The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code

Ben Hardekopf
University of Texas at Austin
benh@cs.utexas.edu

Calvin Lin
University of Texas at Austin
lin@cs.utexas.edu

Abstract

*Pointer information is a prerequisite for most program analyses, and the quality of this information can greatly affect their precision and performance. Inclusion-based (*i.e. *Andersen-style) pointer analysis is an important point in the space of pointer analyses, offering a potential sweet-spot in the trade-off between precision and performance. However, current techniques for inclusion-based pointer analysis can have difficulties delivering on this potential.*

We introduce and evaluate two novel techniques for inclusion-based pointer analysis—one lazy, one eager[1]*—that significantly improve upon the current state-of-the-art without impacting precision. These techniques focus on the problem of online cycle detection, a critical optimization for scaling such analyses. Using a suite of six open-source C programs, which range in size from 169K to 2.17M LOC, we compare our techniques against the best three current inclusion-based analyses—described by Heintze and Tardieu [11], by Pearce* et al. *[22], and by Berndl et al. [4]. The combination of our two techniques results in an algorithm which is on average* $3.2\times$ *faster than Heintze and Tardieu's algorithm,* $6.4\times$ *faster than Pearce et al.'s algorithm, and* $20.6\times$ *faster than Berndl* et al.*'s algorithm.*

We also investigate the use of different data structures to represent points-to sets, examining the impact on both performance and memory consumption. We compare a sparse-bitmap implementation used in the GCC compiler with a BDD-based implementation, and we find that the BDD implementation is on average $2\times$ *slower than using sparse bitmaps but uses* $5.5\times$ *less memory.*

1. Introduction

Pointer information is a prerequisite for most program analyses, including modern whole-program analyses such as program verification and program understanding. The precision and performance of these client analyses depend heavily on the precision of the pointer information that they're given [24]. Unfortunately, precise pointer analysis is NP-hard [14]—any practical pointer analysis must approximate the exact solution. There are a number of different approximations that can be made, each with its own trade-off between precision and performance [12].

[1] Hence the reference to Aesop's fable 'The Ant and the Grasshopper' [1].

PLDI '07 June 10–13, 2007, San Diego, California, USA.

The most precise analyses are flow-sensitive—respecting control-flow dependencies—and context-sensitive—respecting the semantics of function calls. Despite a great deal of work on both flow-sensitive and context-sensitive algorithms [6, 8, 13, 15, 20, 27, 28, 29, 30], none has been shown to scale to programs with millions of lines of code, and most have difficulty scaling to 100,000 lines of code.

If flow- and context-sensitivity aren't feasible for large programs, we're left to consider flow- and context-insensitive analyses. The most precise member of this class is inclusion-based (*i.e.*, Andersen-style) pointer analysis [2], which is closely related to computing the dynamic transitive closure of a graph. Inclusion constraints are generated from the program code and used to construct a *constraint graph*, with nodes to represent each program variable and edges to represent inclusion constraints between the variables. Indirect constraints—those involving pointer dereferences—can't be represented, since points-to information isn't yet available. Points-to information is gathered by computing the transitive closure of the graph; as more information is gained, new edges are added to the constraint graph to represent the indirect constraints. The transitive closure of the final graph yields the points-to solution. The exact algorithm is explained in Section 3.

Inclusion-based pointer analysis has a complexity of $O(n^3)$; the key to making it scalable is to reduce the input size—*i.e.* make n smaller—while maintaining soundness. The primary method used to reduce n is online cycle detection: the analysis looks for cycles in the constraint graph and collapses their components into single nodes. Because the algorithm computes the transitive closure, all nodes in the same cycle are guaranteed to have identical points-to sets and can safely be collapsed together. The method used to find and collapse cycles during the analysis has a significant impact on the algorithm's performance.

In this paper we introduce a new inclusion-based pointer analysis algorithm that employs a novel method of detecting cycles called *Lazy Cycle Detection* (LCD). Rather than aggressively seeking out cycles in the constraint graph, LCD piggybacks on top of the transitive closure computation, identifying potential cycles based on their effect—identical points-to sets. This lazy nature significantly reduces the overhead of online cycle detection.

We also introduce a second method for detecting cycles called *Hybrid Cycle Detection* (HCD). Hybrid Cycle Detection offloads work to a linear-time offline analysis—a static analysis done prior to the actual pointer analysis. The actual pointer analysis is then able to detect cycles without performing any graph traversal. Thus, HCD eagerly pays a small up-front cost to avoid a large amount of later work. While HCD can be used on its own, its true power lies in the fact that it can easily be combined with other inclusion-based pointer analyses to significantly improve their performance.

We compare our new techniques against a diverse group of inclusion-based pointer analyses representing the current state-

of-the-art. This group includes algorithms due to Heintze and Tardieu [11] (HT), Pearce *et al.* [22] (PKH), and Berndl *et al.* [4] (BLQ). All of these algorithms are explained in Section 2.

This paper makes the following contributions to inclusion-based pointer analysis:

- We introduce Lazy Cycle Detection, which recognizes that the effects of cycles—identical points-to sets—can be used to detect them with extremely low overhead. On average LCD is faster than all three current state-of-the-art inclusion-based analyses: 1.05× faster than HT, 2.1× faster than PKH, and 6.8× faster than BLQ.
- We introduce Hybrid Cycle Detection, which dramatically reduces the overhead of online cycle detection by carefully partitioning the task into offline and online analyses. On average HCD improves the performance of HT by 3.2×, PKH by 5×, BLQ by 1.1×, and LCD by 3.2×. HCD is the first ever technique for cycle detection that has been shown to be practical for BDD-based program analyses like BLQ.
- We provide the first empirical comparison of the three current state-of-the-art inclusion-based pointer analysis algorithms, namely, HT, PKH, and BLQ. We find that HT is the fastest—1.9× faster than PKH and 6.5× faster than BLQ.
- We demonstrate that an algorithm that combines Lazy Cycle Detection and Hybrid Cycle Detection (LCD+HCD) is the fastest of the algorithms that we studied and can easily scale to programs consisting of over a million lines of code. It is on average 3.2× faster than HT, 6.4× faster than PKH, and 20.6× faster than BLQ.
- We investigate the memory consumption of the various analyses, and experiment with two different data structures for representing points-to sets: sparse bitmaps as currently used in the GCC compiler, and a BDD-based representation. For the algorithms that we study, we find that the BDD-based representation is an average of 2× slower than sparse bitmaps, but uses 5.5× less memory.

The rest of this paper is organized as follows. In Section 2 we place our techniques in the context of prior work. Section 3 provides background about inclusion-based pointer analysis. Section 4 describes our two new techniques for detecting cycles, and Section 5 presents our experimental evaluation.

2. Related Work

Inclusion-based pointer analysis is described by Andersen in his Ph.D. thesis [2], in which he formulates the problem in terms of type theory. The algorithm presented in the thesis solves the inclusion constraints in a fairly naive manner by repeatedly iterating through a constraint vector. Cycle detection is not mentioned. There have been several significant updates since that time.

Faehndrich *et al.* [9] represent the constraints using a graph and formulate the problem as computing the dynamic transitive closure of that graph. This work introduces a method for partial online cycle detection and demonstrates that cycle detection is critical for scalability. A depth-first search of the graph is performed upon every edge insertion, but the search is artificially restricted for the sake of performance, making cycle detection incomplete.

Heintze and Tardieu introduce a new algorithm for computing the dynamic transitive closure [11]. As new inclusion edges are added to the constraint graph from the indirect constraints, their corresponding new transitive edges are not added to the graph. Instead, the constraint graph retains its pre-transitive form, and during the analysis, indirect constraints are resolved via reachability queries on the graph. Cycle detection is performed as a side-effect of these queries. The main drawback to this technique is unavoidable redundant work—it is impossible to know whether a reachability query will encounter a newly-added inclusion edge (inserted earlier due to some other indirect constraint) until after it completes, which means that potentially redundant queries must still be carried out on the off-chance that a new edge will be encountered. Heintze and Tardieu report excellent results, analyzing a C program with 1.3M LOC in less than a second, but these results are for a field-based implementation. A field-based analysis treats each field of a struct as its own variable—assignments to $x.f$, $y.f$, and $(*z).f$ are all treated as assignments to a variable f, which tends to decrease both the size of the input to the pointer analysis and the number of dereferenced variables (an important indicator of performance). Field-based analysis is unsound for C programs, and while such an analysis is appropriate for the work described by Heintze and Tardieu (the client is a dependency analysis that is itself field-based), it is inappropriate for many others. For the results in this paper, we use a field-insensitive version of their algorithm, which is dramatically slower than the field-based version[2].

Pearce *et al.* have proposed two different approaches to inclusion-based analysis, both of which differ from Heintze and Tardieu in that they maintain the explicit transitive closure of the constraint graph. They originally proposed an analysis [21] that used a more efficient algorithm for online cycle detection than that introduced by Faehndrich *et al.* [9]. In order to avoid cycle detection at every edge insertion, the algorithm dynamically maintains a topological ordering of the constraint graph. Only a newly-inserted edge that violates the current ordering could possibly create a cycle, so only in this case are cycle detection and topological re-ordering performed. This algorithm proved to still have too much overhead, so Pearce *et al.* later proposed a new and more efficient algorithm [22]. Rather than detect cycles at every edge insertion, the entire constraint graph is periodically swept to detect and collapse any cycles that have formed since the last sweep. It is this algorithm that we evaluate in this paper.

Berndl *et al.* [4] describe a field-sensitive inclusion-based pointer analysis for Java that uses BDDs [5] to represent both the constraint graph and the points-to solution. BDDs have been extensively used in model checking as a way to represent large graphs in a very compact form that allows for fast manipulation. Berndl *et al.* were one of the first to use BDDs for pointer analysis. The analysis they describe is specific to the Java language; it also doesn't handle indirect calls because it depends on a prior analysis to construct the complete call-graph. The version of the algorithm that we use in this paper is a field-insensitive analysis for C programs that does handle indirect function calls.

Because Andersen-style analysis was previously considered to be non-scalable, other algorithms, including Steensgaard's near-linear time analysis [25] and Das' One-Level Flow analysis [7], have been proposed to improve performance by sacrificing even more precision. While Steensgaard's analysis has much greater imprecision than inclusion-based analysis, Das reports that for C programs One-Level Flow analysis has precision very close to that of inclusion-based analysis. This precision is based on the assumption that multi-level pointers are less frequent and less important than single-level pointers, which Das' experiments indicate is usually (though not always) true for C, but which may not be true for other languages such as Java and C++. In addition, for the sake of performance Das conservatively unifies non-equivalent variables, much like Steensgaard's analysis; this unification makes it difficult

[2] To ensure that the performance difference is in fact due to field-insensitivity, we also benchmarked a field-based version of our HT implementation. We observed comparable performance to that reported by Heintze and Tardieu [11].

Constraint Type	Program Code	Constraint	Meaning
Base	$a = \&b$	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex$_1$	$a = *b$	$a \supseteq *b$	$\forall v \in pts(b) : pts(a) \supseteq pts(v)$
Complex$_2$	$*a = b$	$*a \supseteq b$	$\forall v \in pts(a) : pts(v) \supseteq pts(b)$

Table 1. Constraint Types

to trace dependency chains among variables. Dependency chains are very useful for understanding the results of program analyses such as program verification and program understanding, and also for automatic tools such as Broadway [10]. Inclusion-based pointer analysis is a better choice than either Steensgaard's analysis or One-Level Flow, *if* it can be made to run in reasonable time even on large programs with millions of lines of code; this is the challenge that we address in this paper.

In the other direction of increasing precision, there have been several attempts to scale a context-sensitive version of inclusion-based pointer analysis. One of the fastest of these attempts is the the algorithm by Whaley *et al.* [28], which uses BDDs to scale a context-sensitive, flow-insensitive pointer analysis for Java to almost 700K LOC (measuring bytecode rather than source lines). However, Whaley *et al.*'s algorithm is only context-sensitive for top-level variables, meaning that all variables in the heap are treated context-insensitively; also, its efficiency depends heavily on certain characteristics of the Java language—attempts to use the same technique for analyzing programs in C have shown greatly reduced performance [3].

Nystrom *et al.* [20] present a context-sensitive algorithm based on the insight that inlining all function calls makes a context-insensitive analysis equivalent to a context-sensitive analysis of the original program. Of course, inlining all function calls can increase the program size exponentially, but intelligent heuristics can make exponential growth extremely unlikely. An important building block of this approach is context-insensitive inclusion-based analysis—it is used while inlining the functions and also for analyzing the resulting program. Nystrom *et al.* manage to scale the context-sensitive analysis to a C program with 200K LOC. The new techniques described in this paper could be used to scale their algorithm even further.

3. Background

Inclusion-based pointer analysis is a set-constraint problem. A linear pass through the program code generates three types of constraints—*base*, *simple*, and *complex* [11]. We eliminate nested pointer dereferences by introducing auxiliary variables and constraints, leaving only one pointer dereference per constraint. Table 1 demonstrates the three types of constraints, how they are derived from the program code, and what the constraints mean. For a variable v, $pts(v)$ represents v's points-to set and $loc(v)$ represents the memory location denoted by v.

Following the example of prior work in this area [9, 11, 22, 4], we solve the set-constraint problem by computing the dynamic transitive closure of a constraint graph. The constraint graph G has one node for each program variable. For each simple constraint $a \supseteq b$, G has a directed edge $b \rightarrow a$. Each node also has a points-to set associated with it, initialized using the base constraints: for each base constraint $a \supseteq \{b\}$, node a's points-to set contains $loc(b)$. The complex constraints are not explicitly represented in the graph; they are maintained in a separate list.

To solve the constraints we compute the transitive closure of G by propagating points-to information along its edges. As we update the points-to sets, we must also add new edges to represent the complex constraints. For each constraint $a \supseteq *b$ and each $loc(v) \in pts(b)$, we add a new edge $v \rightarrow a$. Similarly, for each constraint $*a \supseteq b$ and each $loc(v) \in pts(a)$, we add a new edge $b \rightarrow v$.

```
let G =< V, E >
W ← V
while W ≠ ∅ do
    n ← SELECT-FROM(W)
    for each v ∈ pts(n) do
        for each constraint a ⊇ *n do
            if v → a ∉ E then
                E ← E ∪ {v → a}
                W ← W ∪ {v}
        for each constraint *n ⊇ b do
            if b → v ∉ E then
                E ← E ∪ {b → v}
                W ← W ∪ {b}
    for each n → z ∈ E do
        pts(z) ← pts(z) ∪ pts(n)
        if pts(z) changed then
            W ← W ∪ {z}
```

Figure 1. Dynamic Transitive Closure

Figure 1 shows a basic worklist algorithm that maintains the explicit transitive closure of G. The worklist is initialized with all nodes in G that have a non-empty points-to set. For each node n taken off the worklist, we proceed in two steps:

1. For each $loc(v) \in pts(n)$: for each constraint $a \supseteq *n$ add an edge $v \rightarrow a$, and for each constraint $*n \supseteq b$ add an edge $b \rightarrow v$. Any node that has had a new outgoing edge added is inserted into the worklist.
2. For each outgoing edge $n \rightarrow v$, propagate $pts(n)$ to node v, *i.e.* $pts(v) := pts(v) \cup pts(n)$. Any node whose points-to set has been modified is inserted into the worklist.

The algorithm is presented as it is for clarity of exposition; various optimizations are possible to improve its performance.

4. Our Solutions

The algorithm shown in Figure 1 computes the dynamic transitive closure of the constraint graph but makes no attempt to detect cycles. The particular method used for detecting cycles will in large part determine the efficiency of the analysis—in fact, without cycle detection our larger benchmarks run out of memory before completing, even on a machine with 2GB of memory. When performing online cycle detection, there is a tension between minimizing the overhead caused by repeatedly sweeping the constraint graph, and yet still finding cycles in a timely manner. We now present two new approaches for online cycle detection that balance this tension in different ways.

```
let G =< V, E >
R ← ∅
W ← V
while W ≠ ∅ do
    n ← SELECT-FROM(W)
    for each v ∈ pts(n) do
        for each constraint a ⊇ ∗n do
            if v → a ∉ E then
                E ← E ∪ {v → a}
                W ← W ∪ {v}
        for each constraint ∗n ⊇ b do
            if b → v ∉ E then
                E ← E ∪ {b → v}
                W ← W ∪ {b}
    for each n → z ∈ E do
        if pts(z) = pts(n) ∧ n → z ∉ R then
            DETECT-AND-COLLAPSE-CYCLES(z)
            R ← R ∪ {n → z}
        pts(z) ← pts(z) ∪ pts(n)
        if pts(z) changed then
            W ← W ∪ {z}
```

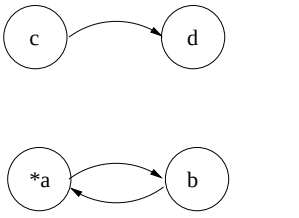
Figure 2. Lazy Cycle Detection

$a = \&c;$
$d = c;$
$b = *a;$
$*a = b;$

(a) Program

$a \supseteq \{c\}$
$d \supseteq c$
$b \supseteq *a$
$*a \supseteq b$

(b) Constraints



(c) Offline Constraint Graph

Figure 3. HCD Offline Analysis Example: (a) Program code; (b) constraints generated from the program code; (c) the offline constraint graph corresponding to the constraints. Note that $*a$ and b are in a cycle together; from this we can infer that in the online constraint graph, b will be in a cycle with all the variables in a's points-to set.

4.1 Lazy Cycle Detection

Cycles in the constraint graph can be collapsed because nodes in the same cycle are guaranteed to have identical points-to sets. We use this fact to create a heuristic for cycle detection: before propagating points-to information across an edge of the constraint graph, we check to see if the source and destination already have equal points-to sets; if so then we use a depth-first search to check for a possible cycle.

This technique is lazy because rather than trying to detect cycles when they are created, *i.e.* when the final edge is inserted that completes the cycle, it waits until the effect of the cycle—identical points-to sets—becomes evident. The advantage of this technique is that we only attempt to detect cycles when we are likely to find them. A potential disadvantage is that cycles may be detected well after they are formed, since we must wait for the points-to information to propagate all the way around the cycle before we can detect it.

The accuracy of this technique depends upon the assumption that two nodes usually have identical points-to sets only because they are in the same cycle; otherwise it would waste time trying to detect non-existent cycles. One additional refinement is necessary to bolster this assumption and make the technique relatively precise: we never trigger cycle detection on the same edge twice. We thus avoid making repeated cycle detection attempts involving nodes with identical points-to sets that are not in a cycle. This additional restriction implies that Lazy Cycle Detection is incomplete—it is not guaranteed to find all cycles in the constraint graph.

The Lazy Cycle Detection algorithm is shown in Figure 2. Before we propagate a points-to set from one node to another, we check to see if two conditions are met: (1) the points-to sets are identical; and (2) we haven't triggered a search on this edge previously. If the conditions are met, then we trigger cycle detection rooted at the destination node. If there exists a cycle, we collapse together all the nodes involved; otherwise we remember this edge so that we won't repeat the attempt later.

4.2 Hybrid Cycle Detection

Cycle detection can be done offline, in a static analysis prior to the actual pointer analysis, such as with Offline Variable Substitution described by Rountev *et al.* [23]. However, many cycles don't exist in the initial constraint graph and only appear as new edges are added during the pointer analysis itself, thus the need for online cycle detection techniques, such as Lazy Cycle Detection. The drawback to online cycle detection is that it requires traversing the constraint graph multiple times searching for cycles; these repeated traversals can become extremely expensive. Hybrid Cycle Detection (HCD) is so-called because it combines both offline and online analyses to detect cycles, thereby getting the best of both worlds—detecting cycles created online during the pointer analysis, without requiring any traversal of the constraint graph.

We now describe the HCD offline analysis, which is a linear-time static analysis done prior to the actual pointer analysis. We build an offline version of the constraint graph, with one node for each program variable plus an additional *ref* node for each variable dereferenced in the constraints (e.g. $*n$). There is a directed edge for each simple and complex constraint: $a \supseteq b$ yields edge $b \rightarrow a$, $a \supseteq *b$ yields edge $*b \rightarrow a$, and $*a \supseteq b$ yields edge $b \rightarrow *a$. Base constraints are ignored. Figure 3 illustrates this process.

Once the graph is built we detect strongly-connected components (SCCs) using Tarjan's linear-time algorithm [26]. Any SCCs containing only non-ref nodes can be collapsed immediately. SCCs containing ref nodes are more problematic: a ref node in the offline constraint graph is a stand-in for a variable's unknown points-to set, e.g. the ref node $*n$ stands for whatever n's points-to set will be when the pointer analysis is complete. An SCC containing a ref node such as $*n$ actually means that n's points-to set is part of the SCC; but since we don't yet know what that points-to set will be, we can't collapse that SCC. The offline analysis knows which variables' points-to sets will be part of an SCC, while the online analysis (*i.e.* the pointer analysis) knows the variables' actual points-to sets. The purpose of Hybrid Cycle Detection is to bridge this gap. Figure 4 shows how the online analysis is affected when an SCC contains a ref node in the offline constraint graph.

We finish the offline analysis by looking for SCCs in the offline constraint graph that consist of more than one node and that also contain at least one ref node. Because there are no constraints of the form $*p \supseteq *q$, no ref node can have a reflexive edge and any non-trivial SCC containing a ref node must also contain a non-ref node. For each SCC of interest we select one non-ref node b, and for each ref node $*a$ in the same SCC, we store the tuple (a, b) in a list L. This tuple signifies to the online analysis that a's points-

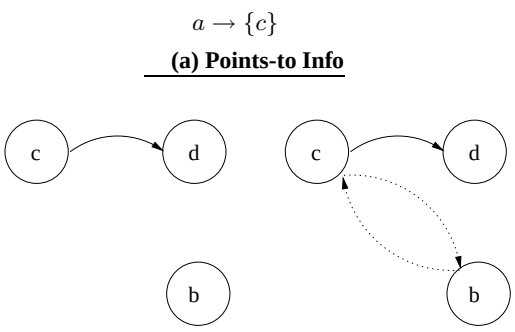


Figure 4. HCD Online Analysis Example: (a) The initial points-to information from the constraints in Figure 3; (b) the online constraint graph before any edges are added; (c) the online constraint graph after the edges are added due to the complex constraints in Figure 3. Note that c and b are now in a cycle together.

```
let G =< V, E >
W ← V
while W ≠ ∅ do
    n ← SELECT-FROM(W)
    if (n, a) ∈ L then
        for each v ∈ pts(n) do
            COLLAPSE(v, a)
            W ← W ∪ {a}
    for each v ∈ pts(n) do
        for each constraint a ⊇ ∗n do
            if v → a ∉ E then
                E ← E ∪ {v → a}
                W ← W ∪ {v}
        for each constraint ∗n ⊇ b do
            if b → v ∉ E then
                E ← E ∪ {b → v}
                W ← W ∪ {b}
    for each n → z ∈ E do
        pts(z) ← pts(z) ∪ pts(n)
        if pts(z) changed then
            W ← W ∪ {z}
```

Figure 5. Hybrid Cycle Detection

to set belongs in an SCC with b, and therefore everything in a's points-to set can safely be collapsed with b.

The online analysis is shown in Figure 5. The algorithm is similar to the basic algorithm shown in Figure 1, except when processing node n we first check L for a tuple of the form (n, a). If one is found then we preemptively collapse together node a and all members of n's points-to set, knowing that they belong to the same cycle. For simplicity's sake the pseudo-code ignores some obvious optimizations.

Hybrid Cycle Detection is not guaranteed to find all cycles in the online constraint graph, only those that can be inferred from the offline version of the graph. Those cycles that it does find, however, are discovered at the earliest possible opportunity and without requiring any traversal of the constraint graph. In addition, while HCD can be used on its own as shown in Figure 5, it can also be easily combined with other algorithms such as HT, PKH, BLQ, and LCD to enhance their performance.

5. Evaluation

5.1 Methodology

To compare the various inclusion-based pointer analyses, we implement field-insensitive versions of five main algorithms: Heintze and Tardieu (HT), Berndl *et al.* (BLQ), Pearce *et al.* (PKH), Lazy Cycle Detection (LCD), and Hybrid Cycle Detection (HCD). We also implement four additional algorithms by integrating HCD with four of the main algorithms: HT+HCD, PKH+HCD, BLQ+HCD, and LCD+HCD. The algorithms are written in C++ and handle all aspects of the C language except for varargs. They use as many common components as possible to provide a fair comparison, and they have all been highly optimized. The source code is available from the authors upon request. Some highlights of the implementations include:

- Indirect function calls are handled as described by Pearce et al [22]. Function parameters are numbered contiguously starting immediately after their corresponding function variable, and when resolving indirect calls they are accessed as offsets to that function variable.
- Cycles are detected using Nuutila *et al.*'s [19] variant of Tarjan's algorithm, and they are collapsed using a union-find data structure with both union-by-rank and path compression heuristics.
- BLQ uses the incrementalization optimization described by Berndl *et al.* [4]. We use the BuDDy BDD library [16] to implement BDDs.
- LCD and HCD are both worklist algorithms—we use the worklist strategy LRF,[3] suggested by Pearce *et al.* [21], to prioritize the worklist. We also divide the worklist into two sections, *current* and *next*, as described by Nielson *et al.* [18]; items are selected from *current* and pushed onto *next*, and the two are swapped when *current* becomes empty. For our benchmarks, the divided worklist yields significantly better performance than a single worklist.
- Aside from BLQ, all the algorithms use sparse bitmaps to implement both the constraint graph and the points-to sets. The sparse bitmap implementation is taken from the GCC 4.1.1 compiler.
- We also experiment with the use of BDDs to represent the points-to sets. Unlike BLQ, which stores the entire points-to solution in a single BDD, we give each variable its own BDD to store its individual points-to set. For example, if $a \rightarrow \{b, c\}$ and $d \rightarrow \{c, e\}$, BLQ would have a single BDD representing the set of tuples $\{(a, b), (a, c), (d, c), (d, e)\}$. Instead, we give a a BDD representing the set $\{b, c\}$ and we give d a BDD representing the set $\{c, e\}$. The use of BDDs instead of sparse bitmaps was a simple modification that required minimal changes to the code.

The benchmarks for our experiments are described in Table 2. Emacs is a text editor; Ghostscript is a postscript viewer; Gimp is an image manipulation program; Insight is a GUI overlaid on top of the gdb debugger; Wine is a Windows emulator; and Linux is the Linux operating system kernel. The constraint generator is separate from the constraint solvers: we generate constraints from the benchmarks using the CIL C front-end [17], ignoring any assignments involving types too small to hold a pointer. External library calls are summarized using hand-crafted function stubs. We pre-process the resulting constraint files using a variant of

[3] **Least Recently Fired**—the node processed furthest back in time is given priority.

Name	LOC	Original Constraints	Reduced Constraints	Base	Simple	Complex
Emacs-21.4a	169K	83,213	21,460	4,088	11,095	6,277
Ghostscript-8.15	242K	169,312	67,310	12,154	25,880	29,276
Gimp-2.2.8	554K	411,783	96,483	17,083	43,878	35,522
Insight-6.5	603K	243,404	85,375	13,198	35,382	36,795
Wine-0.9.21	1,338K	713,065	171,237	39,166	62,499	69,572
Linux-2.4.26	2,172K	574,788	203,733	25,678	77,936	100,119

Table 2. Benchmarks: For each benchmark we show the number of lines of code (computed as the number of non-blank, non-comment lines in the source files), the original number of constraints generated using CIL, the reduced number of constraints after being pre-processed, and a break-down of the forms of the reduced constraints.

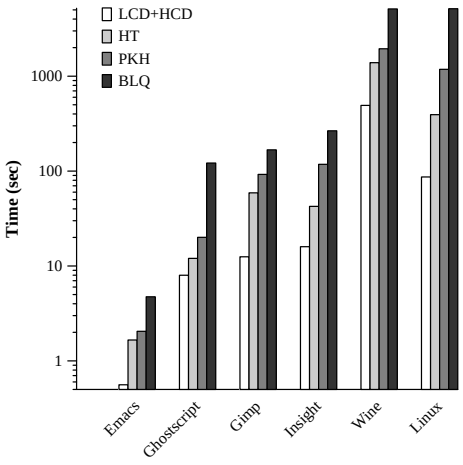


Figure 6. Performance (in seconds) of our new combined algorithm (LCD+HCD) versus three state-of-the art inclusion-based algorithms. Note that the Y-axis is log-scale.

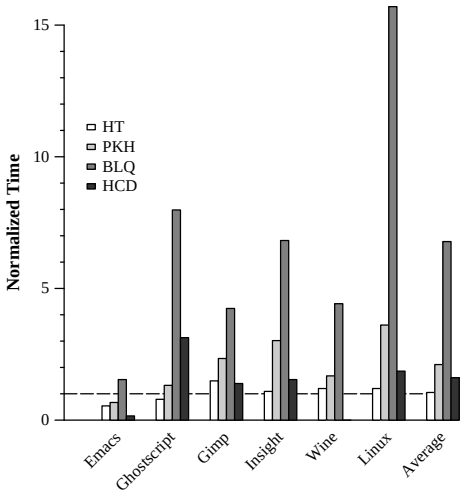


Figure 7. Performance comparison of individual benchmarks, where performance is normalized against LCD. HCD runs out of memory for Wine, so there is no HCD bar for that benchmark.

Offline Variable Substitution [23], which reduces the number of constraints by 60–77%. This pre-processing step takes less than a second for Emacs and Ghostscript, and between 1 and 3 seconds for Gimp, Insight, Wine, and Linux. The results reported are for these reduced constraint files; they include everything from reading in the constraint file from disk, creating the initial constraint graph, and solving that graph.

We run the experiments on a dual-core 1.83 GHz processor with 2 GB of memory, using the Ubuntu 6.10 Linux distribution. Though the processor is dual-core, the executables themselves are single-threaded. All executables are compiled using gcc-4.1.1 and the '–O3' optimization flag. We repeat each experiment three times and report the smallest time; all the experiments have very low variance in performance.

5.2 Time and Memory Consumption

Table 3 shows the performance of the various algorithms. The times for HCD's offline analysis are shown separately and not included in the times for the various algorithms using HCD—they are small enough to be essentially negligible. Table 4 shows the memory consumption of the algorithms. Figure 6 graphically compares (using a log-scale) the performance of our combined algorithm LCD+HCD—the fastest of all the algorithms—against the current state-of-the-art algorithms. Note that all these numbers were gathered using the sparse-bitmap implementations of the algorithms (except for BLQ).

BLQ's memory allocation is fairly constant across all the benchmarks. We allocate an initial pool of memory for the BDDs, which dominates the memory usage and is independent of benchmark size. While we can decrease the initial pool size for the smaller benchmarks without decreasing performance, there is no easy way to calculate what the correct pool size might be for a specific benchmark, so for all the benchmarks we use the smallest pool size that doesn't impair the performance of our largest benchmark.

It is interesting to note the vast difference in analysis time between Wine and Linux for all algorithms other than BLQ. While Wine has 32.5K fewer constraints than Linux, it takes 1.7–7.3× longer to be analyzed, depending on the algorithm used. This discrepancy points out the danger in using the size of the initial input to predict performance when other factors can have at least as much impact. Wine is a case in point: while its initial constraint graph is smaller than that of Linux, its final constraint graph at the end of the analysis is an order-of-magnitude larger than that of Linux, due mostly to Wine's larger average points-to set size. BLQ doesn't display this same behavior, due to its radically different analysis mechanism using BDDs and its lack of cycle detection.

Comparing HT, PKH, BLQ, LCD, and HCD. Figure 7 compares the performance of the main algorithms by normalizing the times for HT, PKH, BLQ, and HCD by that of LCD. Focusing on the current state-of-the-art algorithms, HT is clearly the fastest, being 1.9× faster than PKH and 6.5× faster than BLQ. LCD is on average 1.05× faster than HT and uses 1.2× less memory. HCD runs out of memory for Wine, but excluding that benchmark it is on average 1.8× slower than HT and 1.9× faster than PKH, using 1.4× more memory than HT.

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
HCD-Offline	0.05	0.17	0.26	0.23	0.51	0.62
HT	1.66	12.03	59.00	42.49	1,388.51	393.30
PKH	2.05	20.05	92.30	117.88	1,946.16	1,181.59
BLQ	4.74	121.60	167.56	265.94	5,117.64	5,144.29
LCD	3.07	15.23	39.50	39.02	1,157.10	327.65
HCD	0.46	49.55	59.70	73.92	**OOM**	659.74
HT+HCD	0.46	7.29	11.94	14.82	643.89	102.77
PKH+HCD	0.46	10.52	17.12	21.91	838.08	114.45
BLQ+HCD	5.81	115.00	173.46	257.05	4,211.71	4,581.91
LCD+HCD	0.56	7.99	12.50	15.97	492.40	86.74

Table 3. Performance (in seconds), using bitmaps for points-to sets. Note that the HCD-Offline analysis is reported separately and not included in the times for those algorithms using HCD. The HCD algorithm runs out of memory on the Wine benchmark.

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
HT	17.7	84.9	279.0	231.5	1,867.2	901.3
PKH	17.6	83.9	269.5	194.7	1,448.3	840.7
BLQ	215.6	216.1	216.2	216.1	216.2	216.2
LCD	14.3	74.6	269.0	184.4	1,465.1	830.1
HCD	18.1	138.7	416.1	290.5	**OOM**	1,301.5
HT+HCD	12.4	80.8	253.9	186.5	1,391.4	842.5
PKH+HCD	13.9	79.1	264.6	186.0	1,430.2	807.5
BLQ+HCD	215.8	216.2	216.2	216.2	216.2	216.2
LCD+HCD	13.9	73.5	263.9	183.6	1,406.4	807.9

Table 4. Memory consumption (in megabytes), using bitmaps for points-to sets..

Effects of HCD. Figure 8 normalizes the performance of the main algorithms by that of their HCD-enhanced counterparts. On average adding HCD increases HT performance by 3.2×, increases PKH performance by 5×, increases BLQ performance by 1.1×, and increases LCD performance by 3.2×. HCD also leads to a small decrease in memory consumption for all the algorithms except BLQ—it decreases memory consumption by 1.2× for HT, by 1.1× for PKH, and by 1.02× for LCD. Most of the memory used by these algorithms comes from the representation of points-to sets. HCD improves performance by finding and collapsing cycles much earlier than normal, but it doesn't actually find many more cycles than were already detected without using HCD, so it doesn't significantly reduce the number of points-to sets that need to be maintained. HCD doesn't improve BLQ's performance by much because even though no extra effort is required to find cycles, there is still some overhead involved in collapsing those cycles. Also, the performance of BLQ depends on the sizes of the BDD representations of the constraint and points-to graphs, and because of the properties of BDDs, removing edges from the constraint graph can potentially increase the size of the constraint graph BDD.

While neither LCD nor HCD is by itself the fastest algorithm, the combination of the two, LCD+HCD, yields the fastest algorithm among all those studied: It is 3.2× faster than HT, 6.4× faster than PKH, and 20.6× faster than BLQ.

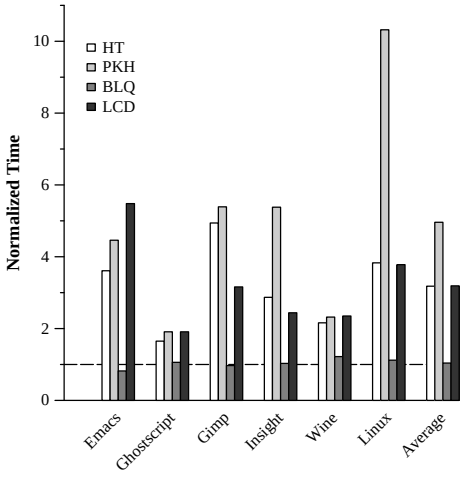


Figure 8. Performance comparison of the individual benchmarks, where the performance of each main algorithm is normalized against its respective HCD-enhanced counterpart.

5.3 Understanding the Results

There are a number of factors that determine the relative performance of these algorithms, but three of the most important are: (1) the number of nodes collapsed due to strongly-connected components; (2) the number of nodes searched during the depth-first traversals of the constraint graph; and (3) the number of propagations of points-to information across the edges of the constraint graph.

The number of nodes collapsed is important because it reduces both the number of nodes and the number of edges in the constraint graph; the more nodes that are collapsed, the smaller the input and the more efficient the algorithm.

The depth-first searches are pure overhead due to cycle detection. As long as roughly as many cycles are being detected, then the fewer nodes that are searched the better.

The number of points-to information propagations is an important metric because propagation is one of the most expensive operations in the analysis. It is strongly influenced by both the number of cycles collapsed and by how quickly they are collapsed. If a cycle

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
HT	3.44	18.55	46.98	65.00	1,551.89	419.38
PKH	4.23	19.55	81.53	96.50	1,172.15	801.13
LCD	4.96	19.34	47.29	64.57	1,213.43	380.26
HCD	3.96	24.65	49.11	65.01	731.20	267.69
HT+HCD	2.58	15.65	33.69	42.33	737.37	209.90
PKH+HCD	3.06	14.70	33.71	43.20	744.35	172.43
LCD+HCD	3.09	13.69	33.04	43.17	625.82	183.97

Table 5. Performance (in seconds), using BDDs for points-to sets.

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
HT	33.1	49.3	100.7	100.0	811.2	274.3
PKH	33.2	33.6	50.4	66.8	226.4	182.1
LCD	33.2	33.2	40.1	33.9	251.1	73.5
HCD	33.1	37.1	36.8	37.0	239.6	65.8
HT+HCD	33.1	37.8	51.2	53.9	410.6	100.7
PKH+HCD	33.1	33.2	36.0	33.2	103.9	45.2
LCD+HCD	33.1	33.2	33.2	33.2	173.6	42.6

Table 6. Memory consumption (in megabytes), using BDDs for points-to sets.

is not detected quickly, then points-to information could be redundantly circulated around the cycle a number of times.

We now examine these three quantities to help explain the performance results seen in the previous section. Due to its radically different analysis mechanism, we don't include BLQ in this examination. [4]

Nodes Collapsed. PKH is the only algorithm guaranteed to detect all strongly-connected components in the constraint graph; however, HT and LCD both do a very good job of finding and collapsing cycles—for each benchmark they detect and collapse over 99% of the nodes collapsed by PKH. HCD by itself doesn't do as well, collapsing only 46–74% of the nodes collapsed by PKH. This deficiency is primarily responsible for HCD's greater memory consumption, though it is offset somewhat by the fact that HCD doesn't need to search the graph to collapse nodes.

Nodes Searched. HCD is, of course, the most efficient algorithm in terms of searching the constraint graph, since it doesn't search at all. HT is the next most efficient algorithm, because it only searches the subset of the graph necessary for resolving indirect constraints. PKH searches 2.6× as many nodes as HT, as it periodically searches the entire graph for cycles. LCD is the least efficient, searching 8× as many nodes as HT.

Propagations. LCD has the fewest propagations, showing that its greater effort at searching for cycles pays off by finding those cycles earlier than HT or PKH. HT has 1.8× as many propagations, and PKH has 2.2× as many. Since they both find as many cycles as LCD (as shown by the number of nodes collapsed), this difference is due to the relative amount of time it takes for each of the algorithms to find cycles. HCD has the most propagations, 5.2× as many as LCD. HCD finds cycles as soon as they are formed, so it finds them much faster than LCD does, but as shown above, it finds substantially fewer cycles than the other algorithms.

[4] It is difficult to find statistics to directly explain BLQ's performance relative to HT, PKH, LCD, and HCD. It doesn't use cycle detection, so it adds orders of magnitude more edges to the constraint graph—but propagation of points-to information is done simultaneously across all the edges using BDD operations, and the performance of the algorithm is due more to how well the BDDs compress the constraint and points-to graphs than anything else.

Effects of HCD. The main benefit of combining HCD with the other algorithms is that it helps these algorithms find cycles much sooner than they would on their own. While it does little to increase the number of nodes collapsed or decrease the number of nodes searched, it greatly decreases the number of propagations, because cycles are collapsed before the points-to information has a chance to propagate around the cycles. The addition of HCD decreases the number of propagations by 10× for HT and by 7.4× for both PKH and LCD.

Discussion Despite its lazy nature, LCD searches more nodes than either HT or PKH, and it propagates less points-to information than either as well. It appears that being more aggressive pays off, which naturally leads to the question: could we do better by being even more aggressive? However, past experience has shown that we must carefully balance the work we do—too much aggression can lead to overhead that overwhelms any benefits it may provide. This point is shown in both Faehndrich *et al.*'s algorithm [9] and Pearce *et al.*'s original algorithm [21]. Both of these algorithms are very aggressive in seeking out cycles, and both are an order of magnitude slower than any of the algorithms evaluated in this paper.

5.4 Representing Points-to Sets

Table 4 shows that the memory consumption of all the algorithms that use sparse bitmaps is extremely high. Profiling reveals that the majority of this memory usage comes from the bit-map representation of points-to sets. BLQ, on the other hand, uses relatively little memory even for the largest benchmarks, due to its use of BDDs. It is thus natural to wonder how the other algorithms would compare—in terms of both analysis time and memory consumption—if they were to instead use BDDs to represent points-to sets. Unlike BLQ, which stores the entire points-to solution in a single BDD, we store each variable's points-to set in its own BDD. For example, if $a \rightarrow \{b, c\}$ and $d \rightarrow \{c, e\}$, BLQ would use a single BDD to represent the set of tuples $\{(a, b), (a, c), (d, c), (d, e)\}$. Instead, we use a BDD to represent a's points-to set, $\{b, c\}$, and we use a separate BDD to represent d's points-to set, $\{c, e\}$. With this design, BDDs can be substituted for sparse bitmaps with minimal changes to the code.

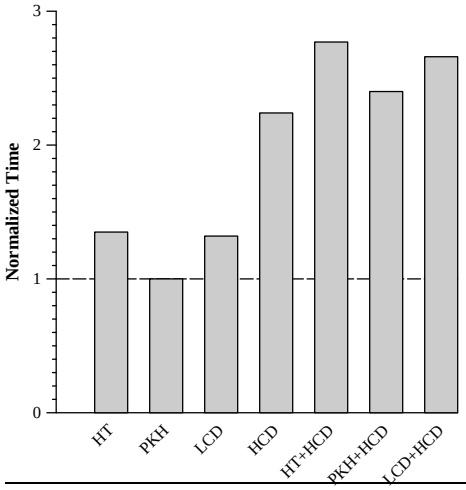


Figure 9. Performances of the BDD-based implementations normalized by their bitmap-based counterparts, averaged over all the benchmarks.

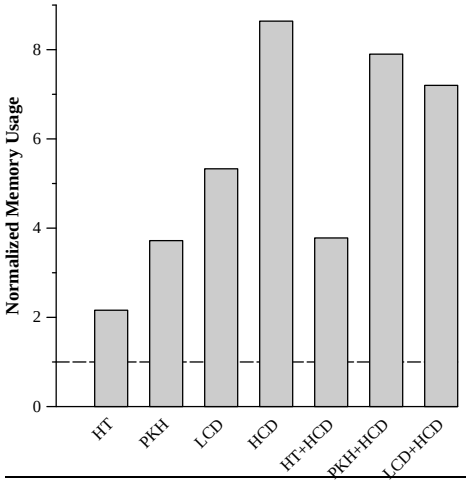


Figure 10. Memory consumption of the bitmap-based implementations normalized by their BDD-based counterparts, averaged over all the benchmarks.

Tables 5 and 6 show the performance and memory consumption of the modified algorithms. Figure 9 graphically shows the performance cost of the modified algorithms by normalizing them by their bitmap-based counterparts, and Figure 10 shows the memory savings by normalizing the bitmap-based algorithms by their BDD-based counterparts. As with BLQ, we allocate an initial pool of memory for the BDDs that is independent of the benchmark size, which is why memory consumption actually increases for the smallest benchmark, Emacs, and never goes lower than 33.1MB for any benchmark.

On average, the use of BDDs increases running time by $2\times$ while it decreases memory usage by $5.5\times$. Most of the extra time comes from a single function, *bdd_allsat*, which is used to extract all the elements of a set contained in a given BDD. This function is used when iterating through a variable's points-to set while adding new edges according to the complex constraints. However, both PKH and HCD are actually faster on all benchmarks except for Emacs (Figure 9 shows that they are slower on average, but this is solely because of Emacs). These are the two algorithms that propagate the most points-to information across constraint edges. BDDs make this operation much faster than using sparse bitmaps, and this advantage makes up for the extra time taken by bdd_allsat.

When BDDs are used, HCD is less effective in improving performance than it was when using bitmaps because HCD decreases the number of propagations required, but using BDDs already makes propagation a fairly cheap operation. However, with BDDs, HCD's effect on memory consumption is much more noticeable, since the constraint graph represents a much larger proportion of the memory usage.

6. Conclusion

We have significantly improved upon the current state-of-the-art in inclusion-based pointer analysis by introducing two novel techniques: *Lazy Cycle Detection* (LCD) and *Hybrid Cycle Detection* (HCD). As their names suggest, both techniques improve the efficiency and effectiveness of online cycle detection, which is critical to the scalability of all inclusion-based pointer analyses. Lazy Cycle Detection detects cycles based on their effects on points-to sets, piggybacking on top of the transitive closure computation that is inherent to this type of analysis. Its lazy nature yields a highly efficient algorithm. Hybrid Cycle Detection takes a different approach, paying a tiny up-front cost to perform an offline analysis that allows the subsequent online analysis to detect cycles without ever having to traverse the constraint graph. Hybrid Cycle Detection can be used to enhance other algorithms for inclusion-based pointer analysis, significantly improving their performance. Our results show that the combination of LCD and HCD is on average the most efficient of all the algorithms we studied. On our suite of six large open source C benchmarks, which range in size from 169K to 2.17M lines of code, the LCD+HCD algorithm is an average of $3.2\times$ faster than the Heintze and Tardieu algorithm, $6.4\times$ faster than the Pearce *et al.* algorithm, and $20.6\times$ faster than the Berndl *et al.* algorithm.

We have also investigated the use of different data structures to represent points-to sets, examining the impact on both performance and memory consumption. In particular, we have compared the sparse-bitmap implementation used in the GCC open-source compiler with a BDD-based implementation, and we have found that the BDD implementation is on average $2\times$ slower but uses $5.5\times$ less memory.

Many program analyses that require pointer information sacrifice precision in the pointer analysis for the sake of reasonable performance. This performance is the attraction of analyses such as Steensgaard's near-linear-time analysis [25] and Das' One-Level Flow analysis [7]. However, the precision of subsequent program analysis is often limited by the precision of the pointer information used [24], so it behooves an analysis to use the most precise pointer information that it can reasonably acquire. Our work has made inclusion-based pointer analysis a reasonable choice even for applications with millions of lines of code.

Acknowledgments

We thank Brandon Streiff and Luke Robison for their help in conducting experiments, Dan Berlin for his help with the GCC compiler internals, and Sam Guyer, E Lewis, Kathryn McKinley, and the anonymous reviewers for their helpful comments on early drafts of this paper. This work was supported by NSF grants ACI-0313263 and CNS-0509354 and by an IBM Faculty Partnership Award.

References

[1] Aesop. *The Ant and the Grasshopper,* from *Aesop's Fables*. Greece, 6^{th} century BC.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIEM, University of Copenhagen, May 1994.

[3] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a c pointer analysis. In *27th international Conference on Software Engineering (ICSE)*, pages 332–341, New York, NY, USA, 2005. ACM Press.

[4] Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Programming Language Design and Implementation (PLDI)*, 2003.

[5] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[6] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Principles of Programming Languages (POPL)*, pages 232–245, New York, NY, USA, 1993. ACM Press.

[7] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Programming Language Design and Implementation (PLDI)*, 1994.

[9] Manuel Faehndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1998. ACM Press.

[10] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2):83–114, 2005.

[11] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *Programming Language Design and Implementation (PLDI)*, 2001.

[12] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[13] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[14] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Symposium on Principles of Programming Languages (POPL)*, pages 93–103, New York, NY, USA, 1991. ACM Press.

[15] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Programming Language Design and Implementation (PLDI)*, 1992.

[16] J. Lind-Nielson. BuDDy, a binary decision package. http://www.itu.dk/research/buddy/.

[17] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Computational Complexity*, pages 213–228, 2002.

[18] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[19] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strong components in a directed graph. Technical Report TKO-B94, Helsinki University of Technology, Laboratory of Information Processing Science, 1995.

[20] Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *International Symposium on Static Analysis*, pages 165–180, 2004.

[21] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *3rd International IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[22] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient Field-Sensitive Pointer Analysis for C. In *ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2004.

[23] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. *ACM SIGPLAN Notices*, 35(5):47–56, 2000.

[24] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. *Lecture Notes in Computer Science*, 1302:16–??, 1997.

[25] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[26] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.

[27] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *15th International Conference on Compiler Construction (CC)*, pages 17–31, 2006.

[28] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis. In *Programming Language Design and Implementation (PLDI)*, 2004.

[29] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Programming Language Design and Implementation (PLDI)*, 1995.

[30] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Programming Language Design and Implementation (PLDI)*, pages 145–157, New York, NY, USA, 2004. ACM Press.