EagerMerge: An Optimistic Technique for Efficient Points-To Analysis

Sudhir Samrit
IIT Madras, India
sudhir@cse.iitm.ac.in

Rupesh Nasre
IIT Madras, India
rupesh@cse.iitm.ac.in

ABSTRACT

We present an information-merging technique for efficient computation of points-to information for C programs. Invalid use of pointers can lead to hard-to-find bugs and may expose security vulnerabilities. Thus, analyzing them is critical for software analysis as well as optimization. Pointer analysis is a key step during compilation, and the computed points-to information is useful for client analyses from varied domains such as bug finding, data-flow analysis, identifying security vulnerabilities, and parallelization, to name a few. Former research on pointer analysis has indicated that the main bottleneck towards scalability is large propagation of points-to information in the *constraint graph*. To reduce the propagation cost, we present a technique based on temporal similarity of points-to sets. The method tracks pointers whose dynamically changing points-to information remains equal *for a while*. Based on the optimism gained by observing the points-to sets over time, the analysis decides to merge the corresponding nodes. Using the notion of merge and split, we build a family of points-to analyses, and compare their relative precisions in the context of existing analyses. We illustrate the effectiveness of our approach using a suite of sixteen SPEC 2000 benchmarks and three large open-source programs, and show that the technique improves the analysis time over BDD and bitmap based Hybrid Cycle Detection, well-known Andersen's analysis, and Deep Propagation, affecting minimal precision (precision is 96.4% on an average). Specifically, it is faster than Deep Propagation by 45%.

CCS Concepts

•Software and its engineering → Compilers; General programming languages;

Keywords

constraint graph; points-to analysis; flow-insensitive analysis; merging; approximation

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931045

1. INTRODUCTION

Points-to analysis is a method to statically find out if two pointers in a program may point to the same memory location at runtime. If they do, then the two pointers are said to be aliases of each other.

As the software size and life grow, bugs do creep in. Pointers often provide a conducive environment for bugs to appear and spread in C/C++ programs [16], despite careful coding and reviews, such as in the Linux kernel [35, 1]. Analyzing pointers is a key step to identifying bugs and security vulnerabilities in industry-size C/C++ programs. The points-to information computed by a pointer analysis bears potential to directly affect the effectiveness of client analyses such as array-bounds checking, null pointer analysis and slicing, apart from the traditional optimizations like common subexpression elimination. Due to its importance, a substantial number of points-to analysis algorithms have been proposed in literature [2, 40, 5, 3, 43, 17].

Background. For analyzing a general purpose C/C++ program in a flow-insensitive manner, it is sufficient to consider all the pointer statements of the following forms: address-of assignment (`p = &q`), copy assignment (`p = q`), load assignment (`p = *q`) and store assignment (`*p = q`) [33]. Load and store assignments are also referred to as *complex assignments*. A heap allocation (such as `malloc`) is represented using an address-of assignment. A flow-insensitive analysis iterates over a set of points-to constraints until a fixed-point is obtained. Typically, the flow of points-to information is represented using a constraint graph `G`, in which a node denotes a pointer variable and a directed edge from node n_1 to node n_2 represents propagation of points-to information from n_1 to n_2. Each node is initialized with the points-to information computed by evaluating the *address-of* constraints. Edges are added to `G` initially by *copy* constraints and then by *complex* (*load* and *store*) constraints as the analysis progresses. This is because the edges introduced by *complex* constraints depend upon the availability of points-to information at nodes which, in turn, depends upon the propagation. In effect, evaluation of complex constraints and propagation of points-to information are cyclically dependent. Thus, as the analysis performs an iterative progression of the points-to information propagation, new edges get introduced in `G` due to the evaluation of the complex constraints, resulting in the computation of more and more points-to information at its nodes. When no more edges can be added and no more points-to information can be computed, `G` gets stabilized and a fixed-point (points-to information at the

Algorithm 1 Points-to Analysis using Constraint Graph

Require: set C of points-to constraints
1: Process address-of constraints
2: Add edges to constraint graph G using copy constraints
3: **repeat**
4: Propagate points-to information in G
5: Add edges to G using load and store constraints
6: **until** fixed-point

	#Points-to sets		
Benchmark	Total	Unique	Unique%
164.gzip	1595	298	18.7
175.vpr	8922	1170	13.1
176.gcc	135628	6987	5.2
177.mesa	32880	1471	4.5
179.art	586	109	18.6
181.mcf	1230	56	4.6
183.equake	1317	213	16.2
186.crafty	6126	1363	22.2
188.ammp	7852	490	6.2
197.parser	13913	978	7.0
252.eon	65265	15573	23.9
253.perlbmk	60486	3129	5.2
254.gap	49531	1956	3.9
255.vortex	26221	5141	19.6
256.bzip2	1147	212	18.5
300.twolf	17843	938	5.3
httpd	133893	19688	14.7
sendmail	74622	12491	16.7
wine	62387	9019	14.5
Total	701444	81282	**11.6**

Figure 1: Total versus unique points-to sets

nodes) is reached. The information can then be used by various clients (e.g., for program understanding, identifying security vulnerabilities, parallelization, etc.). An outline of this analysis is given in Algorithm 1.

Motivation. Majority (over 70%) of the analysis time is spent in propagating information in G (Line 4). Thus, techniques have been developed for efficient propagation of points-to information across the edges of a constraint graph. Online cycle elimination [8] detects cycles in G on-the-fly and collapses all the nodes in a cycle into a representative node. Cycle collapsing is possible because all the nodes in a cycle eventually contain the same points-to information. This significantly reduces the number of pointers tracked, avoids unnecessary propagation across them, and speeds up the overall analysis. Wave and Deep Propagation [33] techniques perform a topological ordering of the edges and propagate only the difference in the points-to information in breadth-first or depth-first manner respectively. These propagation orders significantly improve the points-to analysis time by avoiding repeated propagations in the graph.

Existing optimizations focus on the current *structural* aspects of the constraint graph, while they fail to consider the *values* of points-to sets and their *evolution* during the analysis. Thus, the existing techniques exploit structural patterns (cycles [8], topological ordering [33], dominators [26]) in the current intra-iteration snapshot of the evolving constraint graph and ignore how points-to information changes across iterations. We make a crucial observation that exploiting the evolution of the points-to information can provide significant opportunities for efficient points-to analysis. To substantiate our claim, we list in Figure 1 the number of unique points-to sets at the end of the analysis for the suite of programs in our test harness. We observe that although the number of different pointers is large, the number of unique points-to sets across them is considerably small (11.6% on an average). This indicates a lot of similarity across points-to sets of variables. In other words, several pointers are *pointer-equivalent*. If we could identify such pointers early in the analysis, we can merge the corresponding nodes, propagate less information, and improve overall analysis efficiency.

Unfortunately, it is not a priori predictable when two arbitrary nodes in the constraint graph would be pointer-equivalent, that is, whether they would attain the same points-to sets. Existing techniques do exploit such a fact for specialized structures such as cycle nodes, dominators, etc., where the nodes are guaranteed to have the same points-to information. However, this is not possible for arbitrary pointers in the constraint graph. Moreover, even if two nodes contain the same information at some point during the analysis, there is no guarantee that they would remain pointer-equivalent throughout the analysis. To preserve precision, the analysis would be forced to split the merged nodes later in the analysis.

In this work, we consider the temporal behavior of points-to information for improving propagation without splitting the nodes. In particular, our analysis tracks pairs of pointers (p, q) whose points-to information becomes the same in some iteration i, and then continues checking if their points-to information remains the same in iterations i..i+K. If this condition is met, that is, pointers p and q have the same points-to information for a threshold K iterations, then the two nodes are merged. A key practical aspect of our solution is that once merged, nodes need not be split again. Merging nodes allows our analysis (i) to track less number of pointers, and (ii) to propagate less information in the constraint graph, leading to an overall time- and memory-efficient analysis in practice.

Our contributions are summarized below.

- We study the points-to information computed by the existing inclusion-based analyses, and observe considerable duplication across points-to sets of various pointers. Exploiting this observation, we propose K-Merge, a technique for efficient points-to analysis, which tracks temporal evolution of points-to sets and merges the pointer nodes whose points-to sets remain the same for a threshold duration during the analysis. We build a family of analyses based on merging and splitting of the nodes, and compare their relative precisions in the context of existing pointer analyses.
- We propose EagerMerge, an efficient inclusion-based points-to analysis algorithm exploiting K-Merge. We describe efficient data structures for tracking pointer-pairs across K iterations. Theoretically, the algorithm adds approximations to the computed information, but in practice, we observe that the approximations are insignificantly small for most of the benchmarks (average 3.7%) in our experimental setup.
- We perform detailed experimental evaluation of the proposed algorithm using sixteen SPEC 2000 C/C++

benchmarks and three large open-source programs (namely, *httpd*, *sendmail* and *wine*). Our points-to analysis runs 45% faster than Deep Propagation [33] and consumes 28% less memory compared to it.

2. EAGERMERGE USING AN EXAMPLE

Consider the following set of constraints obtained from a C program. This program is a modified version of an example provided by Pereira and Berlin's set of programs [32].

```
a = &b; h = &j; f = &h; d = &a; i = &e; b = &h;
d = f; b = e;
c = *d; e = *f; *c = e; g = *c; *g = d; *i = f; *e = c;
```

Existing Analysis. The running of Algorithm 1 (Andersen's analysis or Deep Propagation) on the above example is depicted in Figure 2. Each subfigure shows the state of the constraint graph in an iteration. Each circular node represents a pointer in the program, and each directed edge represents flow of points-to information. Each node maintains its points-to set (denoted as a rectangular box). At a high level, a few points can be noted from the figure. First, the number of nodes remains fixed across iterations. Second, the number of edges goes on increasing – thus, points-to analysis is being modeled as an incremental graph algorithm. Third, points-to set of each pointer monotonically increases.

We now explain the evolution of the constraint graph in detail. For the sake of exposition, we assume that no cycle detection is being done in this example, but our technique works seamlessly in the presence of cycle detection. Initially, nodes receive points-to information from address-of constraints (Line 1 of Algorithm 1). This is denoted as Iteration 0 in Figure 2(a). The constraint graph at this stage also contains initial set of edges via copy constraints d = f; b = e, that is, f → d and e → b (Line 2 of Algorithm 1).

In Iteration 1 as shown in Figure 2(b), points-to information is first propagated across the edges which is unified with the current information of the target node (Line 4 of Algorithm 1). This updates points-to information of node d to {a, h}. Then, more edges are added to the constraint graph using complex constraints (Line 5), such as edge $a \rightarrow c$.

This processing is continued in further iterations (Figures 2(c)–2(f)). In each iteration, points-to information is propagated across all the edges, and more edges are added using complex constraints. Andersen's analysis [2] does not demand a specific propagation order, whereas Deep Propagation [33] propagates information in depth-first order.

In Iteration 6 (not shown), no new edges are added and no new propagation of points-to information takes place indicating a fixed-point. The final points-to information is computed for each pointer as shown along with its corresponding node in the constraint graph. For instance, pointer e points to {a, b, h, j} and pointer i points to a singleton {e}.

We make two observations from the above example. First, several pointers have similar points-to information at the fixed-point (which is also true for real-world programs as shown earlier in Figure 1). For instance, pointers a, b, c, e, g, h, j have the same points-to information (Figure 2(f)). Therefore, it makes sense to merge these nodes to reduce redundant propagation. Former approaches such as cycle detection [8] and dominators [26] target identifying *structural* patterns to merge such nodes. EagerMerge works with *values* of points-to sets instead, to merge nodes. The second observation is that several of these pointers have the same points-to information much earlier in the analysis. For instance, pointers b and e have the same points-to information {h, j} in Iteration 2, while pointers b, c, e, g, h, j are pointer-equivalent in Iteration 3. EagerMerge exploits this property to merge the nodes early to reduce propagation.

Proposed Analysis. The same example with EagerMerge is depicted in Figure 3. The constraint graph is defined in a similar manner, except that now each node in the constraint graph represents a unique set of pointers (rather than a single pointer in the earlier case). Initially, each such set is a singleton representing one pointer each (Figure 3(a)). In Iteration 1, first points-to information is propagated across edges. This adds pointee h to the points-to set of d. At this stage, we can notice that the points-to sets of both b and f are the same {h}. So, EagerMerge merges the two nodes, resulting in a new node bf (Figure 3(b)). All the incoming and outgoing edges of b and f are now updated to be incident on and from the merged node bf. The last step of Iteration 1 adds subset edges in the constraint graph using complex constraints. The constraint graph now contains additional edges bf→e, a→ c, h→c, h→e. Interestingly, the original edge f → d now represents a spurious edge b → d, due to merging of b and f (shown as a red bold edge). This spuriousness leads to imprecision in EagerMerge, as one can see from Figure 2(f) that in the original final constraint graph, no such edge exists and that the points-to information of d is not a superset of that of b. However, in EagerMerge, d now contains spurious points-to information.

In Iteration 2, once again the points-to information is propagated across edges (Figure 3(c)). This updates pointsto(e) to {h, j}, pointsto(c) to {b, j}, pointsto(d) to {a, h, j} and pointsto(bf) to {h, j}. At this stage, EagerMerge can merge node e into bf, due to their same points-to information {h, j}, creating a new merged node bef. Merging reduces the number of nodes, as well as (often) the number of edges in the constraint graph, making it smaller. This reduces the memory requirements of the analysis (discussed in Section 4.3). Next, complex constraints add edges, and another spurious edge j → f gets added (shown as a red bold edge), which is not present in the original constraint graph (Figure 2(f)).

In Iteration 3, propagation of points-to information in the graph makes g and j pointer-equivalent to bef. Similarly, c becomes pointer-equivalent to h (Figure 3(d)). The corresponding eager merging reduces the number of nodes in the constraint graph to five, with a reduced set of edges. Complex constraints add more edges at the end of the iteration.

In Iteration 4, cycles across ch, befgj and d make their points-to information same, resulting in merging of these nodes to create a monolithic node bcdefghj with each of them pointing to the points-to set {a, b, h, j} (Figure 3(e)).

Finally, in Iteration 5, node a gets merged with the monolithic node and the computation reaches a fixed-point. The final EagerMerge constraint graph (EMCG) contains two nodes and zero edges, compared to the traditional constraint graph which finally contains ten nodes and twenty edges. Despite a smaller size, in effect, EMCG is a supergraph of the traditional constraint graph. Both the graphs contain logically the same number of nodes (possibly merged in EMCG), but the number of edges and the points-to information in EMCG could be larger. Therefore, EMCG does

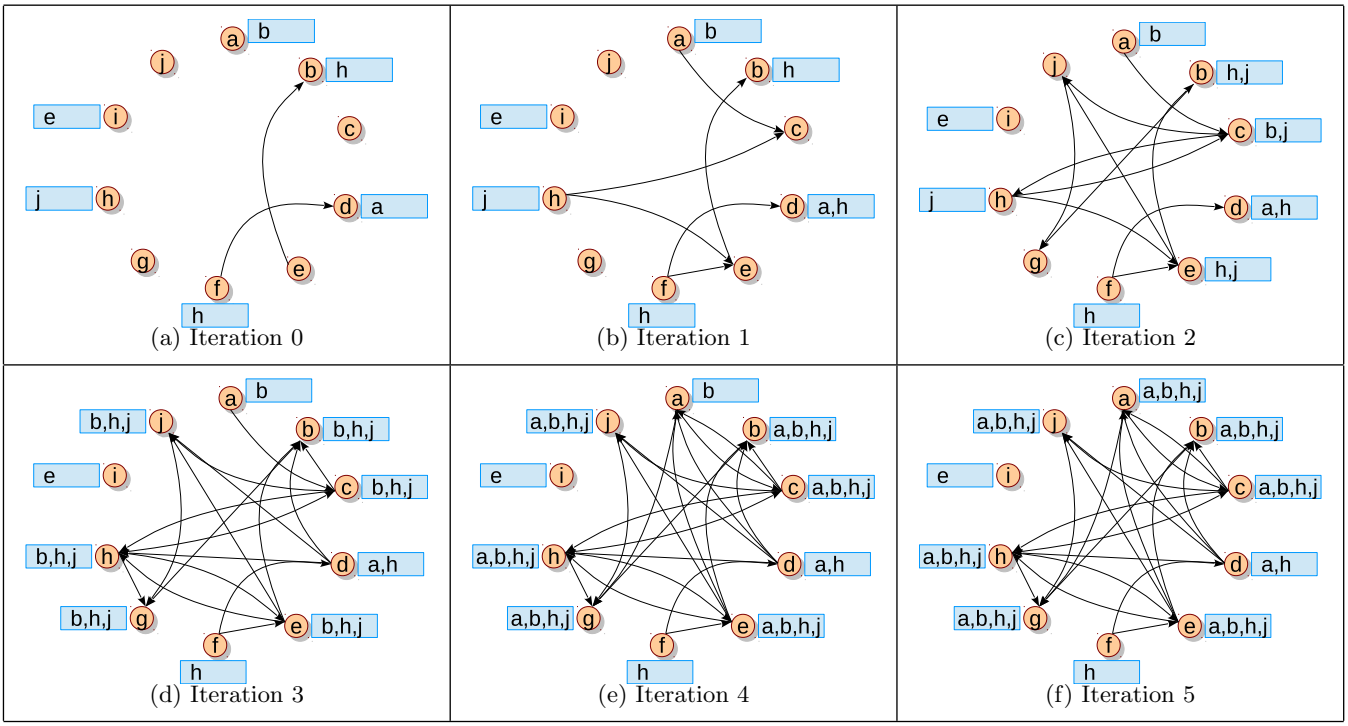

Figure 2: Deep Propagation constraint graph in each iteration for the example in Section 2

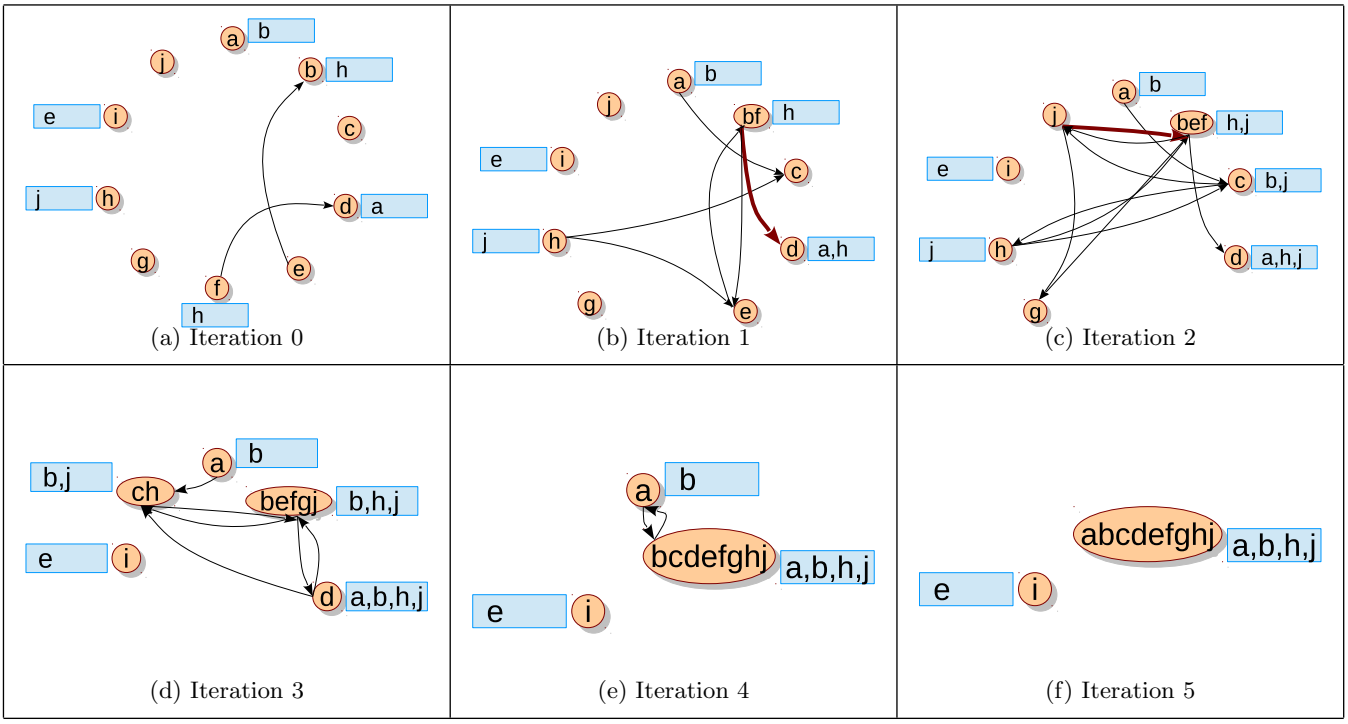

Figure 3: EagerMerge constraint graph in each iteration for the example in Section 2

not lose any information computed by an inclusion-based analysis – and is thus, sound. However, EMCG may compute a superset of points-to information, and hence may lose precision compared to other analyses. For instance, the following points-to facts were spuriously generated by EMCG: pointsto(d)={b,j} and pointsto(f)={a,b,j}. Thus, in this case, EMCG computes a more approximate solution compared to the original analysis. Total number of points-to pairs in the original analysis is 32, whereas in EMCG it is 37, resulting in a precision loss of (37-32)/32 = 15.6%. In practice, we observe that the precision loss due to EagerMerge is negligibly small for most of the benchmarks (Section 4). Thus, EM proves to be a practical technique for efficient points-to analysis.

3. POINTS-TO ANALYSIS USING EM

In this section we first formally define the family of K-Merge analyses based on merging and splitting, and instantiate EagerMerge as a specific analysis in the family. Next, we explain efficient data structures for tracking merged nodes and processing nodes to be split. Later, in Section 3.3, we present the points-to analysis algorithm using K-Merge.

3.1 Family of Analyses

Formally, we define a family of analyses $\mathcal{A}$ where each points-to analysis algorithm $A(K_{merge}, K_{split}) \in \mathcal{A}$ is parameterized with two arguments: K_{merge} denotes the number of iterations for which two nodes remain pointer-equivalent before getting merged, while K_{split} denotes the number of iterations for which a node is not split after any pointer in the merged node ceases to be pointer-equivalent with the other nodes in the group. Andersen's analysis [2] is $A(\infty, -)$, that is, the analysis never merges nodes, and therefore, does not need to consider splitting. An analysis that merges pointer equivalent nodes in a cycle may be viewed as $A(0, \infty)$ as it never splits the nodes again.

A value of zero for K_{split} indicates lossless analysis, as it never allows non-pointer-equivalent nodes to be in a merged state (the analysis may lose precision based on other decisions, but not due to merging). In general, a higher value of K_{split} indicates more precision loss. On the other hand, a higher value of K_{merge} indicates better confidence that the nodes are pointer-equivalent (but no guarantee). Therefore, the value of K_{merge} influences the selection of K_{split}.

Based on K_{merge} and K_{split}, an analysis A computes points-to information. We define a partial order on these analyses based on the computed information which is based on the parameters. So, if $I_{merge} \geq J_{merge}$ and $I_{split} \leq J_{split}$, then $A(I_{merge}, I_{split}) \leq A(J_{merge}, J_{split})$. The top of the associated lattice is $A(\infty, 0)$ which corresponds to the precision of Andersen's analysis, while the bottom is $A(0, \infty)$.

In this work, we present EagerMerge, which is characterized as $A(1, \infty)$. Thus, if two pointers are pointer-equivalent for an iteration, we merge them and never split them again. Since $K_{split} > 0$, there is a possibility of information-loss. We could have reduced the value of K_{split}, but the cost of splitting is high in practice for the small precision gain on an average. In our experimental evaluation (Section **??**), we find that avoiding the split is reasonable for most benchmarks. For instance, EM incurs a precision loss of less than 5% for 17 out of 19 benchmarks. However, in precision-critical environments, an analysis designer may choose a smaller value of K_{split} to regain precision.

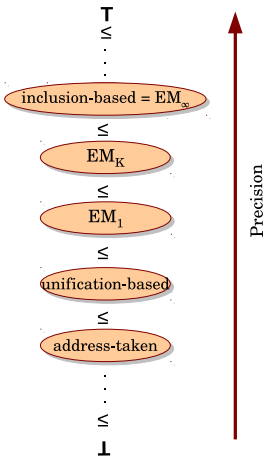

Figure 4: Precision hierarchy for various analyses

3.2 Relative Precisions of Analyses

We now compare the relative precision of K-Merge in the context of other existing analyses. Figure 4 shows several analyses ordered based on their precision (denoted using partial order $\leq$), with precision increasing bottom-up. The bottom element (denoted as $\perp$) is the least precise analysis wherein any pointer may point to any other pointer (or variable). The top element (denoted as $\top$) is the most precise analysis, which one wants to achieve, but is undecidable in general. The precision of K-Merge (denoted as EM_K in Figure 4) is between that of inclusion-based and unification-based analyses. In inclusion-based analysis, for a statement p = q, we maintain the subset constraint pointsto(p) $\supseteq$ pointsto(q). On the other hand, in unification-based analysis, we merge the points-to sets of p and q. EM_K logically performs in between: it performs unification for a subset of the pairs. Hence its precision lies between those of inclusion-based and unification-based analyses. When no merging is done, K-Merge approaches the precision of inclusion-based analysis, denoted by EM_∞.

3.3 Points-to Analysis Algorithm

Algorithm 2 presents points-to analysis using K-Merge for arbitrary K. It uses a data structure called *PEPTable* for efficient insertion and removal of pointer pairs (discussed in Section 3.4). The algorithm starts with processing address-of and copy constraints similar to the traditional inclusion-based analysis. During the fixed-point computation (`repeat-until` loop at Lines 4–18), the constraint graph G_{EM} is saturated by propagating points-to information across edges. For each such active edge (through which new points-to information was propagated), K-Merge checks if the points-to information of the end-points is the same (Line 7), and if so, then the pair of end-points is added to the PEPTable if not already present. PEPTable adds an entry into a bucket 0..K-1 based on the current iteration number. On the other hand, if the two end-points of the active edge have different points-to information, then the pair is removed from PEPTable if already present. Next, each pair of pointers that existed in PEPTable for K iterations enables merging. The algorithm merges such nodes in the constraint graph G_{EM} (`for` loop at Line 13). After

Algorithm 2 Points-to Analysis using K-Merge

Require: set C of points-to constraints
Require: counter K as a threshold number of iterations
1: Process address-of constraints
2: Add edges to constraint graph G_{EM} using copy constraints
3: iteration = 0
4: **repeat**
5: Propagate points-to information in G_{EM}
6: **for** each active edge $u \rightarrow v$ **do**
7: **if** $pointsto(u) = pointsto(v)$ **then**
8: PEPTable.add(u, v, iteration mod K)
9: **else if** PEPTable.exists(u, v) **then**
10: PEPTable.remove(u, v)
11: **end if**
12: **end for**
13: **for** each pair (u, v) in PEPTable[(iteration + 1) mod K] **do**
14: G_{EM}.merge(u, v)
15: **end for**
16: Add edges to G_{EM} using load and store constraints
17: ++iteration
18: **until** fixed-point

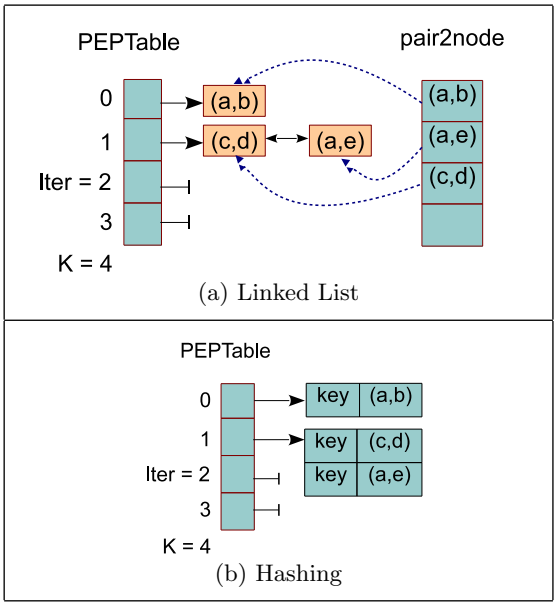

Figure 5: Efficient data structures for EagerMerge

this step, similar to the traditional constraint graph, new edges are added to G_{EM} using complex constraints. This processing continues until no more edges can be added.

Currently, as shown in Algorithm 2, the K-Merge processing is performed after points-to information propagation and before adding new edges. An implementation has the choice of placing the K-Merge processing (Lines 6–15) either before propagation (Line 5) or after adding edges (Line 16) as well. However, the effectiveness of K-Merge is better if information propagation has taken place in G_{EM} – this allows K-Merge to not only add pointer-equivalent pairs to PEPTable but also to remove non-pointer-equivalent pairs from it. Further, if merging takes place prior to edge-addition, then the number of added edges gets reduced, decreasing the overall cost of the graph updates.

3.4 Data Structures

Efficient implementation of K-merge, K-split requires careful consideration of data structures. For instance, if K-merge is 3, then we need to be able to track pairs of pointers (merged nodes) for 3 iterations: say from i to $i + 2$. At the same time, there could be other pairs which start being pointer equivalent in iteration j, so we need to track them from j to $j + 2$. We found it non-trivial to process pointer pairs efficiently during our implementation. Specifically, following questions arise during the data structure design.

Q1. In each iteration, the number of iterations for each pointer-equivalent pair needs to be incremented. How can this be done efficiently?

Q2. Given a pair, how to efficiently remove it from a bucket?

A naïve way to address Q1 is to process all tracked pairs in each iteration. However, this is quite expensive – work done for *each pair* is $O(K)$ across K iterations. A better way is to group all pointer-equivalent pairs from each iteration and to update their iteration number together. This is efficiently implementable by tracking pairs into iteration-buckets, and assigning iteration numbers to each bucket instead of to each pair. This improves upon the per-pair processing and requires $O(K)$ work *per bucket* across K iterations. Since the number of buckets is bounded by K, total work is $O(K^2)$. We improve upon this further to $O(K)$ total work across K iterations by noting that an increment of an iteration implies increment of the previous iteration by two, and increment of the iteration previous to it by three, and so on. This is possible by maintaining a circular list (or array) of buckets and a pointer (or offset) to an element denoting the current iteration. By using the counter (that is, the current iteration), we can know the number of iterations for which a bucket continued to contain pointer-equivalent pairs. We call this data structure as pointer-equivalent pairs (PEP) table. Each bucket in the table can be implemented as a (doubly) linked list or using hashing.

Linked List. The linked list implementation of bucket in PEPTable is depicted in Figure 5(a). Q2 arises when in some iteration, two pointers stop being pointer-equivalent before K iterations (due to propagation of points-to information), and we need to remove the corresponding pair from PEPTable. For an efficient access, we need a map (denoted as *pair2node* in Figure 5(a)) from pair (p_1, p_2) to a pointer to the pair stored in PEPTable. Using this pointer, we remove the node pair from the linked list. On a merge of pointers p and q, a pair (p, q) is created and added to *pair2node*, and inserted into the bucket for the current iteration. Performance considerations prohibit using a vector instead of a list, and demand the bucket list to be doubly linked.

Hashing. The implementation of bucket in PEPTable using Hashing is shown in Figure 5(b). We use Cantor pairing function [4] for hashing. For deleting a node pair, each bucket in PEPTable needs to be searched, which can be done in time logarithmic in the bucket size. On a merge of pointers p and q, a pair (p, q) needs to be inserted in the current iteration's hash-bucket in a sorted manner.

Table 1: Benchmark characteristics and performance comparison of EM versus other techniques

			DP	HCD	FIA		EM	
Benchmark	#Vars	#Cons	Time(s)	Time(s)	Time(s)	loss %	Time(s)	loss %
164.gzip	1595	1773	0.00	0.00	0.00	0.00	0.00	0.00
175.vpr	8922	10449	0.02	0.02	0.01	0.00	0.03	**36.09**
176.gcc	135628	194545	1.79	16.50	1.80	0.00	3.14	0.00
177.mesa	32880	37700	0.12	0.23	0.11	0.00	0.21	0.00
179.art	586	603	0.00	0.00	0.00	0.00	0.00	0.00
181.mcf	1230	1509	0.00	0.01	0.00	0.00	0.00	0.00
183.equake	1317	1279	0.00	0.00	0.00	0.00	0.00	**23.90**
186.crafty	6126	6768	0.00	0.00	0.00	0.00	0.01	0.00
188.ammp	7852	8737	0.02	0.02	0.02	0.00	0.02	0.00
197.parser	13913	18148	0.08	0.09	0.11	0.00	0.10	0.00
252.eon	65265	76521	11.80	15.20	7.70	**35.93**	3.34	0.08
253.perlbmk	60486	84487	1.42	10.30	1.34	2.48	1.76	2.46
254.gap	49531	59436	0.08	0.07	0.08	0.00	0.27	0.00
255.vortex	26221	36770	0.09	0.38	0.08	0.00	0.17	0.00
256.bzip2	1147	1081	0.00	0.00	0.00	0.00	0.00	0.00
300.twolf	17843	19806	0.03	0.03	0.03	0.00	0.06	0.00
httpd	133893	167764	77.60	331.00	79.40	**249.13**	36.10	0.11
sendmail	74622	89049	15.60	55.00	13.80	**536.20**	13.30	4.28
wine	62387	66507	7.81	48.50	6.50	**17.45**	5.58	1.49
Total	701444	882932	**116.46**	**477.34**	**110.97**	**172.22**	**64.09**	**1.00**

4. EXPERIMENTAL EVALUATION

We evaluate the effectiveness of our approach using all the sixteen C/C++ benchmarks from SPEC 2000 suite and three large open source programs, namely *httpd, sendmail* and *wine*. Most of these benchmarks are formerly used in literature [13, 24, 27]. *httpd* is the Apache HTTP server, *sendmail* is an email-delivery tool, and *wine* is a Windows emulator for Linux. Benchmark characteristics (number of points-to constraints and number of pointers in the LLVM [20] IR) are given in Table 1.

We compare our EM analysis with the following highly optimized implementations. All these implementations are flow- and context-insensitive.

- *ANDERS*: This is our set constraint-based implementation of Andersen's analysis [2]. It uses sparse bitmaps to store points-to information.

- *HCD*: This is the *Hybrid Cycle Detection* (HCD) algorithm implemented using bitmaps from Hardekopf and Lin [12] (taken from Hardekopf's website [10]). This method first uses linear time offline algorithm for preprocessing and later uses this preprocessed information for detecting cycles at the time of the analysis. We use the bitmap-based implementation as it outperforms the corresponding BDD-based version.

- *DP*: This is the Deep Propagation method from Pereira and Berlin [33] (downloaded from Pereira's website [32]), which propagates points-to information in the constraint graph to all the reachable nodes in depth-first manner. It uses sparse bitmaps to store points-to sets and has been shown to scale well.

- *FIA*: This is the Fast Interprocedural Analysis from DeFouw et al. [6]. We compare with this approach as it also uses merging for approximation. However, it merges a node with all of its successor nodes after it has been visited for a certain number of times during propagation of points-to information. The method does not check for similar points-to information of the nodes-to-be-merged. We implemented this method by using bitmap for storing points to information.

In terms of analysis time, DP performs consistently better than ANDERS, HCD and FIA on our benchmarks, and except FIA all other methods compute the same points-to information. Therefore, we focus on comparing EM against DP alone. EM's analysis time benefits on ANDERS, HCD and FIA are higher.

We could run DP without making any modifications. However, DP expects input in a special *.gen* format. Therefore, we converted points-to constraints from LLVM IR into *.gen* format using a simple LLVM pass. As is the standard practice, we time only the initialization and the solving phases for each method, and not the constraint-loading time.

For comparing precision, we use the total number of points-to pairs computed by an algorithm. Higher the number, smaller is the precision. Precision is quantified relative to the points-to information computed by Andersen's analysis. Thus, an analysis (such as EM∞ where K=∞, also denoted as A(∞, -)) has zero precision loss. In general, the precision loss percentage is computed as 100 * (|EM.ptsto| - |ANDERS.ptsto|) / |ANDERS.ptsto|.

4.1 Analysis Time

The very purpose of EM is to improve efficiency of the existing inclusion-based analyses. Due to intelligent merges, we expect propagation cost to reduce considerably, resulting in an improved analysis time. Note that an arbitrary merging of pointers may improve execution time, but would considerably hurt precision.

Table 1 shows the analysis time comparison of DP and EM with K=1. On several benchmarks where absolute analysis times are small (such as 175.vpr, 176.gcc, etc.), EM is marginally slower than DP. This happens because the saved

Table 2: Memory comparison of various techniques

			Memory (MB)				Number of points-to facts			
Benchmark	#Vars	#Cons	DP	HCD	FIA	EM	DP	HCD	FIA	EM
164.gzip	1595	1773	1	1	1	1	2808	2808	2808	2808
175.vpr	8922	10449	4	5	4	5	63960	63960	63960	87045
176.gcc	135628	194545	165	308	166	170	19274216	19274216	19274216	19274216
177.mesa	32880	37700	28	49	28	33	1625842	1625842	1625842	1625842
179.art	586	603	1	1	1	1	2451	2451	2451	2451
181.mcf	1230	1509	1	1	1	1	2414	2414	2414	2414
183.equake	1317	1279	1	1	1	1	3422	3422	3422	4240
186.crafty	6126	6768	2	2	2	2	10567	10567	10567	10567
188.ammp	7852	8737	5	8	5	6	63574	63574	63574	63574
197.parser	13913	18148	9	14	9	11	789224	789224	789224	789224
252.eon	65265	76521	503	998	317	143	49636034	49636034	67467900	49674074
253.perlbmk	60486	84487	201	377	199	201	15263141	15263141	15641184	15637944
254.gap	49531	59436	15	19	16	23	131998	131998	131998	132001
255.vortex	26221	36770	31	57	31	36	1419017	1419017	1419017	1419017
256.bzip2	1147	1081	1	1	1	1	1671	1671	1671	1671
300.twolf	17843	19806	7	9	7	10	141893	141893	141893	141893
httpd	133893	167764	1370	2610	855	868	68846405	68846405	240366813	68922293
sendmail	74622	89049	517	1040	447	458	29803958	29803958	189612505	31080308
wine	62387	66507	375	798	294	303	17662174	17662174	20743756	17925768
Average	36918	46470	**170**	**332**	**126**	**120**	**10776040**	**10776040**	**29335011**	**10884071**

cost of propagation does not get amortized by the additional cost of merging. On the other hand, when the propagation cost gets higher (especially for larger benchmarks such as 252.eon, httpd, etc.) merging the pointer-equivalent nodes starts becoming beneficial. In fact, we can observe that EM's benefit increases with the increasing analysis time of DP. This indicates that merging of nodes is useful especially for long-running benchmarks. The benefit mainly stems from reduced propagation due to node-merging (which, in turn, would also reduce the number of analysis iterations). The approximate method FIA outperforms the precise methods in terms of analysis time, but has a considerable precision loss (172.22%). This indicates that a greedy merging of nodes based only on traversal and connectivity does not lead to good approximation. FIA is successful in improving the analysis time, but at the cost of considerable precision. In contrast, EM has a minimal precision loss on an average. Overall, across all the 19 benchmarks, EM is the fastest, and consumes 45%, 86% and 42% less time than DP, HCD and FIA methods respectively.

4.2 Precision

Due to approximations added by EM, it is of utmost importance to measure the effect on the amount of computed points-to information. It is easy to see that EM computes a superset of the information computed by Andersen's inclusion-based analysis. Ideally, we want EM's precision to be as close to an inclusion-based analysis as possible.

Table 1 also shows the precision loss due to EagerMerge over the base analysis (DP) precision. It is computed as a fraction of additional points-to information computed by EM over that of DP. We observe that except for 183.equake and 175.vpr, the precision loss of EM is considerably small (less than 5%). This supports our decision for not splitting a node when the points-to information of the nodes it represents differs. In case of both 183.equake and 175.vpr, the points-to information of several pointers gets initialized to the same variables, but the pointers are indirectly assigned to different global variables during the main processing loop. Therefore, eager merging of pointers results in reduced precision. At this stage, we can expect that increasing K should regain precision especially for these two benchmarks (which is evident from the results in Section 4.4). Alternatively, using splitting of the merged nodes would also be a possibility. On the other hand, EM performs far better than FIA especially for large benchmarks in terms of precision.

4.3 Memory

Since EM and FIA merge nodes, we expect their memory requirements to be lower than that of DP. However, the two approximate methods also maintain additional data structures to enable approximations. Similarly, the two exact methods, DP and HCD, use bitmaps to store points-to information but use different algorithms. Hence, their memory requirements would also differ. In particular, DP is designed to reduce memory requirement [33], hence it is expected to use less memory than HCD.

Table 2 shows the memory requirements of EM versus other methods. We observe that both the approximate methods considerably reduce the memory consumption due to node-merging, and the memory requirement is proportional to the corresponding analysis time (Figure 1). For relatively smaller benchmarks, the cost of maintaining the additional data structures in EM dominates, resulting in increased memory requirement. However, for large benchmarks such as *httpd* and *sendmail*, effectiveness of merging overcomes the additional data structure cost. Overall, across all the 19 benchmarks in our test harness, EM consumes 28%, 63% and 5% less memory than DP, HCD and FIA respectively.

Table 2 also shows the number of points-to facts computed by each method. As expected, DP and HCD compute the same points-to information. FIA and EM compute more points-to facts than the exact methods, but FIA, on an av-

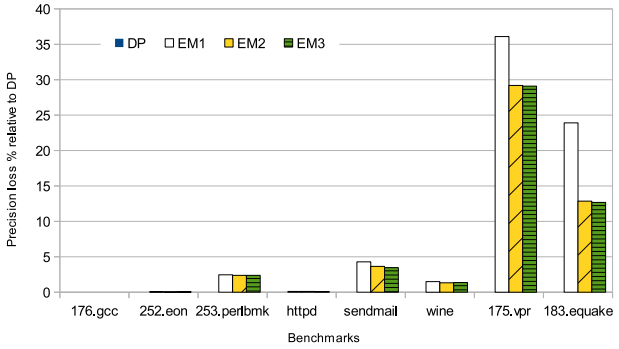

Figure 6: Effect of K on analysis precision. EM1 means EagerMerge with K=1 and so on.

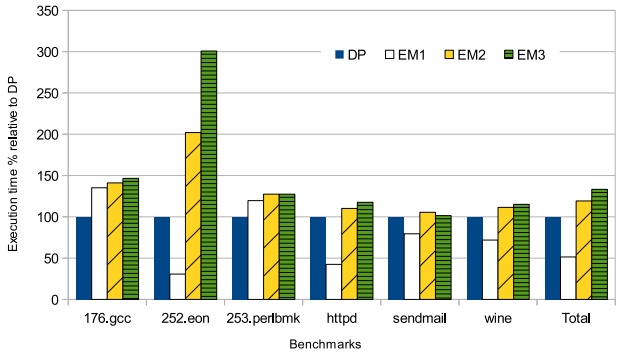

Figure 7: Effect of K on the analysis time. EM1 means EagerMerge with K=1 and so on.

erage, computes much larger points-to information than EM. On the other hand, EM computes only a small overapproximation of the information computed by exact methods.

4.4 Effect of Parameter K

Figure 6 shows the effect of parameter K on the analysis precision for large benchmarks (0 indicates no precision loss). We vary K from 1 to 3, that is, we do not merge nodes until their points-to information remains the same for K iterations. Overall, we observe a clear improvement in precision with increasing K. However, the result is more pronounced for certain benchmarks such as sendmail, 175.vpr and 183.equake, which reach fixed-point within four iterations. In contrast, benchmarks such as 253.perlbmk and 176.gcc reach fixed-point in 10 and 13 iterations respectively, and therefore, increasing K from 1 to 3 only marginally improves their precision. We expect the trade-off of increasing K to be visible in the analysis time, which we discuss next.

Figure 7 shows the effect of parameter K on the analysis time for large benchmarks (EM is better suited for large benchmarks). In the plot, we present EM execution time relative to DP execution time. Thus, in Figure 7, the first bar for each benchmark is set to 100%, which corresponds to DP. We observe that increasing K usually increases the analysis time. Increased value of K indicates that the analysis behaves in a more cautious manner and waits until it sees two pointers being equivalent for a larger number of iterations. Such a cautious approach increases confidence while merging, but it also reduces the number of merges.

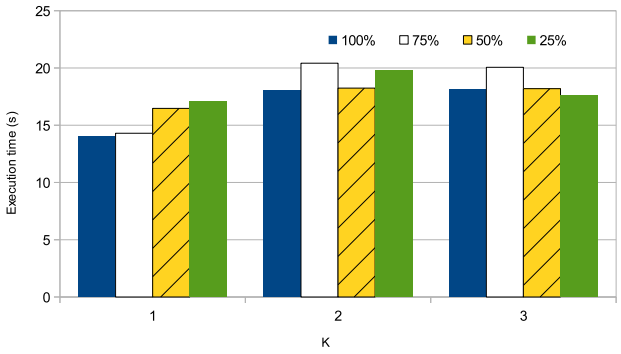

Figure 8: Effect of partial merging on the analysis time for *sendmail*. Each bar represents the execution time when the corresponding percentage of pointer-pairs were merged.

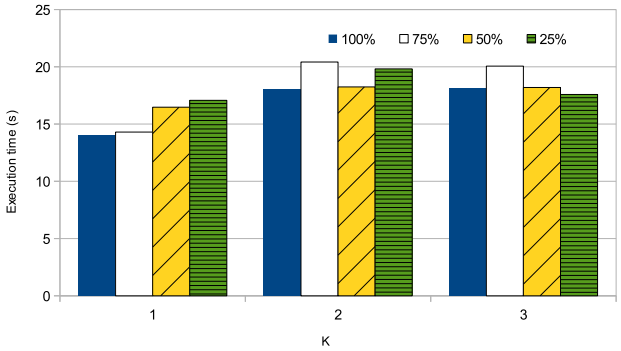

Figure 9: Effect of partial merging on the analysis time for *wine*.

Increased execution time on increasing K indicates that losing the merging opportunities makes the analysis propagate more points-to information in the constraint graph.

4.5 Effect of Partial Merge

In the experiments so far, we merged all the eligible nodes. That is, if two pointers are equivalent for K iterations, we merge all of them. It is possible to alter the merging strategy to allow only a subset of the eligible nodes to get merged. This may be helpful to further reduce the precision-loss.

We experimented with merging only a fraction of the eligible candidate pairs. Figure 8 shows the effect of partial merging on the analysis time for sendmail. We observe that choosing a random subset need not lead to a deterministic benefit or performance degradation. Specifically, for K=1, we see that choosing smaller and smaller subset of pairs increases analysis time, whereas for K=2, it alternates between reduced and increased performances, while for K=3, it increases first and then decreases. Due to this varied result, we cannot conclusively deduce the effect of partial merging on the analysis time. Figure 9 shows the effect of partial merging on the analysis time for wine. We observe mostly a similar trend as that for sendmail.

Overall, we observe that optimistic EM provides an effective way to improve efficiency of points-to analysis.

5. RELATED WORK

Survey on pointer analysis techniques is presented by Hind and Pioli [15], and recently by Smaragdakis and Balatsouras [37].

Pointer analysis algorithms are largely classified as unification-based and inclusion-based. In the former, points-to sets of the pointers involved in an assignment are merged, while in the latter, a subset relationship is maintained across the two points-to sets. Due to spuriously added aliasing relations, unification turns out to be less precise than inclusion. However, due to merging, pointers can be analyzed faster with union-find data structures.

Steensgaard [40] proposed an almost linear time algorithm for analyzing C programs using unification. In contrast, inclusion-based algorithms, as proposed by Andersen [2], incur a cubic computational complexity with difference propagation [31]. Our proposal of EagerMerge offers benefits in between those of unification and inclusion, as briefly discussed in Section 3.2. In that sense, it shares the goal of Shapiro and Horwitz [36]. However, the mechanisms to achieve the goal are considerably different. Shapiro and Horwitz limit the number of points-to edges of each pointer node in the points-to hierarchy to achieve the desired precision, whereas EagerMerge optimistically chooses currently tracked pointer-equivalent nodes for merging.

A naive implementation of Andersen's analysis turns out to be inefficient in practice. Therefore, several novel techniques have been developed to improve upon the original Andersen's analysis [3, 14, 22, 42, 44, 23, 41]. Binary Decision Diagrams (BDD) [3, 42] are used to store points-to information in a succinct manner. Although the space reduction using BDD is significant, it also incurs a performance penalty over sparse bitmaps, since accessing and merging points-to information especially for load and store constraints involve a complex logic over the boolean hierarchy.

The idea of *bootstrapping* [17] uses a divide-and-conquer strategy to first divide the large problem of pointer analysis by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [40]) and later, a more precise algorithm analyzes each partition. Due to the small partition sizes, the overall analysis scales well with the program size. The analysis over different alias partitions can also be performed in parallel. Nasre et al. [29] convert points-to constraints into a set of linear equations and solve it using a standard linear solver.

Storing complete calling context information achieves a good precision, but at the cost of storage and analysis time. For a complete context-sensitive analysis, potentially, the storage requirement and the analysis time can be exponential in the number of functions in the program making it non-scalable. Therefore, approximate representations have been introduced to trade off precision for scalability. Das [5] proposed *one level flow*, while Lattner et al. [21] unified contexts, while Nasre et al. [30] hashed contexts to alleviate the need to store the complete context information in a context-sensitive analysis. In the context of Java programs, several variants of context-sensitive analyses have also been devised to tame the complexity [19, 39, 7].

Inclusion based analysis can be improved using several novel enhancements proposed in literature. Online cycle elimination [8] breaks dependence cycles amongst pointer variables on the fly. Offline variable substitution [34] and set-based pre-processing [38] operate over constraints prior to the constraint evaluation to find pointer equivalent variables and rewrites constraints with the reduced set of variables to improve the analysis time. Hardekopf and Lin [11] provide a suite of offline analyses based on Hash-based Value Numbering to further improve the effectiveness of offline methods. Guyer and Lin [9] propose client-driven pointer analysis that adjusts analysis precision based on the client needs. Our work is complementary to these approaches and can be combined with these for improved benefits.

Wave and Deep Propagation techniques [33] perform a breadth-wise and depth-wise propagation of points-to information in a constraint graph. Various heuristics like Greatest Input Rise, Greatest Output Rise, and Least Recently Fired [18] work on the amount and recency of information computed at various nodes in the constraint graph to achieve a quicker fixed-point. Prioritized constraint evaluation [28] dynamically orders constraints to produce useful edges early in the constraint graph. Dominator-based analysis [26] exploits structural properties of dominator trees to find pointer-equivalent variables. All these methods preserve the precision of the underlying inclusion-based analysis. In contrast, EagerMerge optimistically merges nodes with the hope that the corresponding pointers would be pointer-equivalent even at the end of the analysis. We illustrated that this optimism works well in practice.

The closest to our work is Fast Inter-procedural class Analysis (FIA) [6]. FIA is a general parameterized analysis framework that integrates propagation-based and unification based analysis primitives. After a node has been visited K times, it is merged with *each of its successor nodes*. Although, similar to EM, this method also adds approximations, there are notable differences. First, the FIA method tracks visit information *within* iteration whereas EM works *across* iterations during the analysis, finding better opportunities. Second, FIA does not check for points-to information, i.e., it merges a node with all of its successors without considering if they have similar or different points-to information. In contrast, EM tracks nodes having the same points-to information. This difference in strategies significantly affects the analysis precision as shown in Table 1 (172.22% versus 1% precision loss for FIA and EM respectively).

Another closely related work [25] proposes merging of approximately equivalent pointers and approximately location-equivalent pointees during inclusion-based analysis. The similarity in points-to sets is decided using Jaccard's coefficient, with a tunable threshold. EagerMerge differs with this work as we work with fully-equivalent pointers.

Overall, we believe EM provides significant efficiency benefits losing minimal precision.

6. CONCLUSION

We formulated K-Merge, a technique based on merging of points-to sets, and instantiated it as EagerMerge to merge pointer-equivalent nodes in each iteration. We illustrate that EagerMerge provides almost 50% benefit over an optimized baseline with minimal precision loss, making it practical. In future, we would like to study if the benefits of EM carry forward in context- and flow-sensitive, as well as demand-driven and client-driven versions of pointer analysis.

7. ACKNOWLEDGMENTS

This work is partially supported by IIT Madras Grant CSE/13-14/636/NFSC/RUPS.

8. REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. ACM.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to Analysis Using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[4] Georg Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover Publications, 1915.

[5] Manuvir 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.

[6] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 222–236, New York, NY, USA, 1998. ACM.

[7] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale Exhaustive Points-to Analysis for Java in Under a Minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 535–551, New York, NY, USA, 2015. ACM.

[8] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

[9] Samuel Z. Guyer and Calvin Lin. Client-driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] Ben Hardekopf. BDD-based Lazy Cycle Detection, http://www.cs.ucsb.edu/ benh/research/downloads.html.

[11] Ben Hardekopf and Calvin Lin. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 265–280, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] Ben Hardekopf and Calvin 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, New York, NY, USA, 2007. ACM.

[13] Ben Hardekopf and Calvin 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, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[14] Nevin Heintze and Olivier Tardieu. Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[15] Michael Hind and Anthony 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, New York, NY, USA, 2000. ACM.

[16] David Hovemeyer and William Pugh. Finding More Null Pointer Bugs, but Not Too Many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM.

[17] Vineet Kahlon. Bootstrapping: A Technique for Scalable Flow and Context-sensitive Pointer Alias Analysis. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 249–259, New York, NY, USA, 2008. ACM.

[18] Atsushi Kanamori and Daniel Weise. Worklist Management Strategies for Dataflow Analysis, MSR Technical Report, MSR-TR-94-12, 1994.

[19] George Kastrinis and Yannis Smaragdakis. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, New York, NY, USA, 2013. ACM.

[20] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[21] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[22] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[23] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An Incremental Points-to Analysis with CFL-Reachability. In *Proceedings of the 22nd International Conference on Compiler Construction*, CC'13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[24] Mario Mendez-Lojo, Martin Burtscher, and Keshav

Pingali. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[25] Rupesh 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.

[26] Rupesh Nasre. Exploiting the Structure of the Constraint Graph for Efficient Points-to Analysis. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 121–132, New York, NY, USA, 2012. ACM.

[27] Rupesh Nasre. Time- and space-efficient flow-sensitive points-to analysis. *ACM Trans. Archit. Code Optim.*, 10(4):39:1–39:27, December 2013.

[28] Rupesh Nasre and R. Govindarajan. Prioritizing Constraint Evaluation for Efficient Points-to Analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 267–276, Washington, DC, USA, 2011. IEEE Computer Society.

[29] Rupesh Nasre and Ramaswamy Govindarajan. Points-to Analysis As a System of Linear Equations. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 422–438, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] Rupesh Nasre, Kaushik Rajan, R. Govindarajan, and Uday 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, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Control*, 12(4):311–337, December 2004.

[32] Fernando Pereira. Wave Propagation / Deep Propagation website, http://compilers.cs.ucla.edu/fernando/projects/pta/home/.

[33] Fernando Magno Quintao Pereira and Daniel Berlin. Wave Propagation and Deep Propagation for Pointer Analysis. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 126–135, Washington, DC, USA, 2009. IEEE Computer Society.

[34] Atanas Rountev and Satish 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, New York, NY, USA, 2000. ACM.

[35] Cindy Rubio-González and Ben Liblit. Defective Error/Pointer Interactions in the Linux Kernel. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 111–121, New York, NY, USA, 2011. ACM.

[36] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-insensitive Points-to Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 1–14, New York, NY, USA, 1997. ACM.

[37] Yannis Smaragdakis and George Balatsouras. Pointer Analysis. *Foundations and TrendsÃĆÂő in Programming Languages*, 2(1):1–69, 2015.

[38] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. Set-based Pre-processing for Points-to Analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 253–270, New York, NY, USA, 2013. ACM.

[39] Smaragdakis, Yannis and Kastrinis, George and Balatsouras, George. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM.

[40] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[41] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Software: Practice and Experience*, 44(12):1485–1510, 2014.

[42] John Whaley and Monica S. Lam. An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 180–195, London, UK, UK, 2002. Springer-Verlag.

[43] John Whaley and Monica S. Lam. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[44] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing 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, New York, NY, USA, 2010. ACM.