Prioritizing Constraint Evaluation for Efficient Points-to Analysis

Rupesh Nasre
Computer Science and Automation,
Indian Institute of Science,
Bangalore, India – 560012
Email: nasre@csa.iisc.ernet.in

R. Govindarajan
Computer Science and Automation,
Indian Institute of Science,
Bangalore, India – 560012
Email: govind@csa.iisc.ernet.in

***Abstract*—Pervasive use of pointers in large-scale real-world applications continues to make points-to analysis an important optimization-enabler. Rapid growth of software systems demands a scalable pointer analysis algorithm. A typical inclusion-based points-to analysis iteratively evaluates constraints and computes a points-to solution until a fixpoint. In each iteration, (i) points-to information is propagated across directed edges in a constraint graph `G` and (ii) more edges are added by processing the points-to constraints. We observe that prioritizing the order in which the information is processed within each of the above two steps can lead to efficient execution of the points-to analysis. While earlier work in the literature focuses only on the propagation order, we argue that the other dimension, that is, prioritizing the constraint processing, can lead to even higher improvements on how fast the fixpoint of the points-to algorithm is reached. This becomes especially important as we prove that finding an optimal sequence for processing the points-to constraints is NP-Complete. The prioritization scheme proposed in this paper is general enough to be applied to any of the existing points-to analyses. Using the prioritization framework developed in this paper, we implement prioritized versions of Andersen's analysis, Deep Propagation, Hardekopf and Lin's Lazy Cycle Detection and Bloom Filter based points-to analysis. In each case, we report significant improvements in the analysis times (33%, 47%, 44%, 20% respectively) as well as the memory requirements for a large suite of programs, including SPEC 2000 benchmarks and five large open source programs.**

I. Introduction

Static analysis of programs is important to achieve faster runtime execution, more secure code, less buggy programs, invariant guarantees and a better program understanding. As more and more emphasis is laid on guaranteeing secure code, data-race free programs and memory-leak free programs, the demand on scalable and precise static analysis increases. Further, the rapid growth of code bases places even further demand on scalable static analysis techniques. Points-to analysis is an important static analysis which is an enabler for several compiler optimizations. The effectiveness of several compiler optimizations depends heavily on the underlying pointer analysis. Thus, the scalability requirement of pointer analysis and its importance is unquestionable. Indeed, the literature on points-to analysis is rich with several interesting ideas [1], [2], [3], [4], [5], [6], [7].

For analyzing a general purpose C program, it is sufficient to consider all pointer statements of the following forms: address-of assignment ($p = \&q$), copy assignment ($p = q$), load assignment ($p = *q$) and store assignment ($*p = q$) [7]. Load and store assignments are also referred to as *complex* assignments in literature (e.g., [7]). We use the terms *constraint*, *statement* and *assignment* interchangeably in this article. We deal with flow-insensitive, context-sensitive inclusion-based points-to analyses in this work.

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 fixpoint
```

A flow-insensitive analysis iterates over a set of points-to constraints until a fixpoint 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 `n1` to node `n2` represents propagation of points-to information from `n1` to `n2`. 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* 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. 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 and no more points-to information can be computed, `G` gets stabilized and a fixpoint (at the nodes) is reached. An outline of this analysis is given in Algorithm 1.

Techniques have been developed for efficient *propagation* of the points-to information across the edges of a constraint graph, i.e, Line 4 of Algorithm 1. Online cycle elimination [8] detects cycles in `G` on-the-fly and collapses all the nodes in a cycle into a representative node. This speeds up the propagation of points-to information since all the nodes in a cycle eventually contain the same points-to information. Wave and Deep Propagation [7] perform a topological ordering of the edges and propagate difference in the points-to information in a breadth-first or depth-first manner. Various heuristics like Greatest Input Rise, Greatest Output Rise, and Least Recently Fired [9] work on the amount and recency of information computed at various nodes in the constraint graph to achieve a quicker fixpoint.

All of the above techniques essentially focus on the propagation order (Line 4 of Algorithm 1) and prioritize the order in which the points-to propagation takes place. However, these techniques do not attempt to dictate which evaluation order of the constraints (Line 5 of Algorithm 1) would prove more beneficial for faster points-to information computation. Specifically, there are two aspects of the constraint evaluation that are hitherto not exploited in literature: (i) how many edges a constraint adds, and (ii) where in G a constraint adds edges. We observe that both these parameters are important and can significantly influence the fast convergence of the fixpoint computation. It should be noted that neither the propagation order nor the constraint evaluation order changes the fixpoint of the points-to solution.

We develop a framework that deals with the priority ordering of the points-to constraints and the propagation of points-to information. The two criteria mentioned above give rise to a priority assigned to each constraint. The priority is dynamic in nature and can change as the analysis progresses. Our prioritized analysis not only evaluates constraints in the priority order, but also evaluates certain constraints repeatedly based on priority. The result is a skewed evaluation of important and useful constraints early and in a repeated manner to reach the (same) fixpoint solution faster.

To summarize, while earlier approaches [8], [9], [7] focus on the propagation order of the points-to information, we address *the evaluation order of the points-to constraints*.

Major contributions of this paper are as below.

- We prove that finding a sequence of the points-to constraints to reach the fixpoint in an optimal number of steps in a flow-insensitive inclusion-based analysis is NP-Complete (Section II).
- We develop a priority based greedy analysis framework for efficient computation of points-to information (Section III). The framework is general and can be used for other static analyses.
- We instantiate our framework by defining *constraint priority* based on the structure of and the number of points-to facts changed by a constraint. We extend it to a dynamic constraint ordering that can be easily applied to a particular points-to analysis algorithm (Section IV).
- We show the effectiveness of our approach by applying it on top of the state-of-the-art algorithms (Andersen's analysis [1], BDD-based Lazy Cycle Detection [10], Deep Propagation [7] and Bloom Filters [11]) for SPEC 2000 benchmarks and five large open source programs (*httpd*, *sendmail*, *gdb*, *wine-server* and *ghostscript*) (Section V). Our experimental evaluation shows a significant improvement in the analysis times: 33% in Andersen's analysis, 47% in Lazy Cycle Detection, 44% in Deep Propagation and 20% in Bloom Filter based analysis. A positive side-effect of the application of prioritized constraint evaluation is the reduction in the memory requirement of the original analyses: 17% in Andersen's analysis and 23% in Deep Propagation.

II. Optimal Ordering of Constraints

Given our observation that prioritizing the processing of points-to constraints in Line 5 of Algorithm 1 improves the analysis time, it raises the question: *does there exist an order in which the constraints should be processed which can ensure optimal number of steps for reaching the fixpoint?* For instance, given a set of points-to constraints

$\mathtt{a} = \&\mathtt{x}, \mathtt{b} = \&\mathtt{y}, \mathtt{p} = \&\mathtt{q}, *\mathtt{q} = \mathtt{b}, *\mathtt{p} = \mathtt{a},$

it takes two iterations to reach the fixpoint, if they are processed in the above order. However, processing the constraints in the following order ensures fixpoint in one iteration.

$\mathtt{a} = \&\mathtt{x}, \mathtt{b} = \&\mathtt{y}, \mathtt{p} = \&\mathtt{q}, *\mathtt{p} = \mathtt{a}, *\mathtt{q} = \mathtt{b}.$

Since existing techniques decouple propagation and evaluation of the *complex* constraints, they do not reorder the constraints, which results in requiring multiple iterations to reach the fixpoint.

In this section, we prove that computing such a sequence even for a restricted scenario where only copy constraints are allowed is NP-Complete.

Theorem 1. *Computing the flow-insensitive inclusion-based points-to solution in an optimal number of steps from a set of copy constraints is NP-Complete.*

Proof: We reduce Set Cover problem to points-to analysis. Consider a Set Cover instance SC($\mathcal{U}, \mathcal{S}$, K) with universe $\mathcal{U}$ and a set $\mathcal{S}$ of subsets $\mathcal{S}_i$. The decision version of the Set Cover problem states that given a universe $\mathcal{U}$ and a set $\mathcal{S}$ with subsets $\mathcal{S}_i$ possibly having elements in common, whether there exists a set of K subsets whose union contains all the elements contained in any $\mathcal{S}_i$. The problem SC($\mathcal{U}, \mathcal{S}$, K) is known to be NP-Complete [12].

We reduce SC($\mathcal{U}, \mathcal{S}$, K) to PTA(C, S, K) which is a flow-insensitive points-to analysis over a set of points-to (copy) constraints C with an initial points-to information and the fixpoint defined with respect to pointer S. The decision version of PTA checks if the constraints in C can be evaluated in a manner such that the fixpoint with respect to S is reached in K steps. A step indicates evaluation of a constraint.

The reduction is performed as below. For each element $s \in \mathcal{S}_i$, we create an initial points-to information $\mathcal{S}_i \rightarrow \{s\}$, i.e., $\mathcal{S}_i$ points-to s. For each set $\mathcal{S}_i$, we create a copy statement $\mathcal{S} = \mathcal{S}_i$. This transformation from SC($\mathcal{U}, \mathcal{S}, K$) to PTA($C$, S, K) is linear in the number of sets and the number of elements. Thus, SC polynomially reduces to PTA.

If an efficient (polynomial) solution exists for PTA, then the solution can be mapped back to SC. Thus, if there is a sequence of K steps to obtain the fixpoint, and since the fixpoint would contain all the points-to information, then the subsets $\mathcal{S}_i$ corresponding to the chosen copy constraints ($\mathcal{S} = \mathcal{S}_i$) would cover all the elements $s \in \mathcal{U}$ that correspond to the points-to facts in the fixpoint, forming the set cover.

Similarly, if a set cover of size K exists, then no other subset would be able to add any new points-to information to $\mathcal{S}$ and the K subsets would form an optimal sequence of K steps to obtain the fixpoint over the constraints.

Thus, PTA is NP-Hard. (a)

It is easy to see that a given sequence of K constraints can act as a polynomial time verifier to check if PTA evaluates to a fixpoint. Thus, PTA is in NP. (b)

From (a) and (b), PTA is NP-Complete. ■

III. Prioritized Computation of Constraints

We first explain our priority-based approach using an example, then discuss various priority schemes followed by our prioritization framework. Our priority based approach may be viewed similar to the maximum benefit approach in the online set cover problem [13].

A. Motivating Example

Consider the program fragment given in Figure 1(a). Let the initial points-to information due to the address-of constraints be $\mathtt{a} \rightarrow \{\mathtt{a},\mathtt{q},\mathtt{r},\mathtt{s},\mathtt{t}\}$ and $\mathtt{p} \rightarrow \{\mathtt{b},\mathtt{c},\mathtt{d}\}$. Figure 1(b) illustrates the constraint graph G at the end of different iterations of Andersen's analysis [1] with points-to information propagated using Deep Propagation [7]. For simpler exposition, we assume that online cycle elimination [8] is not performed.

A node is represented as $\mathtt{n}\{\mathtt{P}\}$ where n is a pointer and $\{\mathtt{P}\}$ is its points-to set computed so far. Directed edge from node $\mathtt{n}_1$ to $\mathtt{n}_2$ represents the propagation of points-to information from $\mathtt{n}_1$ to $\mathtt{n}_2$. As the behavior of the pointers $\mathtt{q},\mathtt{r},\mathtt{s},\mathtt{t}$ is the same in this example, we use a single node to represent all of them. Prior to Iteration 1, edges d to e and a to b are added to G using copy constraints (refer Algorithm 1). Iteration 1 starts with propagating the points-to set {a,q,r,s,t} from node a to b. Since the points-to set of d is empty, no information flows to e. The next step of processing *complex* constraints adds the edges as indicated below.

`*e = c`: none.
`c = *a`: a to c, qrst to c.
`*a = p`: p to a, p to qrst.

The new edges introduced in this iteration are shown as thick lines. Thus, at the end of the first iteration, the points-to information computed at different nodes as well as the constraint graph are shown in Figure 1(b), Iteration 1.

The analysis continues propagating more points-to information and then adding more edges (shown as thick lines) in each iteration until it reaches the fixpoint in Iteration 5. The final points-to information computed at the nodes by Andersen's analysis is shown in Figure 1(b) Iteration 5. In all the iterations of the analysis, a fixed ordering of the constraints is used, which is typically the order in which the constraints appear in the program (Figure 1(a)).

Next, we explain how introducing a prioritized version of Andersen's analysis would evaluate the same set of constraints. The priority scheme can use various mechanisms for ordering the constraints. We use a mechanism wherein the priority of a constraint is the number of new points-to facts it adds in the previous iteration. Thus, the constraint priority is dynamic and may change across iterations. At the start of the analysis, i.e., before Iteration 1, the constraints can be ordered using any ordering, including the program order. In this example, we choose to use a dependence order. That is, a constraint $\mathtt{c}_1$ gets more priority over another constraint $\mathtt{c}_2$ if $\mathtt{c}_1$ may *define* a variable that $\mathtt{c}_2$ *uses*. Thus, $\mathtt{p} = \mathtt{q}$ gets higher priority over $\mathtt{r} = \mathtt{p}, \mathtt{r} = *\mathtt{p}, *\mathtt{p} = \mathtt{r}$. The constraint ordering at the start of Iteration 1 is shown in the top drawing of Figure 1(c). The value in parentheses following a constraint is the number of new points-to pairs the constraint adds in that iteration. For instance, the constraint `*a = p` adds 18 new points-to pairs to G in Iteration 1.

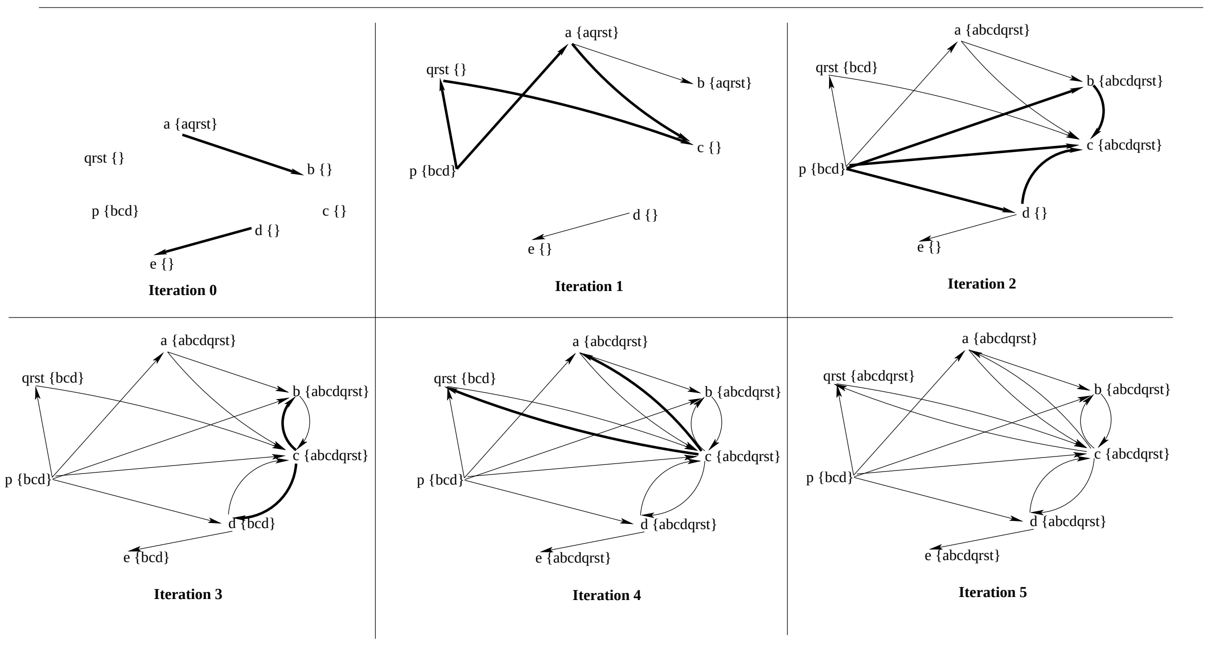
As in the case of unprioritized Andersen's analysis above, prior to Iteration 1, the copy constraints `e = d` and `b = a` add edges d to e and a to b respectively. Iteration 1 of our prioritized approach adds directed edges and computes points-to information as shown in the top drawing of Figure 1(c). Specifically, the constraint `*a = p` adds edges from p to qrst and p to a. This allows for addition of the points-to set b,c,d to those of qrst, a and b, i.e., 18 new points-to pairs. Similarly, the constraint `c = *a` adds 4 new edges: from nodes qrst, a, b, d to c and 8 new points-to pairs: $\mathtt{c} \rightarrow \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{q},\mathtt{r},\mathtt{s},\mathtt{t}\}$. The last constraint `*e = c` does not add any edges or new points-to information. Contrasting the state of G in Iteration 1 of the prioritized Andersen's analysis with that of Andersen's analysis, we observe that two additional edges are added in the prioritized analysis, namely b to c and d to c. Thus, compared to Iteration 1 of Andersen's analysis, Iteration 1 of the prioritized version adds the following additional points-to information to the solution: $\mathtt{a},\mathtt{b},\mathtt{q},\mathtt{r},\mathtt{s},\mathtt{t} \rightarrow \{\mathtt{b},\mathtt{c},\mathtt{d}\}, \mathtt{c} \rightarrow \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{q},\mathtt{r},\mathtt{s},\mathtt{t}\}$. In general, our priority based analysis enables addition of more edges to the constraint graph early resulting in more possibilities for early propagation of points-to information. Further note that if the constraint `*a = p` is evaluated twice, then the edges from p to b,c,d would be added, making provision for propagation of more points-to pairs. We exploit this fact for skewed evaluation in our algorithm (Section IV).

Our priority based analysis framework keeps track of the number of points-to facts that are newly added by each constraint evaluation and accordingly assigns priority to the constraint. Multiple constraints may receive the same priority forming clusters of constraints. A customary way of representing various priorities is using levels. Thus, constraints which add i new points-to facts are assigned a priority level $\mathtt{P}_\mathtt{i}$. As the analysis progresses, constraints are mapped to different priority levels. As an example, since $\mathtt{c} = *\mathtt{a}$ adds 8 new points-to pairs in Iteration 1, it is moved to $\mathtt{P}_8$ (to be used in Iteration 2). Similarly, the constraint $*\mathtt{a} = \mathtt{p}$ is moved to $\mathtt{P}_{18}$. The constraint $*\mathtt{e} = \mathtt{c}$ remains at $\mathtt{P}_0$.

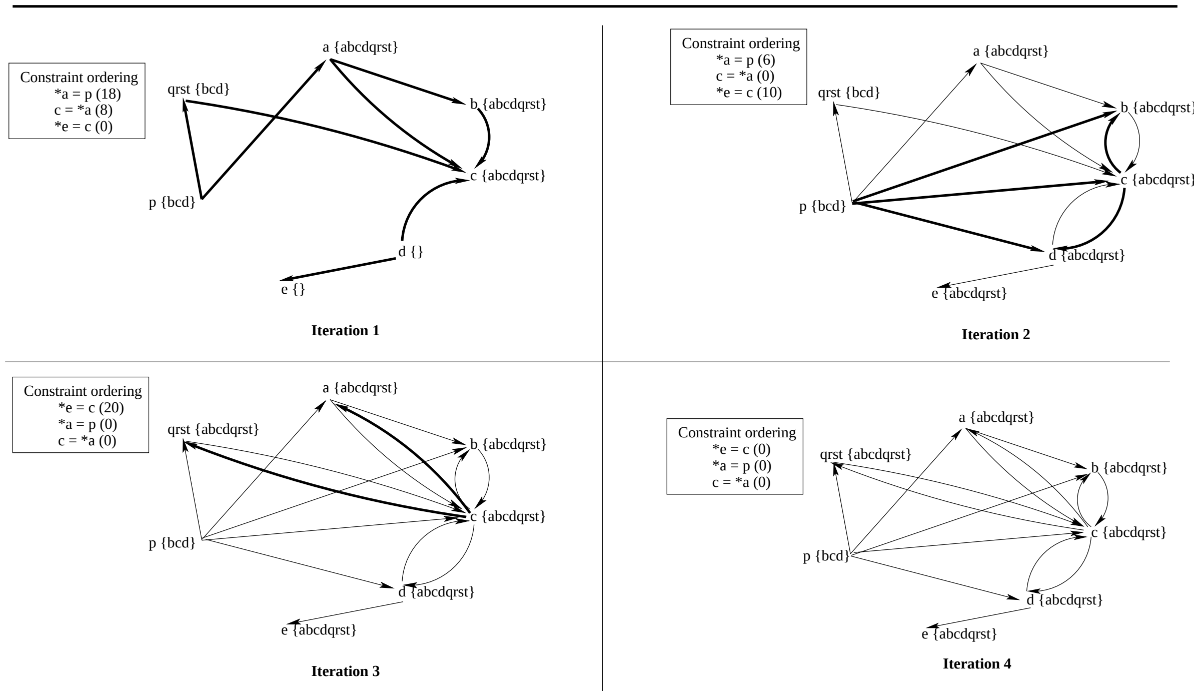
Input constraints *e = c, c = *a, e = d, b = a, *a = p

Fixed processing order e = d, b = a; *e = c, c = *a, *a = p

(a)



(b) **Andersen's Analysis**



(c) **Prioritized Andersen's Analysis**

Fig. 1. (a) Input constraints and fixed constraint ordering for Andersen's analysis. (b) Constraint graphs for Andersen's analysis. (c) Constraint graphs for Prioritized Andersen's analysis.

The prioritized ordering of the constraints at the start of Iteration 2 (shown in Figure 1(c) Iteration 2) remains the same as in Iteration 1. The new edges added (thick lines) and the points-to information computed at various nodes are as shown in Figure 1(c). In this iteration, the constraint $*\mathtt{a} = \mathtt{p}$ adds 6 new points-to pairs ($\mathtt{d}, \mathtt{e} \rightarrow \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$), the constraint $\mathtt{c} = *\mathtt{a}$ adds no new points-to information and the constraint $*\mathtt{e} = \mathtt{c}$ adds 10 new points-to pairs ($\mathtt{d}, \mathtt{e} \rightarrow \{\mathtt{a}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}\}$). Note that the edges c to b and c to d get added in this iteration, whereas in the case of the original Andersen's analysis, the same edges are added in Iteration 3 (Figure 1(b)). The points-to information at node e depends upon the information propagated via this edge (pointee set $\{\mathtt{a}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}\}$). The third drawing of Figure 1(c) shows the prioritized ordering of the constraints at the start of Iteration 3, sorted by the number of points-to pairs each constraint added in Iteration 2. The points-to information computed is as shown in Figure 1(c). In this iteration, only $*\mathtt{e} = \mathtt{c}$ adds 20 new points-to pairs ($\mathtt{qrst} \rightarrow \{\mathtt{a}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}\}$).

The method converges after Iteration 4.

As shown in the example, a priority based analysis evaluates constraints in such an order that it enables addition of more edges and more useful edges early in the constraint graph to ensure quick fixpoint computation. The propagation of the points-to information via these edges is done by the underlying analysis (Andersen's method [1] or Deep Propagation [7], etc.) which is not dictated by our method. The above example also suggests that the fixpoint computation of an analysis can be improved by going beyond the conventional mechanism of treating all the constraints with the same priority. Although the example is illustrated in the context of points-to analysis, it is equally applicable to other static analyses such as Mod/Ref analysis. The above example also illustrated two priority schemes – one based on the dependence across constraints based on *def-use* chain and another based on the amount of information a constraint changes. We carefully categorize different priority schemes in the next subsection and formalize the prioritization framework in Section III-C.

B. Priority Schemes

There are several ways in which the points-to constraints can be prioritized. We classify them into two types.

- *Linguistic scheme:* In this scheme, the constraints are prioritized based on their structure and the constraint variables. For instance, the priority mechanism in the last subsection for ordering constraints prior to Iteration 1 is a linguistic scheme. Another linguistic scheme may prioritize all load and store constraints over copy constraints.
- *Effect-driven scheme:* In this scheme, the constraints are prioritized based on the evaluation effects, e.g., the number of times a constraint gets evaluated or the number of points-to facts it adds (as in the example above).

It is possible to come up with a hybrid scheme that uses a combination of the above two schemes. For instance, one could assign different priority levels based on an effect-driven scheme, and a linguistic scheme can be used to order the constraints within the same priority level.

C. Prioritization Framework

We now formalize our notion of a priority based framework. A prioritized framework $\mathcal{R}$ is a 4-tuple.

$\mathcal{R} = \langle C, \mathcal{P}, \mathcal{F}, \leq \rangle$, where

- C is the set of input constraints,
- $\mathcal{P}$ is the set of priority levels of size $N_{\mathcal{P}}$,
- $\mathcal{F} = \{f_1, f_2, ..., f_k\}$ is a family of priority functions. In i^{th} iteration, the analysis uses one of f_j for some $j \in 1..k$. Then, a constraint $c \in C$ is assigned a priority p if $f_j(c) = p$, and
- $\leq$ is a partial order defined on the priority levels assigned to a pair of constraints c_x and c_y at the i^{th} iteration. If $f_i(c_x) = p_x$ and $f_i(c_y) = p_y$, then $p_x \leq p_y$ *iff* p_y is at a higher priority level than p_x.

IV. Prioritization Algorithm

We instantiate our prioritization framework with a set of two functions. The first function f_1 is used for the first iteration whereas the other function f_2 is used for the subsequent iterations of the analysis. Thus, $\mathcal{F} = \{f_1, f_2\}$. The function f_1 assigns priority to a constraint according to its depth in the dependence graph of constraints. Thus, it uses a linguistic scheme to define a constraint priority. For instance, if c_1 defines a variable that c_2 uses, then c_1 gets higher priority than c_2. Note that one could use any suitable priority function of choice. The function f_2 assigns a priority level to a constraint c in iteration $i + 1$ according to the amount of points-to information newly added by c in iteration i. The complete analysis developed using the prioritization framework is given in Algorithm 2. The function `evaluate()` in Line 10 implements a single iteration of the points-to information computation and propagation using any method like Andersen's analysis [1], Deep Propagation [7] or Lazy Cycle Detection [14].

Similar to Algorithm 1, our algorithm first processes the address-of and copy constraints (Lines 1–2). Line 3 finds the dependence across constraints and and Line 4 partitions them in different priority levels depending upon the topological ordering of the nodes.

The `repeat-until` loop at Lines 5–17 iterates through various constraints at various priority levels until none of the constraint evaluations changes the points-to information, suggesting that the fixpoint is reached. Each iteration of this loop corresponds to the different iterations of the points-to analysis illustrated in Figure 1 of Section III.

Constraints in each priority level are processed, starting from the highest priority level, in the `for` loop (Lines 6–16). In Line 10, the points-to information is computed using an underlying points-to analysis method. The method returns a single integer suggesting the amount of new points-to information computed. Based on the returned integer value, a new priority level is assigned to the constraint (Line 11). If the new priority level is the same as the current priority,

Algorithm 2 Prioritized Points-to Analysis.

```
Require: set C of points-to constraints
    process address-of constraints and remove from C
    add edges to G using copy constraints and remove from C
3:  sort C using dependence order
    partition the constraints in different priority levels
    repeat
6:    for all level ∈ Highest priority level .. Lowest priority level do
        times = 0
        repeat
9:        for all c ∈ $P_{level}$ do
            diff = evaluate(c)
            new-level = priority-level(diff)
12:         $P_{level} = P_{level} \setminus c$
            $P_{new-level} = P_{new-level} \cup \{c\}$
          end for
15:     until inner fixpoint or ++times > threshold
      end for
    until outer fixpoint
```

the same constraint may get processed again shortly, as it has changed the points-to information. The `repeat-until` loop (Lines 8–15) shows that the constraints in the same priority level get processed repeatedly until an inner fixpoint (fixpoint for the constraints within the same priority level). This essentially allows the skewed processing of some constraints, as they get evaluated more often than others.

Lines 12 and 13 remove a constraint from its current priority level and put it in a new level, if its new priority level is different from the current one. Note that the new priority level computed is directly proportional to the change in the points-to information. This essentially means that the constraints which add more new points-to pairs are given higher priority. Since adding more edges typically results in the propagation of more points-to information, constraints that add more edges get higher priority.

We explain the computation of the new priority level at Line 11 next. The number of priority levels used in our method requires to be carefully chosen. Keeping this number same as the difference (`diff`) in the points-to information may require a large number of priority levels to be considered, as a few constraints may change hundreds of points-to facts. Moreover, this, in most cases, is unnecessary, as the fixpoint computation benefits from clusters of constraints having approximately the same priorities, rather than an isolated high priority constraint. Therefore, we combine a range of priority values into a single priority level. This *bucketization* proves helpful in skewed evaluation of constraints in a priority level.

Further, bucketization exploits an important observation about constraint solving (as a side-effect): several interdependent constraints, at different times during the analysis, modify the same amount of points-to information. We observed this empirically and on inspection, realized that after initial warm-up, several of the *load* and *store* constraints ($p = *q$ and $*p = q$) start adding a fixed number of points-to information. Therefore, interdependent constraints, due to bucketization, get grouped at the same priority level. Thus, iterating over these interdependent constraints helps in reaching the fixpoint faster.

However, the importance of iterating over the constraints at the same priority level should not be over-emphasized. Due to the cyclic nature of (self or transitive) dependence, in most cases, it suffices to iterate only twice over the constraints at a priority level. The number of indirections present in hand-written programs is typically quite small. Hence, looping beyond two iterations gradually reduces the gain (the amount of points-to information added). Therefore, the `repeat-until` loop from Lines 8–15 iterates for at most `threshold` number of times over the constraints at the same priority level. The condition `inner fixpoint` takes care of not iterating an $(i+1)^{th}$ time if the i^{th} iteration does not change the points-to information.

We remark that the priority scheme does not require the constraint graph explicitly. The algorithm works on constraints rather than on individual pointers.

A. Algorithm Complexity

Let N be the number of pointer variables and M be the number of points-to constraints in a program. Finding dependence across constraints in Line 3 of Algorithm 2 is $O(M^2N)$. Each call to `evaluate` in Line 10 requires $O(N)$ time. `for`-loop at Line 9 executes L times where L is the number of priority levels. The inner `repeat-until` loop executes at most `threshold` number of times while the outer one at Line 5 executes $O(M)$ times. Thus, the complexity of the algorithm is $O(M^2N + M \times L \times \texttt{threshold} \times L \times N)$, which is $O(M^2N + L^2MN)$ since `threshold` is typically a small number (in our implementation it is 2). A useful heuristic is to choose $L^2 \approx O(M)$ in which case, it simplifies to $O(M^2N)$. Assuming $M \approx O(N)$, the complexity becomes $O(N^3)$, same as that of Andersen's analysis [1] or Deep Propagation [7].

V. Experimental Evaluation

We evaluate the effectiveness of prioritized points-to analysis using 16 SPEC C/C++ benchmarks and five large open source programs (*httpd*, *sendmail*, *gdb*, *wine-server* and *ghostscript*). The characteristics of the benchmark programs are given in Table I. *KLOC* is the kilo lines of unprocessed source code. *Total Inst* is the total number of instructions in the LLVM [15] intermediate code after optimizing at -O2 level. *Pointer Inst* is the total number of pointer instructions processed by the analysis. *Func* is the number of functions defined in each program. We evaluate the impact of our prioritization approach on the following methods.

- *anders*: This is the base Andersen's algorithm [1] that uses a simple iterative procedure over the points-to constraints to reach a fixpoint solution. The underlying data structure used is a sorted vector of pointees per pointer. Our implementation of *anders* incorporates difference propagation to propagate only the changed points-to

Benchmark	KLOC	# Total Inst	# Pointer Inst	# Func
gcc	222.185	328,425	119,384	1,829
perlbmk	81.442	143,848	52,924	1,067
gap	71.367	118,715	39,484	877
vortex	67.216	75,458	16,114	963
mesa	59.255	96,919	26,076	1,040
crafty	20.657	28,743	3,467	136
twolf	20.461	49,507	15,820	215
vpr	17.731	25,851	6,575	228
eon	17.679	126,866	43,617	1,723
ammp	13.486	26,199	6,516	211
parser	11.394	35,814	11,872	356
gzip	8.618	8,434	991	90
bzip2	4.650	4,832	759	90
mcf	2.414	2,969	1,080	42
equake	1.515	3,029	985	40
art	1.272	1,977	386	43
httpd	125.877	220,552	104,962	2,339
sendmail	113.264	171,413	57,424	1,005
ghostscript	438.204	906,398	488,998	6,991
gdb	474.591	576,624	362,171	7,127
wine-server	178.592	110,785	66,501	2,105

TABLE I
BENCHMARK CHARACTERISTICS

information and implements online cycle elimination [8] and offline variable substitution [16].

- *bddlcd*: This is *Lazy Cycle Detection* (LCD) algorithm implemented using BDD from Hardekopf and Lin [10] obtained from the first author's website [17]. We extend it for context-sensitivity.
- *bloom*: The bloom filter method uses an approximate representation for storing both the points-to facts and the context information using a bloom filter. As this representation results in false-positives, the method is approximate and introduces precision loss. For our experiments, we use the *medium* configuration [11] which results in less than 2% precision loss for the chosen benchmarks.
- *deep*: This is the context-insensitive Deep Propagation method [7] obtained from the first author's website [18]. The idea is to propagate points-to information in the constraint graph to all the reachable nodes along a path, before the other paths are considered. It uses a sparse bitmap representation to store points-to sets.

deep is context-insensitive while other implementations are context-sensitive; further, all methods are flow-insensitive and field-insensitive. Context-sensitivity is implemented using an invocation-graph based approach [19]. For each method *a*, we denote its prioritized version as *p-a*. All prioritized versions implement effect-driven scheme with all the optimizations described in Section IV with the number of priority levels set to 203 (see Section V-C). The experiments are carried out on an Intel Xeon machine with 2 GHz clock, 4 MB L2 cache and 4 GB RAM.

A. Analysis Time

The analysis times (in seconds) of various methods are shown in Table II. Comparing *anders* versus *p-anders*, we observe a 13%–41% reduction in the analysis time (average 33%) due to prioritized scheduling of points-to constraints. This emphasizes the importance of a good constraint order.

In case of *bddlcd*, we observe a larger benefit due to prioritization (44% on an average, excluding *gdb*). The benefit is an outcome of an interplay between *bddlcd* algorithm and prioritized scheduling. Cycle detection benefits from evaluating the cyclic constraints together which change an equal number of points-to pairs and hence get grouped into the same priority level. The prioritized scheduling approach evaluates all of them in close-proximity, often giving correct hints to the cycle detection mechanism, resulting in an overall efficient analysis. In case of *gdb*, the prioritized LCD version goes out of memory (see discussion in Section V-B).

In case of *bloom*, both the versions (*bloom* and *p-bloom*) analyze all the benchmarks successfully to completion, with *p-bloom* achieving 20% reduction in the analysis time.

The execution times of context-insensitive analysis (*deep* and *p-deep*) are significantly lower due to the relatively lower computational requirements of the context-insensitive algorithm. But even in this case, introducing prioritization results in an improvement that is either smaller for benchmarks which require few hundred milliseconds analysis time or not observed (for the benchmarks where the analysis time is a few milliseconds). [1] However, for the larger benchmarks such as *httpd*, *ghostscript* and *gdb*, the improvements are significant, resulting in more than 50% reduction in the analysis time. The reason is quite similar to that in case of *bddlcd*. Online cycle detection implemented as part of *deep* benefits from prioritized scheduling. However, another artifact of *deep* facilitates higher benefits with prioritized scheduling. Deep Propagation works on the topological ordering of the directed *copy* edges across pointer nodes in the constraint graph. Effect-driven prioritization of constraints adds *copy* edges that propagate (approximately) the same number of points-to constraints in an iteration, resulting in most of the propagation path available for deep propagation.

B. Memory

The memory requirements (in MB) for various methods are shown in Table III. In general, one would expect the memory requirements to remain almost the same. However, the internal structure and implementation of the algorithm play a key role in the overall memory requirement.

p-anders consistently requires significantly less memory than *anders*. On an average, the memory requirement for the prioritized version gets reduced by 17%. The memory savings are largely due to the difference propagation and the use of temporary data structures during constraint evaluation. Instead of keeping small difference information for propagation across iterations as in *anders*, our effect-driven prioritized scheme combines difference information together and propagates them along complete paths in consecutive evaluations. This benefit

[1] Prioritization improves the performance even in context-insensitive versions of *anders*, *bloom* and *bddlcd*, although by a smaller margin. Due to space limitation, we do not report those numbers here.

Benchmark	Context-sensitive						Context-insensitive	
	anders	p-anders	bddlcd	p-bddlcd	bloom	p-bloom	deep	p-deep
gcc	329.463	286.474	17,411.208	7,984.474	10,237.702	8,534.893	1.740	1.176
perlbmk	143.448	98.375	5,879.913	3,159.513	2,632.044	2,144.364	1.744	1.396
vortex	91.283	69.732	4,725.745	3,397.158	1,998.501	1,693.492	0.116	0.088
eon	93.495	79.264	2,391.831	1,515.148	1,241.602	848.439	11.701	2.320
parser	35.445	26.387	618.337	330.953	145.777	124.844	0.176	0.072
gap	128.478	84.963	330.233	186.818	152.102	124.994	0.092	0.044
vpr	29.456	20.119	199.510	95.647	88.826	63.339	0.024	0.008
crafty	29.337	22.128	154.983	91.551	46.899	34.436	0.004	0.004
mesa	89.388	65.143	21.732	12.095	10.041	8.945	0.248	0.108
ammp	34.236	23.285	54.648	31.399	15.185	12.586	0.032	0.012
twolf	41.499	33.774	27.375	13.470	5.132	4.031	0.032	0.016
gzip	25.234	14.885	6.533	3.134	1.808	1.170	0.004	0.004
bzip2	23.322	13.968	4.703	3.907	1.348	1.199	0.004	0.004
mcf	22.395	17.147	32.049	18.384	5.040	4.805	0.004	0.004
equake	24.306	17.178	4.054	3.665	1.100	0.866	0.004	0.004
art	26.459	19.153	7.678	4.144	2.400	2.004	0.004	0.004
httpd	224.534	193.287	47.399	24.785	52.793	42.785	53.727	23.722
sendmail	172.743	136.246	117.528	96.590	25.346	17.867	12.729	10.613
ghostscript	4,384.238	3,183.843	20,612.772	12,371.973	2,597.863	2,101.794	207.03	126.140
gdb	9,338.228	5,847.285	24,871.681	OOM	22,847.375	18,035.790	587.829	294.066
wine-server	201.323	147.289	36.689	23.499	23.686	18.398	8.165	5.488
average	737.539	495.235	3,693.171	1468.415*	2,006.313	1,610.526	42.162	22.157

* The average is calculated ignoring the *OOM* entry.

TABLE II
ANALYSIS TIME (SECONDS)

is similar in spirit to what Deep Propagation achieves over Wave Propagation [7].

Both *bloom* and *p-bloom* complete successfully on all the benchmarks and do not use difference propagation. Hence the memory requirements are quite similar (555 MB versus 539 MB on an average).

p-deep outperforms *deep* in terms of the memory requirement by 23% on an average. Memory savings are largely due to difference propagation.

On the other hand, *p-bddlcd* requires 45% more memory than *bddlcd*. In fact, in case of *gdb*, *p-bddlcd* runs out of memory whereas the non-prioritized version completes successfully. Unlike other algorithms discussed here, *bddlcd* is worklist based. A constraint may get added to the worklist while its another instance is already present. Thus, the worklist size is not bound by the total number of points-to constraints. Having a prioritized scheme requires multiple such worklists to be created, pushing different instances of the same constraint into different worklists based on the current priority of the constraint. Thus, using multiple worklists increases the amount of memory consumed.

C. Effect of configuration parameters

Effect of bucketization. First, we experimented with several values for the number of priority levels. The sensitivity of the analysis time (execution time to complete the points-to analysis) to the number of priority levels (buckets) for *p-anders* is shown in Figure 2. Note that the values are normalized with respect to *anders*. To avoid clutter, we show the effect on only four representative benchmarks *ghostscript*, *gdb*, *perlbmk* and *gzip* along with the average over all the benchmarks listed in Table I. We observe that the analysis

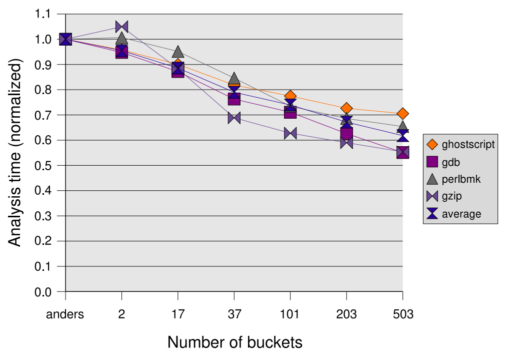


Fig. 2. Effect of bucketization

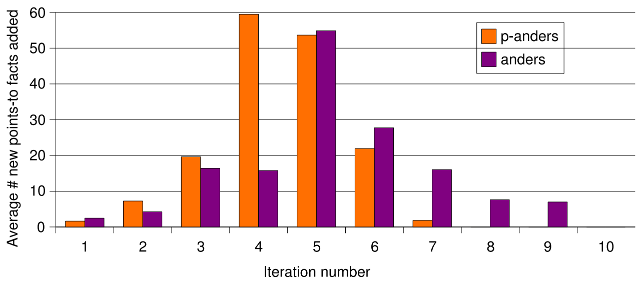


Fig. 3. Effect of prioritization

time steadily reduces with the increasing number of buckets. However, the number of buckets should not be arbitrarily increased. It is important to keep related constraints together so that an inner fixpoint over the related constraints would be beneficial (Section V-A). Using too many priority levels may move related constraints in different priority levels and would

	Context-sensitive						Context-insensitive	
Benchmark	anders	p-anders	bddlcd	p-bddlcd	bloom	p-bloom	deep	p-deep
gcc	2859	2174	2633	3794	1202	1192	83	73
perlbmk	2133	1878	1888	3223	502	499	100	93
vortex	1857	1553	1527	2284	231	222	16	16
eon	1276	907	2798	3697	414	408	248	66
parser	478	419	1016	1438	149	142	4	4
gap	457	397	1289	1680	301	298	8	8
vpr	735	688	964	1356	112	112	2	2
crafty	672	600	739	935	96	95	1	1
mesa	894	825	1682	2466	223	220	14	14
ammp	427	372	935	1330	103	102	3	2
twolf	624	485	926	1256	153	148	4	4
gzip	514	446	802	1053	71	70	1	1
bzip2	633	582	693	1009	68	69	1	1
mcf	403	379	551	716	68	70	1	1
equake	546	501	593	953	68	68	1	1
art	597	524	664	972	65	65	1	1
httpd	791	686	1156	1754	739	736	674	425
sendmail	914	799	1592	2425	442	438	256	224
ghostscript	1958	1644	2470	3528	2322	2317	2871	2364
gdb	2194	1635	3299	OOM	3931	3667	3556	2765
wine-server	774	615	1182	1886	385	378	185	149
average	1035	862	1400	1888*	555	539	382	296

* The average is calculated ignoring the *OOM* entry.

TABLE III
MEMORY REQUIREMENT (MB)

reduce the benefit of the inner fixpoint. Further, after a point, increasing the number of buckets starts giving diminishing returns.

Effect of skewed evaluation. We measure for each iteration the number of times (all the constraints in) each priority level reaches an inner fixpoint in the second iteration. We observe that on an average around 84% of the occupied priority levels reach fixpoint in second iteration. This shows the effectiveness of our prioritized ordering which enables faster fixpoint computation.

We would like to note that values of the configuration parameters play an important role in the analysis efficiency and must be chosen carefully. However, our experience suggests that the parameters vary according to the program characteristics and there is no simple rule to arrive at optimal values for all the programs.

Effect of prioritization. We counted the average number of new points-to facts generated by each constraint for *anders* and *p-anders* in each iteration for *vortex* (see Figure 3).[2] Due to prioritizing appropriate constraints, *p-anders* computes the points-to facts faster and reaches the fixpoint in 8 iterations compared to 10 as in *anders*. A similar behavior is observed for other benchmarks.

D. Comparison with priority queue

In all our experiments the buckets are implemented as a hashtable wherein the priority level acts as the key. One may argue that the prioritization can possibly be more efficiently implemented as a priority queue. Theoretically, a priority queue incurs, on an average, an O(log n) complexity for each insertion and removal of an element. In our hashtable-based implementation, insertion and removal are O(1) operations, given a reference to a constraint. To study how the two implementations perform in practice, we developed a priority queue-based analysis with C++ STL. Our preliminary results on SPEC 2000 benchmarks indicate that the hashtable-based Andersen's analysis (*p-anders*) requires 33% – 93% lesser time than the priority queue-based analysis. Although more experimentation is necessary to draw further conclusions, it seems clear that the priority levels implemented as a hashtable perform better than that implemented as a true priority queue.

[2]Due to constraints getting evaluated multiple times in *p-anders*, the notion of iteration is not well defined.

VI. RELATED WORK

An excellent survey on pointer analysis techniques is presented by Hind and Pioli [20].

Several novel techniques have been developed to improve upon the original Andersen's analysis [4], [21], [22], [23]. Binary Decision Diagrams (BDD) [4], [23] are used to store points-to information in a succinct manner. The idea of *bootstrapping* [6] uses divide and conquer strategy by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [2]) 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 the alias partitions can be done in parallel. Nasre et al. [24] 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. Therefore, approximate representations have been introduced to trade off precision for scalability. Das [3] proposed *one level flow*, Lattner et al. [25] unified contexts, while Nasre et al. [11] hashed contexts to alleviate the need to store the complete context information.

Inclusion based analysis can also 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 [16] operates over constraints prior to the constraint evaluation to find out pointer equivalent variables. Except for the offline variable substitution, all the other enhancements operate dynamically on the constraint evaluations. Our linguistic scheme based on dependences across constraints falls in the offline category. However, in general, similar to the effect-driven scheme, it is dynamic in nature and can be used online to update the constraint dependences as more points-to information gets computed. The linguistic scheme also finds resemblance with the node listing approach [26] for dataflow analysis of structured programs.

Wave and Deep Propagation techniques [7] perform a breadth-wise and depth-wise propagation of points-to information in a constraint graph. Various techniques proposed for worklist management [9] also identify heuristics to reach the fixpoint faster. Specifically, Greatest Output Rise (GOR) algorithm comes close to our effect-driven priority scheme. However, similar to Deep and Wave Propagation, the worklist management algorithms deal with the propagation of points-to information in the constraint graph and are orthogonal to our prioritized analysis of the points-to constraints. Our prioritization framework is more comprehensive and applicable to a variety of existing techniques.

VII. Conclusions

In this paper, we proposed a prioritized order of processing constraints in the points-to analysis method to improve its efficiency. First, we proved that finding an optimal sequence of points-to constraints for even a restricted flow-insensitive version is NP-Complete. Subsequently, we identified two new dimensions for evaluating points-to constraints: how many edges a constraint adds and where in the constraint graph it adds edges. Based on this observation, we presented a prioritization framework for evaluating a set of points-to constraints. We illustrated the generality of the proposed framework by implementing prioritized versions of Andersen's analysis, Lazy Cycle Detection using BDD, Bloom-filter based analysis and Deep Propagation. Experimental evaluation shows that the presented priority scheme can greatly benefit the state-of-the-art algorithms to reach a fixpoint faster. In addition to improving the analysis time, the proposed approach also reduces the memory requirement of the algorithms that use difference propagation.

While the framework is illustrated in the context of points-to analysis, the idea of prioritized evaluation is general and applicable to other static and dynamic analyses. We believe that further work on prioritizing constraints can open up interesting possibilities for performing optimizations.

Acknowledgment

We thank Aditya Kanade, Akash Lal, Aditya Thakur and Kapil Vaswani for their helpful comments on an initial draft of the paper.

References

[1] L. O. Andersen, "Program analysis and specialization for the C programming language," in PhD Thesis, DIKU, University of Copenhagen, 1994.
[2] B. Steensgaard, "Points-to analysis in almost linear time," in POPL, 1996, pp. 32–41.
[3] M. Das, "Unification-based pointer analysis with directional assignments," in PLDI, 2000, pp. 35–46.
[4] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," in PLDI, 2003.
[5] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in PLDI, 2004, pp. 131–144.
[6] V. Kahlon, "Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis," in PLDI, 2008.
[7] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in CGO, 2009, pp. 126–135.
[8] M. Fähndrich, J. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in PLDI, 1998.
[9] A. Kanamori and D. Weise, "Worklist management strategies for dataflow analysis," in MSR Technical Report, MSR-TR-94-12, 1994.
[10] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in PLDI, 2007, pp. 290–299.
[11] R. Nasre, K. Rajan, R. Govindarajan, and U. P. Khedker, "Scalable context-sensitive points-to analysis using multi-dimensional bloom filters," in APLAS, 2009.
[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to algorithms," in McGraw Hill, 2009.
[13] B. Awerbuch, Y. Azar, A. Fiat, and T. Leighton, "Making commitments in the face of uncertainty: how to pick a winner almost every time (extended abstract)," in STOC, 1996.
[14] B. Hardekopf and C. Lin, "Exploiting pointer and location equivalence to optimize pointer analysis," in SAS, 2007.
[15] LLVM, "The LLVM compiler infrastructure," in http://llvm.org, .
[16] A. Rountev and S. Chandra, "Off-line variable substitution for scaling points-to analysis," in PLDI, 2000, pp. 47–56.
[17] B. Hardekopf, "http://www.cs.utexas.edu/users/benh/."
[18] Deep-Propagation, "http://compilers.cs.ucla.edu/fernando/projects/pta/."
[19] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in PLDI, 1994, pp. 242–256.
[20] M. Hind and A. Pioli, "Which pointer analysis should i use?" in ISSTA, 2000, pp. 113–123.
[21] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using CLA: a million lines of C code in a second," in PLDI, 2001.
[22] O. Lhotak and L. Hendren, "Scaling Java points-to analysis using spark," in CC, 2003.
[23] J. Whaley and M. Lam, "An efficient inclusion-based points-to analysis for strictly-typed languages," in SAS, 2002.
[24] R. Nasre and G. Ramaswamy, "Points-to analysis as a system of linear equations," in SAS, 2010.
[25] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in PLDI, 2007, pp. 278–289.
[26] K. W. Kennedy, "Node listings applied to data flow analysis," in POPL, 1975, pp. 10–21.