	0.93
svn	0.92
pine	0.95
gdb	0.75

Table 2. Similarity between the frequent object sets of the flow-insensitive and the flow-sensitive pointer analyses

Thus, we analyze results from the flow-insensitive analysis to find frequent object sets and use this as an approximation to merge the objects before the flow-sensitive analysis begins. The flow-sensitive algorithm itself remains as before, but now works on a modified input in which frequent object sets are replaced by their summary objects. We use the ECLAT [41] algorithm to determine frequent objects. An overview of our method is shown in Algorithm 1

Algorithm 1 High level description of Frequent Object Set Merging (FOSM)

```
1: procedure FREQUENTOBJECTSETMERGING
2:     Perform Andersen's flow-insensitive points-to
       analysis (first step in SFS)
3:     Use ECLAT to identify maximal frequent object sets
       in the results computed by Andersen's analysis
4:     Build data-flow graph, required as input to SFS
5:     For each frequent object set f, define a summary
       object o_f
6:     For each frequent object o ∈ f, replace o by o_f in
       the data-flow graph
7:     Run the main analysis of SFS using the
       modified data-flow graph as input
8: end procedure
```

4.1 Correctness

We now establish the correctness of FOSM. We claim that the points-to set computed by a flow-sensitive analysis after merging objects is a superset of the points-to set computed without merging, for each pointer in the program, at every program point.

Let o_1 and o_2 be two objects that are to be merged (which is done prior to the main analysis), into a summary object o'. This means that every reference to either o_1 or o_2 in the

program will be replaced[2] with o'. During the analysis, any object that would have been added to the points-to set of o_1 (or o_2) at a program point p, will now be added to the points-to set of o' at that program point. In other words, the points-to set of o' at a program point p will include the points-to sets of both o_1 and o_2 at that point.

A similar argument can also be applied to pointers ($top\text{-}level$ variables). Any pointer that would have pointed to either of o_1 or o_2 (or both) would now point to the summary object o'. This can be interpreted as the pointer including both o_1 and o_2 in its points-to set.

Last, since it may be unsafe to perform strong updates on abstract objects that may represent more than one concrete memory location (such as arrays and heaps)[10], we perform only weak updates on the merged summary object, as stated in the previous section.

4.2 Precision Loss

Here, we give examples to illustrate why merging memory objects into a single summary location can lead to loss in precision of the final points-to results.

1. Consider two pointers *p,q* for which a pointer analysis computes the points-to sets, respectively as $\{o_1\}$,$\{o_2\}$. Suppose we merge o_1 and o_2 into a single object, say o' before the analysis. The analysis would now compute both p and q as having the same points-to set $\{o'\}$. Thus the two pointers which didn't alias earlier now do.
2. Extending the previous example, suppose at a program point p_1, the points-to sets of o_1 and o_2 as computed by a pointer analysis are $\{o_3\}$ and $\{o_4\}$. Merging o_1 and o_2 would cause the merged object o' to have its points-to set as $\{o_3, o_4\}$. Thus, a use of either of o_1 or o_2 in a load/store will now propagate the combined points-to set, resulting in a loss of precision.

These examples show that (1) pointers that point to objects being merged will lose precision, and (2) points-to sets at each program point, of the objects being merged may lose precision.

4.3 Implementation

We build on Hardekopf's LLVM implementation of the staged flow-sensitive analysis [3]. After the first phase of the analysis (i.e., Andersen's flow-insensitive analysis), we use the points-to sets computed by it as input to the frequent itemset mining algorithm. We use a publicly available implementation[4] of the ECLAT algorithm to find frequent object sets in the points-to results of the flow-insensitive pointer analysis. This implementation[5] has many improvements over the original ECLAT algorithm[41] and can quickly compute maximal frequent itemsets even on some of the larger benchmarks we use.

Once the frequent object sets are determined, we modify the input to the main flow-sensitive analysis by merging frequently occurring object sets into single summary objects. The actual flow-sensitive analysis remains unmodified.

As a post-processing step, we compute the percentage of pointers that alias (see Section 5.3.1). Two pointers are said to alias if the intersection of their points-to sets is non-empty[18]. Checking all pairs of pointers in the program for possible aliasing is a time consuming process, especially for larger programs. We use a publicly available implementation of the Pestrie data structure[38] [5] and modify it to report the number of pointer pairs that alias. This data structure is designed for efficient persistent storage of pointer information. It also efficiently supports a number of queries on the stored pointer information, including aliasing between different pointers.

5. Experimental Results

In this section, we evaluate our FOSM method by comparing the execution time and precision of the results to the staged flow-sensitive pointer analysis[14].

We conducted our experiments on a set of 10 benchmarks, all of which have been used previously in pointer analysis research[14, 26]. Gcc, Perlbmk and Gap are some of the bigger benchmarks from SPEC2006[17]. Ex is a text editor, Nethack is a text based game, Vim is a text editor, Python is a language interpreter, Svn is a version control system, Pine is an email client and Gdb is a debugger.

We used an 8-core machine with 16GB of memory, running Debian GNU/Linux 6.0, to conduct these experiments. However, our flow-sensitive analysis runs as a single thread utilizing only one core in the system.

5.1 Performance

We first compare the execution time of flow-sensitive pointer analysis with frequent object set merging (FOSM) against the original staged flow-sensitive (SFS) analysis[14]. Table 3(a) shows the execution times (in seconds) for each benchmark. The table also lists the time taken by the frequent itemset mining algorithm (ECLAT). We add the time taken by the actual analysis after merging frequent object sets and the time taken by ECLAT when computing speedup w.r.t SFS; i.e., Speedup = SFS/(FOSM+ECLAT). As a comparison, we also give the time taken by Andersen's analysis for these benchmarks (shown as FI, column 2 in Table 3(a)).

Even smaller benchmarks such as `ex` and `176.gcc`, whose frequent object sets are not significantly large (see Table 1) show a speedup of at least 2x. Most of the bigger benchmarks (such as `pine`, `gdb`) show higher speedup ranging from 4x to 13x. `Python` and `pine` are two of the

[2] It may be desirable to keep track which object is replaced by which summary object in a real implementation, but it does not affect our discussion here.

[3] `http://www.cs.ucsb.edu/~benh/research/downloads.html`

[4] `http://www.borgelt.net/eclat.html`

[5] `https://github.com/richardxx/pestrie`

Benchmark	Time(s)				Speedup
	FI	SFS	FOSM	ECLAT	
ex	0.12	0.17	0.08	0	2.04
176.gcc	0.75	3.06	0.60	0.75	2.26
nethack	1.30	5.73	0.96	0.42	4.14
254.gap	3.52	19.71	0.76	1.03	10.98
253.perlbmk	4.57	95.57	6.32	5.2	8.29
vim	8.58	168.73	20.38	7.23	6.10
python	13.03	411.28	21.36	48.12	5.91
svn	16.64	4269.46	396.52	25.14	10.12
pine	41.99	6671.23	110.97	403.3	12.97
gdb	44.33	10591.46	653.66	335.9	10.70

(a)

Benchmark	$\frac{\text{FOSM}}{\text{SFS}}$ %
ex	65.29%
176.gcc	46.74%
nethack	51.44%
254.gap	84.54%
253.perlbmk	80.17%
vim	87.88%
python	67.32%
svn	74.52%
pine	65.99%
gdb	65.78%

(b)

Table 3. (a) Speedup in running time of the analysis with frequent object set merging (b) Fraction of worklist items processed after frequent object set merging

larger benchmarks where the time taken by ECLAT (column 4 in Table 3(a)) is more than the time for the actual analysis (column 3). While this limits the speedup to 5.91x in `python`, merging the large frequent object set (containing 22053 objects) in `pine` brings down its actual analysis time significantly, enabling a good speedup. Overall, we see an average (geomean) speedup by a factor of 6.2x for our FOSM method over the state-of-the-art flow-sensitive pointer analysis method (SFS). It should be noted that this speedup includes the overhead of computing the frequent object sets.

5.2 Convergence

By approximating many memory locations (objects) with a single summary location, we not only reduce propagation of large points-to sets, but also achieve faster convergence of the analysis. Table 3(b) shows the number of worklist items processed after merging frequent object sets, as a fraction of the number of worklist items processed without the approximation. On an average (geomean), about 33% less worklist items are processed after merging frequent object sets.

5.3 Precision

In this section, we study the effects of merging memory objects on the precision of the analysis. We consider two client analysis queries and study the impact of precision loss on them.

5.3.1 Alias Pairs

A typical use of pointer analysis is to compute alias pairs in the program[18]. Two pointers are said to alias if the intersection of their points-to sets is non-empty. In this experiment, we consider all pairs of pointers in the program and test how many of them alias.

We show the results in terms of the percentage of possible pointer pairs that may alias. This is a commonly used precision measure for points-to analysis[4, 9, 30]. Higher the percentage, lower is the precision of the analysis. The percentage of pointer pairs that alias for both our analysis (FOSM) as well as Hardekopf's fully flow-sensitive staged analysis (SFS) [14] are shown in Table 4(a). To present a complete picture, we also provide the corresponding precision data (shown as FI, column 2 in Table 4(a)) for flow-insensitive (Andersen's) analysis.

While some benchmarks such as `176.gcc` and `pine` show very little precision loss, `python` and `253.perlbmk` incur relatively higher precision loss. These benchmarks have a higher percentage of their objects (29.4% and 25.5% respectively) occurring with a higher frequency (30.5% and 20.3%), indicating that a larger number of frequent objects were merged. Even in these benchmarks, the precision loss is within 5%.

5.3.2 Dependent Load/Store

Some program analysis techniques such as Taint Analysis [36] and Loop Invariance Detection[39] require the knowledge of conflicting (dependent) loads and stores in the program. Any load or store that may refer to the same memory object as a given store can be considered to be in conflict with the given store. Similarly, stores that may refer to the same memory object as a given load can be considered to be in conflict with the given load.

To study the effects of precision loss due to our approach, we compute the percentage of conflicting load/store pairs in each function of the program. Higher the percentage of conflicting pairs, lower is the precision of the analysis. We again compare the precision of our analysis (FOSM) with Hardekopf's fully flow-sensitive staged analysis (SFS) [14]. The results are shown in Table 4(b)

The precision loss characteristics here are similar to what we observed when using *Alias Pairs* as a precision measure, although the actual numbers vary. `Ex`, `176.gcc` and `254.gap` have very low precision loss while `253.perlbmk`, python and svn have relatively higher precision loss (upto 7%) compared to the others. This is again due to a higher percentage of their objects occurring with a higher frequency. Note that the actual numbers used to measure precision here are more than what we saw when using *Alias Pairs* as a measure since the candidates for conflicting loads/stores consid-

Benchmark	% Alias			%Loss (FOSM-SFS)
	FI	SFS	FOSM	
ex	6.13	1.72	1.88	0.15
176.gcc	20.49	4.73	4.74	0.01
nethack	6.90	2.81	2.86	0.05
254.gap	26.58	13.38	13.83	0.45
253.perlbmk	36.17	18.58	22.26	3.67
vim	12.10	2.27	3.44	1.16
python	37.62	9.71	14.22	4.50
svn	28.8	7.47	7.65	0.18
pine	36.01	12.16	12.23	0.07
gdb	35.47	12.74	13.05	0.31

(a)

Benchmark	Ld/St Conflict (%)		%Loss
	SFS	FOSM	
ex	9.59	9.77	0.17
176.gcc	7.75	7.76	0.01
nethack	14.28	16.68	2.39
254.gap	23.48	23.84	0.36
253.perlbmk	29.81	36.28	6.46
vim	30.99	34.59	3.60
python	20.86	25.53	4.66
svn	21.22	28.18	6.96
pine	8.99	12.32	3.33
gdb	14.69	19.74	5.05

(b)

Table 4. (a) Percentage of pointer pairs that alias. (b) Percentage of conflicting load/stores in each function.

ered here are local to a function, while in measuring *Alias Pairs*, we considered all pointers in the program.

5.4 Choice of objects to be merged

In addition to the experiments in the previous two subsections, we also tried to merge the frequent object sets as computed by the flow-sensitive analysis, to measure the loss that could happen because of our use of a flow-insensitive analysis to approximate the frequent object sets. In other words, if we had oracle knowledge of frequent objects sets in the flow-sensitive pointer analysis, would the precision loss reduce? We found that using the frequent object sets from the final flow-sensitive analysis (and feeding it back to the frequent itemset mining algorithm in a subsequent run) does not result in any significant improvement in precision

Selecting a subset (which may have a higher frequency of occurrence) of the maximal frequent object sets as a candidate for merging may reduce precision loss further. However, determining the right subset may require iterating through all the subsets. We leave this exploration for future work.

6. Related work

Although a few earlier works in pointer analysis have performed merging of pointers and/or objects, none of these works, to the best of our knowledge, have used it to approximate flow-sensitive pointer analysis.

Most earlier work[13, 33] that perform some kind of merging of nodes, do so only when it is guaranteed that the merging will not cause a precision loss. These algorithms look for variables with identical points-to (or pointed-to-by) sets, by searching the points-to graph for cycles. Combining such variables that form a cycle into a single new variable during the analysis will not result in a precision loss.

The first attempt at approximating Andersen's (flow-insensitive) analysis via merging variables, at the cost of precision loss was by Nasre[28]. This work allowed merging of two variables based on them having *similar* points-to or pointed-to-by sets. A known similarity measure, called the Jaccard similarity[31] was used. The disadvantage with this method is that the quality of the final result depends on the order of similarity checks and merging of objects. Also, repeated checks for similarity during the analysis is an overhead. In our work, we do not incrementally merge objects based on similarity. Instead, we determine the objects to be merged beforehand using a frequent itemset mining algorithm[31, 41] and merge objects before the actual analysis begins. We take advantage of the staged flow-sensitive analysis by using results from the first stage (flow-insensitive pointer analysis) to determine frequent object sets for the second stage (flow-sensitive pointer analysis). Another advantage of performing object merging prior to the main analysis is that it decouples resource optimization from the main analysis, making it independent of the implementation of the main analysis.

The paper by Hasti and Horwitz[15] talks about an algorithm to improve precision of flow-insensitive analysis by performing repeated rewrites into the SSA form[7]. They however do not comment on whether the method described will achieve the precision of a flow-sensitive analysis. This question has also been raised in more recent papers[14, 23], but without an answer. We answer the question with the following example, which shows that their algorithm will not always achieve full flow-sensitive precision. In Figure 3, a in block 2 can only point to x. However, Hasti and Horwitz's algorithm would compute that a may point to either of x or y. This is because block 2 and block 3 depend on each other to become precise.

Other methods that were used to approximate flow-sensitive pointer analysis in the past include using probabilistic data structures[29] and performing strong updates over a flow-insensitive analysis to get a precision in between that of flow-sensitive and flow-insensitive analyses[23].

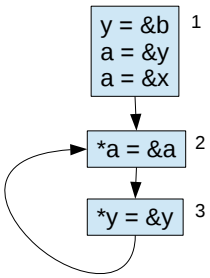


Figure 3. Simple example to illustrate imprecision of Hasti and Horwitz's algorithm

7. Conclusion

In this work, we studied the structure of points-to sets of pointers in a flow-sensitive pointer analysis and identified the frequent occurrence of certain object sets in these points-to sets. We used the results from a less-precise analysis to determine these frequent object sets as an approximation to the frequent object sets in a flow-sensitive analysis. We showed that these frequent object sets can be merged into a single summary location at the cost of some precision loss. Such a merging of frequent object sets leads to an average (geomean) improvement by a factor of 6.2x in the execution time of flow-sensitive pointer analysis.

Acknowledgment

We thank Ben Hardekopf, Christian Borgelt and Xiao Xiao for making their implementations of the staged flow-sensitive pointer analysis, Eclat and Pestrie algorithms respectively, public. We also thank members of the HPC lab at Indian Institute of Science for their support.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, 1994. ISBN 1-55860-153-8.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

[3] L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[4] S. Blackshear, B.-Y. E. Chang, S. Sankaranarayanan, and M. Sridharan. The flow-insensitive precision of andersen's analysis in practice. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 60–76, 2011. ISBN 978-3-642-23701-0.

[5] C. Borgelt. Efficient implementations of apriori and eclat. In *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL). CEUR Workshop Proceedings 90*, page 90, 2003.

[6] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7. . URL `http://doi.acm.org/10.1145/1455770.1455778`.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991. ISSN 0164-0925.

[8] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 35–46, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. . URL `http://doi.acm.org/10.1145/349299.349309`.

[9] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, 2001.

[10] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 246–266, 2010. ISBN 3-642-11956-5, 978-3-642-11956-9.

[11] P. Ferrara. A fast and precise alias analysis for data race detection. In *Proceedings of the Third Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'08)*, volume Electronic Notes in Theoretical Computer Science. Elsevier, April 2008.

[12] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. In *Science of Computer Programming*, 2005.

[13] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 290–299, 2007. ISBN 978-1-59593-633-2.

[14] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, 2011.

[15] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 97–105, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. . URL `http://doi.acm.org/10.1145/277650.277668`.

[16] L. Hendren. Context-sensitive points-to analysis: Is it worth it. In *Compiler Construction, 15th International Conference, volume 3923 of LNCS*, pages 47–64. Springer, 2006.

[17] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006. ISSN 0163-5964.

[18] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engi-*

neering, PASTE '01, pages 54–61, 2001. ISBN 1-58113-413-4.

[19] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 113–123, 2000. ISBN 1-58113-266-2.

[20] S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, Jan. 1997. ISSN 0164-0925.

[21] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 27–36, 2006. ISBN 1-59593-374-3.

[22] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009. ISBN 0849328802, 9780849328800.

[23] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, 2011. ISBN 978-1-4503-0490-0.

[24] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 343–353, 2011. ISBN 978-1-4503-0443-6.

[25] L. Li, C. Cifuentes, and N. Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 85–96, 2013. ISBN 978-1-4503-2100-6.

[26] M. Méndez-lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *In Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA10*, 2010.

[27] V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 19–28, 2013. ISBN 978-1-4799-1021-2.

[28] R. Nasre. Approximating inclusion-based points-to analysis. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 66–73, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0794-9.

[29] R. Nasre. Time- and space-efficient flow-sensitive points-to analysis. *ACM Trans. Archit. Code Optim.*, 10(4):39:1–39:27, Dec. 2013. ISSN 1544-3566.

[30] R. Nasre, K. Rajan, R. Govindarajan, and U. P. Khedker. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 47–62, 2009. ISBN 978-3-642-10671-2.

[31] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011. ISBN 1107015359, 9781107015357.

[32] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, Sept. 1994. ISSN 0164-0925.

[33] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 47–56, 2000. ISBN 1-58113-199-2.

[34] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programmin g*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM. ISBN 1-58113-346-4. . URL `http://doi.acm.org/10.1145/379539.379553`.

[35] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, 1996. ISBN 0-89791-769-3.

[36] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, 2009.

[37] S. Verdoolaege and T. Grosser. Polyhedral extraction tool.

[38] X. Xiao, J. Zhou, C. Zhang, and Q. Zhang. Persistent pointer information. In *Proceedings of the 35th ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[39] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 738–763, 2012.

[40] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 218–229, 2010. ISBN 978-1-60558-635-9.

[41] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical report, Rochester, NY, USA, 1997.