Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis

Ben Hardekopf and Calvin Lin

The University of Texas at Austin, Austin TX 78712, USA
{benh,lin}@cs.utexas.edu

Abstract. Pointer information is a prerequisite for most program analyses, and inclusion-based, *i.e.* Andersen-style, pointer analysis is widely used to compute such information. However, current inclusion-based analyses can have prohibitive costs in time and space, especially for programs with millions of lines of code. We present a suite of offline optimizations that exploit pointer and location equivalence to shrink the input to the subsequent pointer analysis without affecting precision, dramatically reducing both analysis time and memory consumption. Using a suite of six open-source C programs ranging in size from 169K to 2.17M LOC, we demonstrate that our techniques on average improve analysis time by 1.3–2.7× and reduce memory consumption by 3.2–6.9× over the best current techniques.

1 Introduction

Most program analyses require pointer information, from traditional compiler optimizations to software verification, security analysis, and program understanding. Many of these analyses are interprocedural and require a highly scalable whole-program pointer analysis to compute pointer information. The precision of the computed information can have a profound impact on the usefulness of the subsequent program analysis. Inclusion-based, *i.e.* Andersen-style, pointer analysis is widely-used because of its relative precision and potential for scalability. Inclusion-based analysis scales to millions of lines of code, but memory consumption is prohibitively high [6]. Memory consumption can be greatly reduced by using BDDs to represent points-to sets, but this significantly increases analysis time [6]. Our goal is to break this trade-off by reducing both memory consumption and analysis time for inclusion-based pointer analysis, without affecting the precision of the results.

Inclusion-based analysis is the most precise flow- and context-insensitive pointer analysis. It extracts inclusion constraints from the program code to approximate points-to relations between variables, representing the constraints using a *constraint graph*, with nodes to represent each program variable and edges to represent the constraints between variables. Indirect constraints—those that involve pointer dereferences—can't be directly represented in the graph, since points-to information isn't available until after the analysis has completed. The analysis satisfies the constraints by computing the dynamic transitive closure of the graph; as new points-to information becomes available, new edges are added to the graph to represent the indirect constraints. The transitive closure of the final graph yields the points-to solution.

Inclusion-based analysis has a complexity of $O(n^3)$ time and $O(n^2)$ space, where n is the number of variables; the key to scaling the analysis is to reduce the input size—*i.e.* make n smaller—while ensuring that precision is not affected. This goal is accomplished by detecting equivalences among the program variables and collapsing together equivalent variables. Existing algorithms only recognize a single type of equivalence, which we call *pointer equivalence*: program variables are pointer equivalent iff their points-to sets are identical. There are several existing methods for exploiting pointer equivalence. The primary method is *online cycle detection* [5–7, 10, 11]. Rountev *et al.* [12] introduce another method called *Offline Variable Substitution* (OVS). An offline analysis is a static analysis performed prior to the actual pointer analysis; in this case, OVS identifies and collapses a subset of the pointer equivalent variables before feeding the constraints to the pointer analysis.

In this paper, we introduce a suite of new offline optimizations for inclusion-based pointer analysis that go far beyond OVS in finding pointer equivalences. We also introduce and exploit a second notion of equivalence called *location equivalence*: program variables are location equivalent iff they always belong to the same points-to sets, *i.e.* any points-to set containing one must also contain the other. Our new optimizations are the first to exploit location equivalence to reduce the size of the variables' points-to sets without affecting precision. Together, these offline optimizations dramatically reduce both the time and memory consumption of subsequent inclusion-based pointer analysis. This paper presents the following major results:

- Using three different inclusion-based pointer analysis algorithms [7, 10, 6], we demonstrate that our optimizations on average reduce analysis time by 1.3–2.7× and reduce memory consumption by 3.2–6.9×.
- We experiment with two different data structures to represent points-to sets: (1) sparse bitmaps, as currently used in the GCC compiler, and (2) a BDD-based representation. While past work has found that the bitmap representation is 2× faster but uses 5.5× more memory than the BDD representation [6], we find that, due to our offline optimizations, the bitmap representation is on average 1.3× faster and uses 1.7× *less* memory than the BDD representation.

This paper makes the following conceptual contributions:

- We present Hash-based Value Numbering (HVN), an offline optimization which adapts a classic compiler optimization [3] to find and exploit pointer equivalences.
- We present HRU (**H**VN with de**R**eference and **U**nion), an extension of HVN that finds additional pointer equivalences by interpreting both union and dereference operators in the constraints.

– We present LE (**L**ocation **E**quivalence), an offline optimization that finds and exploits location equivalences to reduce variables' points-to set sizes without affecting precision.

2 Related Work

Andersen introduces inclusion-based pointer analysis in his Ph.D. thesis [1], where he formulates the problem in terms of type theory. Andersen's algorithm solves the inclusion constraints in a fairly naive manner by repeatedly iterating through a constraint vector.

The first use of pointer equivalence to optimize inclusion-based analysis comes from Faehndrich *et al.* [5], who represent constraints using a graph and then derive points-to information by computing the dynamic transitive closure of that graph. The key optimization is a method for partial online cycle detection.

Later algorithms expand on Faehndrich *et al.*'s work by making online cycle detection more complete and efficient [6, 7, 10, 11]. In particular, Heintze and Tardieu [7] describe a field-based analysis, which is capable of analyzing over 1 million lines of C code in a matter of seconds. Field-based analysis does not always meet the needs of the client analysis, particularly since field-based analysis is unsound for C; a field-insensitive version of their algorithm is significantly slower [6].

Rountev *et al.* [12] introduce Offline Variable Substitution (OVS), a linear-time static analysis whose aim is to find and collapse pointer-equivalent variables. Of all the related work, OVS is the most similar to our optimizations and serves as the baseline for our experiments in this paper.

Both pointer and location equivalence have been used in other types of pointer analyses, although they have not been explicitly identified as such; Steensgaard's analysis [14], Das' One-Level Flow [4], and the Shapiro-Horwitz family of analyses [13] all sacrifice precision to gain extra performance by inducing artificial pointer and location equivalences. By contrast, we detect and exploit actual equivalences between variables without losing precision.

Location equivalence has also been used by Liang and Harrold to optimize dataflow analyses [8], but only post-pointer analysis. We give the first method for soundly exploiting location equivalence to optimize the pointer analysis itself.

3 Pointer Equivalence

Let $\mathcal{V}$ be the set of all program variables; for $v \in \mathcal{V} : pts(v) \subseteq \mathcal{V}$ is v's points-to set, and $pe(v) \in \mathcal{N}$ is the *pointer equivalence label* of v, where $\mathcal{N}$ is the set of natural numbers. Variables x and y are pointer equivalent iff $pts(x) = pts(y)$. Our goal is to assign pointer equivalence labels such that $pe(x) = pe(y)$ implies that x and y are pointer equivalent. Pointer equivalent variables can safely be collapsed together in the constraint graph to reduce both the number of nodes and edges in the graph. The benefits are two-fold: (1) there are fewer points-to sets to maintain; and (2) there are fewer propagations of points-to information along the edges of the constraint graph.

The analysis generates inclusion constraints using a linear pass through the program code; control-flow information is discarded and only variable assignments are considered. Function calls and returns are treated as gotos and are broken down into sets of parameter assignments. Table 1 illustrates the types of constraints and defines their meaning.

Table 1. Inclusion Constraint Types.

Program Code	**Constraint**	**Meaning**
$a = \&b$	$a \supseteq \{b\}$	$b \in pts(a)$
$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$
$a = *b$	$a \supseteq *b$	$\forall v \in pts(b) : pts(a) \supseteq pts(v)$
$*a = b$	$*a \supseteq b$	$\forall v \in pts(a) : pts(v) \supseteq pts(b)$

Our optimizations use these constraints to create an *offline constraint graph*,[1] with VAR nodes to represent each variable, REF nodes to represent each dereferenced variable, and ADR nodes to represent each address-taken variable. A REF node $*a$ stands for the unknown points-to set of variable a, while ADR node $\&a$ stands for the address of variable a. Edges represent the inclusion relationships: $a \supseteq \{b\}$ yields edge $\&b \rightarrow a$; $a \supseteq b$ yields $b \rightarrow a$; $a \supseteq *b$ yields $*b \rightarrow a$; and $*a \supseteq b$ yields $b \rightarrow *a$.

Before describing the optimizations, we first explain the concepts of *direct* and *indirect* nodes [12]. Direct nodes have all of their points-to relations explicitly represented in the constraint graph: for direct node x and the set of nodes $\mathcal{S} = \{y : y \rightarrow x\}$, $pts(x) = \bigcup_{y \in \mathcal{S}} pts(y)$. Indirect nodes are those that may have points-to relations that are not represented in the constraint graph. All REF nodes are indirect because the unknown variables that they represent may have their own points-to relations. VAR nodes are indirect if they (1) have had their address taken, which means that they can be referenced indirectly via a REF node; (2) are the formal parameter of an indirect function call; or (3) are assigned the return value of an indirect function call. All other VAR nodes are direct.

All indirect nodes are conservatively treated as possible sources of points-to information, and therefore each is given a distinct pointer equivalence label at the beginning of the algorithm. ADR nodes are definite sources of points-to information, and they are also given distinct labels. For convenience, we will use the term 'indirect node' to refer to both ADR nodes and true indirect nodes because they will be treated equivalently by our optimizations.

Figure 1 shows a set of constraints and the corresponding offline constraint graph. In Figure 1 all the REF and ADR nodes are marked indirect, as well as VAR nodes a and d, because they have their address taken. Because a and d can

[1] The offline constraint graph is akin to the *subset graph* described by Rountev *et al.* [12].

now be accessed indirectly through pointer dereference, we can no longer assume that they only acquire points-to information via nodes h and i, respectively.

$b \supseteq \{a\}$	$a \supseteq h$	$h \supseteq *b$
$b \supseteq \{d\}$	$c \supseteq b$	$i \supseteq *e$
$c \supseteq \{a\}$	$d \supseteq i$	$k \supseteq *j$
$e \supseteq \{a\}$	$e \supseteq f$	
$e \supseteq \{d\}$	$f \supseteq e$	
	$g \supseteq f$	

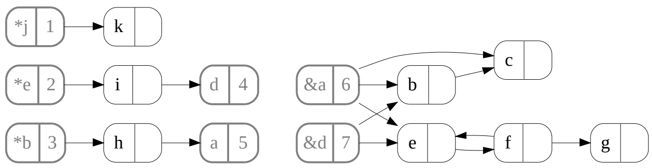

Fig. 1. Example offline constraint graph. Indirect nodes are grey and have already been given their pointer equivalence labels. Direct nodes are black and have not been given pointer equivalence labels.

3.1 Hash-based Value Numbering (HVN)

The goal of HVN is to give each direct node a pointer equivalence label such that two nodes share the same label only if they are pointer equivalent. HVN can also identify non-pointers—variables that are guaranteed to never point to anything. Non-pointers can arise in languages with weak types systems, such as C: the constraint generator can't rely on the variables' type declarations to determine whether a variable is a pointer or not, so it conservatively assumes that everything is a pointer. HVN can eliminate many of these superfluous variables; they are identified by assigning a pointer equivalence label of 0. The algorithm proceeds as follows:

1. Find and collapse strongly-connected components (SCCs) in the offline constraint graph. If any node in the SCC is indirect, the entire SCC is indirect. In Figure 1, e and f are collapsed into a single (direct) node.
2. Proceeding in topological order, for each direct node x let $\mathcal{L}$ be the set of positive incoming pointer equivalence labels, *i.e.* $\mathcal{L} = \{pe(y) : y \to x \wedge pe(y) \neq 0\}$. There are three cases:
   (a) $\mathcal{L}$ is empty. Then x is a non-pointer and $pe(x) = 0$.
   Explanation: in order for x to potentially be a pointer, there must exist a path to x either from an ADR node or some indirect node. If there is no such path, then x must be a non-pointer.
   (b) $\mathcal{L}$ is a singleton, with $p \in \mathcal{L}$. Then $pe(x) = p$.
   Explanation: if every points-to set coming in to x is identical, then x's points-to set, being the union of all the incoming points-to sets, must be identical to the incoming sets.
   (c) $\mathcal{L}$ contains multiple labels. The algorithm looks up $\mathcal{L}$ in a hashtable to see if it has encountered the set before. If so, it assigns $pe(x)$ the same label; otherwise it creates a new label, stores it in the hashtable, and assigns it to $pe(x)$.

Explanation: x's points-to set is the union of all the incoming points-to sets; x must be equivalent to any node whose points-to set results from unioning the same incoming points-to sets.

Following these steps for Figure 1, the final assignment of pointer equivalence labels for the direct nodes is shown in Figure 2. Once we have assigned pointer equivalence labels, we merge nodes with identical labels and eliminate all edges incident to nodes labeled 0.

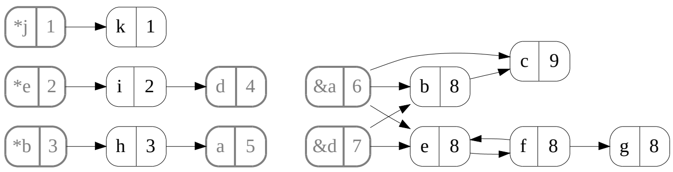

Fig. 2. The assignment of pointer equivalence labels after HVN.

Complexity. The complexity of HVN is linear in the size of the graph. Using Tarjan's algorithm for detecting SCCs [15], step 1 is linear. The algorithm then visits each direct node exactly once and examines its incoming edges. This step is also linear.

Comparison to OVS. HVN is similar to Rountev *et al.*'s [12] OVS optimization. The main difference lies in our insight that labeling the condensed offline constraint graph is essentially equivalent to performing value-numbering on a block of straight-line code, and therefore we can adapt the classic compiler optimization of hash-based value numbering for this purpose. The advantage lies in step 2c: in this case OVS would give the direct node a new label without checking to see if any other direct nodes have a similar set of incoming labels, potentially missing a pointer equivalence. In the example, OVS would not discover that b and e are equivalent and would give them different labels.

3.2 Extending HVN

HVN does not find all pointer equivalences that can be detected prior to pointer analysis because it does not interpret the *union* and *dereference* operators. Recall that the union operator is implicit in the offline constraint graph: for direct node x with incoming edges from nodes y and z, $pts(x) = pts(y) \cup pts(z)$. By interpreting these operators, we can increase the number of pointer equivalences detected, at the cost of additional time and space.

HR algorithm. By interpreting the dereference operator, we can relate a VAR node v to its corresponding REF node $*v$. There are two relations of interest:

1. $\forall x, y \in \mathcal{V} : pe(x) = pe(y) \Rightarrow pe(*x) = pe(*y)$.
2. $\forall x \in \mathcal{V} : pe(x) = 0 \Rightarrow pe(*x) = 0$.

The first relation states that if variables x and y are pointer-equivalent, then so are $*x$ and $*y$. If x and y are pointer-equivalent, then by definition $*x$ and $*y$ will be identical. Whereas HVN would give them unique pointer equivalence labels, we can now assign them the same label. By doing so, we may find additional pointer equivalences that had previously been hidden by the different labels.

The second relation states that if variable x is a non-pointer, then $*x$ is also a non-pointer. It may seem odd to have a constraint that dereferences a non-pointer, but this can happen when code that initializes pointer values is linked but never called, for example with library code. Exposing this relationship can help identify additional non-pointers and pointer equivalences.

Figure 3 provides an example. HVN assigns b and e identical labels; the first relation above tells us we can assign $*b$ and $*e$ identical labels, which exposes the fact that i and h are equivalent to each other, which HVN missed. Also, variable j is not mentioned in the constraints, and therefore the VAR node j isn't shown in the graph, and it is assigned a pointer equivalence label of 0. The second relation above tells us that because $pe(j) = 0$, $pe(*j)$ should also be 0; therefore both $*j$ and k are non-pointers and can be eliminated.

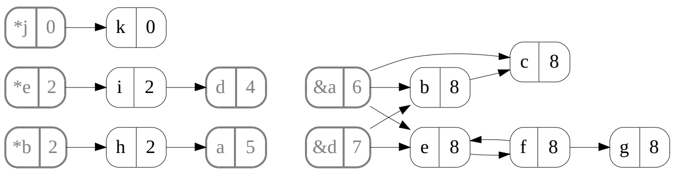

Fig. 3. The assignment of pointer equivalence labels after HR and HU.

The simplest method for interpreting the dereference operator is to iteratively apply HVN to its own output until it converges to a fixed point. Each iteration collapses equivalent variables and eliminates non-pointers, fulfilling the two relations we describe. This method adds an additional factor of $O(n)$ to the complexity of the algorithm, since in the worst case it eliminates a single variable in each iteration until there is only one variable left. The complexity of HR is therefore $O(n^2)$, but in practice we observe that this method generally exhibits linear behavior.

HU algorithm. By interpreting the union operator, we can more precisely track the relations among points-to sets. Figure 3 gives an example in VAR node c. Two different pointer equivalence labels reach c, one from $\&a$ and one from b. HVN therefore gives c a new pointer equivalence label. However, $pts(b) \supseteq pts(\&a)$, so when they are unioned together the result is simply $pts(b)$. By keeping track of this fact, we can assign c the same pointer equivalence label as b.

Let f_n be a fresh number unique to n; the algorithm will use these fresh values to represent unknown points-to information. The algorithm operates on the condensed offline constraint graph as follows:

1. Initialize points-to sets for each node. $\forall v \in \mathcal{V} : pts(\&v) = \{v\}$; $pts(*v) = \{f_{*v}\}$; if v is direct then $pts(v) = \emptyset$, else $pts(v) = \{f_v\}$.
2. In topological order: for each node x, let $\mathcal{S} = \{y : y \rightarrow x\} \cup \{x\}$. Then $pts(x) = \bigcup_{y \in \mathcal{S}} pts(y)$.
3. Assign labels s.t. $\forall x, y \in V : pts(x) = pts(y) \Leftrightarrow pe(x) = pe(y)$.

Since this algorithm is effectively computing the transitive closure of the constraint graph, it has a complexity of $O(n^3)$. While this is the same complexity as the pointer analysis itself, HU is significantly faster because, unlike the pointer analysis, we do not add additional edges to the offline constraint graph, making the offline graph much smaller than the graph used by the pointer analysis.

Putting It Together: HRU. The HRU algorithm combines the HR and HU algorithms to interpret both the dereference and union operators. HRU modifies HR to iteratively apply the HU algorithm to its own output until it converges to a fixed point. Since the HU algorithm is $O(n^3)$ and HR adds a factor of $O(n)$, HRU has a complexity of $O(n^4)$. As with HR this worst-case complexity is not observed in practice; however it is advisable to first apply HVN to the original constraints, then apply HRU to the resulting set of constraints. HVN significantly decreases the size of the offline constraint graph, which decreases both the time and memory consumption of HRU.

4 Location Equivalence

Let $\mathcal{V}$ be the set of all program variables; for $v \in \mathcal{V} : pts(v) \subseteq \mathcal{V}$ is v's points-to set, and $le(v) \in \mathcal{N}$ is the *location equivalence label* of v, where $\mathcal{N}$ is the set of natural numbers. Variables x and y are location equivalent iff $\forall z \in \mathcal{V} : x \in pts(z) \Leftrightarrow y \in pts(z)$. Our goal is to assign location equivalence labels such that $le(x) = le(y)$ implies that x and y are location equivalent. Location equivalent variables can safely be collapsed together in all points-to sets, providing two benefits: (1) the points-to sets consume less memory; and (2) since the points-to sets are smaller, points-to information is propagated more efficiently across the edges of the constraint graph.

Without any pointer information it is impossible to compute all location equivalences. However, since points-to sets are never split during the pointer analysis, any variables that are location equivalent at the beginning are guaranteed to be location equivalent at the end. We can therefore safely compute a subset of the equivalences prior to the pointer analysis. We use the same offline constraint graph as we use to find pointer equivalence, but we will be labeling ADR nodes instead of direct nodes. The algorithm assigns each ADR node a label based on its outgoing edges such that two ADR nodes have the same label iff they have the same set of outgoing edges. In other words, ADR nodes $\&a$ and $\&b$ are assigned the same label iff, in the constraints, $\forall z \in \mathcal{V} : z \supseteq \{a\} \Leftrightarrow z \supseteq \{b\}$. In Figure 1, the ADR nodes $\&a$ and $\&d$ would be assigned the same location equivalence label.

While location and pointer equivalences can be computed independently, it is more precise to compute location equivalence *after* we have computed pointer equivalence. We modify the criterion to require that ADR nodes $\&a$ and $\&b$ are assigned the same label iff $\forall y, z \in V, (y \supseteq \{a\} \wedge z \supseteq \{b\}) \Rightarrow pe(y) = pe(z)$. In other words, we don't require that the two ADR nodes have the same set of outgoing edges, but rather that the nodes incident to the ADR nodes have the same set of pointer equivalence labels.

Once the algorithm has assigned location equivalence labels, it merges all ADR nodes that have identical labels. These merged ADR nodes are each given a fresh name. Points-to set elements will come from this new set of fresh names rather than from the original names of the merged ADR nodes, thereby saving space, since a single fresh name corresponds to multiple ADR nodes. However, we must make a simple change to the subsequent pointer analysis to accommodate this new naming scheme. When adding new edges from indirect constraints, the pointer analysis must translate from the fresh names in the points-to sets to the original names corresponding to the VAR nodes in the constraint graph. To facilitate this translation we create a one-to-many mapping between the fresh names and the original ADR nodes that were merged together. In Figure 1, since ADR nodes $\&a$ and $\&d$ are given the same location equivalence label, they will be merged together and assigned a fresh name such as $\&l$. Any points-to sets that formerly would have contained a and d will instead contain l; when adding additional edges from an indirect constraint that references l, the pointer analysis will translate l back to a and d to correctly place the edges in the online constraint graph.

Complexity. LE is linear in the size of the constraint graph. The algorithm scans through the constraints, and for each constraint $a \supseteq \{b\}$ it inserts $pe(a)$ into ADR node $\&b$'s set of pointer equivalence labels. This step is linear in the number of constraints (*i.e.* graph edges). It then visits each ADR node, and it uses a hash table to map from that node's set of pointer equivalence labels to a single location equivalence label. This step is also linear.

5 Evaluation

5.1 Methodology

Using a suite of six open-source C programs, which range in size from 169K to 2.17M LOC, we compare the analysis times and memory consumption of OVS, HVN, HRU, and HRU+LE (HRU coupled with LE). We then use three different state-of-the-art inclusion-based pointer analyses—Pearce *et al.* [10] (PKH), Heintze and Tardieu [7] (HT), and Hardekopf and Lin [6] (HL)—to compare the optimizations' effects on the pointer analyses' analysis time and memory consumption. These pointer analyses are all field-insensitive and implemented in a common framework, re-using as much code as possible to provide a fair comparison. The source code is available from the authors upon request.

The offline optimizations and the pointer analyses are written in C++ and handle all aspects of the C language except for varargs. We use sparse bitmaps taken from GCC 4.1.1 to represent the constraint graph and points-to sets. The constraint generator is separate from the constraint solvers; we generate constraints from the benchmarks using the CIL C front-end [9], ignoring any assignments involving types too small to hold a pointer. External library calls are summarized using hand-crafted function stubs.

The benchmarks for our experiments are described in Table 2. We run the experiments on an Intel Core Duo 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 with 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. Times include everything from reading the constraint file from disk to computing the final solution.

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), a description of the benchmark, and the number of constraints generated by the CIL front-end.

Name	**Description**	**LOC**	**Constraints**
Emacs-21.4a	text editor	169K	83,213
Ghostscript-8.15	postscript viewer	242K	169,312
Gimp-2.2.8	image manipulation	554K	411,783
Insight-6.5	graphical debugger	603K	243,404
Wine-0.9.21	windows emulator	1,338K	713,065
Linux-2.4.26	linux kernel	2,172K	574,788

5.2 Cost of Optimizations

Tables 3 and 4 show the analysis time and memory consumption, respectively, of the offline optimizations on the six benchmarks. OVS and HVN have roughly the same times, with HVN using 1.17× more memory than OVS. On average, HRU and HRU+LE are 3.1× slower and 3.4× slower than OVS, respectively. Both HRU and HRU+LE have the same memory consumption as HVN. As stated earlier, these algorithms are run on the output of HVN in order to improve analysis time and conserve memory; their times are the sum of their running time and the HVN running time, while their memory consumption is the maximum of their memory usage and the HVN memory usage. In all cases, the HVN memory usage is greater.

Table 3. Offline analysis times (sec).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	0.29	0.60	1.74	0.96	3.57	2.34
HVN	0.29	0.61	1.66	0.95	3.39	2.36
HRU	0.49	2.29	4.31	4.28	9.46	7.70
HRU+LE	0.53	2.54	4.75	4.64	10.41	8.47

Table 4. Offline analysis memory (MB).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	13.1	28.1	61.1	39.1	110.4	96.2
HVN	14.8	32.5	71.5	44.7	134.8	114.8
HRU	14.8	32.5	71.5	44.7	134.8	114.8
HRU+LE	14.8	32.5	71.5	44.7	134.8	114.8

Figure 4 shows the effect of each optimization on the number of constraints for each benchmark. On average OVS reduces the number of constraints by 63.4%, HVN by 69.4%, HRU by 77.4%, and HRU+LE by 79.9%. HRU+LE, our most aggressive optimization, takes 3.4× longer than OVS, while it only reduces the number of constraints by an additional 16.5%. However, inclusion-based analysis is $O(n^3)$ time and $O(n^2)$ space, so even a relatively small reduction in the input size can have a significant effect, as we'll see in the next section.

5.3 Benefit of Optimizations

Tables 5–10 give the analysis times and memory consumption for three pointer analyses—PKH, HT, and HL—as run on the results of each offline optimization; OOM indicates the analysis ran out of memory. The data is summarized in Figure 5, which gives the average performance and memory improvement for the three pointer analyses for each offline algorithm as compared to OVS. The offline analysis times are added to the pointer analysis times to make the overall analysis time comparison.

Analysis Time. For all three pointer analyses, HVN only moderately improves analysis time over OVS, by 1.03–1.18×. HRU has a greater effect despite its much higher offline analysis times; it improves analysis time by 1.28–1.88×. HRU+LE has the greatest effect; it improves analysis time by 1.28–2.68×. An

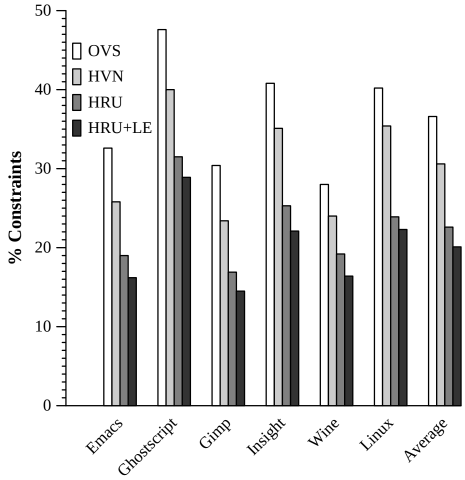

Fig. 4. Percent of the original number of constraints that is generated by each optimization.

Table 5. Online analysis times for the PKH algorithm (sec).

	Emacs	**Ghostscript**	**Gimp**	**Insight**	**Wine**	**Linux**
OVS	1.99	19.15	99.22	121.53	1,980.04	1,202.78
HVN	1.60	17.08	87.03	111.81	1,793.17	1,126.90
HRU	0.74	13.31	38.54	57.94	1,072.18	598.01
HRU+LE	0.74	9.50	21.03	33.72	731.49	410.23

Table 6. Online analysis memory for the PKH algorithm (MB).

	Emacs	**Ghostscript**	**Gimp**	**Insight**	**Wine**	**Linux**
OVS	23.1	102.7	418.1	251.4	1,779.7	1,016.5
HVN	17.7	83.9	269.5	194.8	1,448.5	840.8
HRU	12.8	68.0	171.6	165.4	1,193.7	590.4
HRU+LE	6.9	23.8	56.1	58.6	295.9	212.4

Table 7. Online analysis times for the HT algorithm (sec).

	Emacs	**Ghostscript**	**Gimp**	**Insight**	**Wine**	**Linux**
OVS	1.63	13.58	64.45	46.32	**OOM**	410.52
HVN	1.84	12.84	59.68	42.70	**OOM**	393.00
HRU	0.70	9.95	37.27	37.03	1,087.84	464.51
HRU+LE	0.54	8.82	18.71	23.35	656.65	332.36

Table 8. Online analysis memory for the HT algorithm (MB).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	22.5	97.2	359.7	266.9	**OOM**	1,006.8
HVN	17.7	85.0	279.0	231.5	**OOM**	901.3
HRU	10.8	70.3	205.3	156.7	1,533.0	700.7
HRU+LE	6.4	34.9	86.0	69.4	820.9	372.2

Table 9. Online analysis times for the HL algorithm (sec).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	1.07	9.15	17.55	20.45	534.81	103.37
HVN	0.68	8.14	13.69	17.23	525.31	91.76
HRU	0.32	7.25	10.04	12.70	457.49	75.21
HRU+LE	0.51	6.67	8.39	13.71	345.56	79.99

Table 10. Online analysis memory for the HL algorithm (MB).

	Emacs	Ghostscript	Gimp	Insight	Wine	Linux
OVS	21.0	93.9	415.4	239.7	1,746.3	987.8
HVN	13.9	73.5	263.9	183.7	1,463.5	807.9
HRU	9.2	63.3	170.7	121.9	1,185.3	566.6
HRU+LE	4.5	22.2	33.4	27.6	333.1	162.6

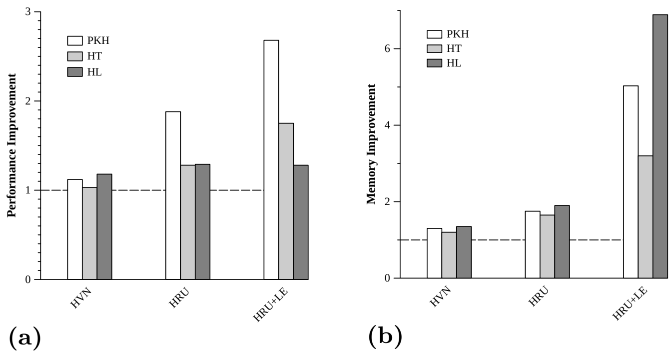

Fig. 5. (a) Average performance improvement over OVS; **(b)** Average memory improvement over OVS. For each graph, and for each offline optimization $X \in \{$HVN, HRU, HRU+LE$\}$, we compute $\frac{OVS_{time/memory}}{X_{time/memory}}$.

important factor in the analysis time of these algorithms is the number of times they propagate points-to information across constraint edges. PKH is the least efficient of the algorithms in this respect, propagating much more information than the other two; hence it benefits more from the offline optimizations. HL propagates the least amount of information and therefore benefits the least.

Memory. For all three pointer analyses HVN only moderately improves memory consumption over OVS, by 1.2–1.35×. All the algorithms benefit significantly from HRU, using 1.65–1.90× less memory than for OVS. HRU's greater reduction in constraints makes for a smaller constraint graph and fewer points-to sets. HRU+LE has an even greater effect: HT uses 3.2× less memory, PKH uses 5× less memory, and HL uses almost 7× less memory. HRU+LE doesn't further reduce the constraint graph or the number of points-to sets, but on average it cuts the average points-to set size in half.

Room for Improvement. Despite aggressive offline optimization in the form of HRU plus the efforts of online cycle detection, there are still a significant number of pointer equivalences that we do not detect in the final constraint graph. The number of actual pointer equivalence classes is much smaller than the number of detected equivalence classes, by almost 4× on average. In other words, we could conceivably shrink the online constraint graph by almost 4× if we could do a better job of finding pointer equivalences. This is an interesting area for future work. On the other hand, we do detect a significant fraction of the actual location equivalences—we detect 90% of the actual location equivalences in the five largest benchmarks, though for the smallest (Emacs) we only detect 41%. Thus there is not much room to improve on the LE optimization.

Bitmaps vs. BDDs. The data structure used to represent points-to sets for the pointer analysis can have a great effect on the analysis time and memory consumption of the analysis. Hardekopf and Lin [6] compare the use of sparse bitmaps versus BDDs to represent points-to sets and find that on average the BDD implementation is 2× slower but uses 5.5× less memory than the bitmap implementation. To make a similar comparison testing the effects of our optimizations, we implement two versions of each pointer analysis: one using sparse bitmaps to represent points-to sets, the other using BDDs for the same purpose. Unlike BDD-based pointer analyses [2, 16] which store 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 $v \to \{w, x\}$ and $y \to \{x, z\}$, the BDD-based analyses would have a single BDD that represents the set of tuples $\{(v, w), (v, x), (y, x), (y, z)\}$. Instead, we give v a BDD that represents the set $\{w, x\}$ and we give y a BDD that represents the set $\{w, z\}$. The two BDD representations take equivalent memory, but our representation is a simple modification that requires minimal changes to the existing code.

The results of our comparison are shown in Figure 6. We find that for HVN and HRU, the bitmap implementations on average are 1.4–1.5× faster than the

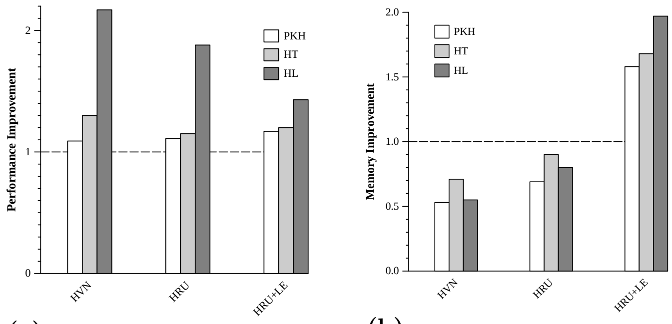

Fig. 6. (a) Average performance improvement over BDDs;**(b)** Average memory improvement over BDDs. Let BDD be the BDD implementation and BIT be the bitmap implementation; for each graph we compute $\frac{BDD_{time/memory}}{BIT_{time/memory}}$.

BDD implementations but use 3.5–4.4× more memory. However, for HRU+LE the bitmap implementations are on average 1.3× faster and use 1.7× *less* memory than the BDD implementations, because the LE optimization significantly shrinks the points-to sets of the variables.

6 Conclusion

In this paper we have shown that it is possible to reduce both the memory consumption and analysis time of inclusion-based pointer analysis without affecting precision. We have empirically shown that for three well-known inclusion-based analyses with highly tuned implementations, our offline optimizations improve average analysis time by 1.3–2.7× and reduce average memory consumption by 3.2–6.9×. For the fastest known inclusion-based analysis [6], the optimizations improve analysis time by 1.3× and reduce memory consumption by 6.9×. We have also found the somewhat surprising result that with our optimizations a sparse bitmap representation of points-to sets is both faster and requires less memory than a BDD representation.

In addition, we have provided a roadmap for further investigations into the optimization of inclusion-based analysis. Our optimization that exploits location equivalence comes close to the limit of what can be accomplished, but our other optimizations identify only a small fraction of the pointer equivalences. Thus, the exploration of new methods for finding and exploiting pointer equivalences should be a fruitful area for future work.

Acknowledgments. We thank Brandon Streiff and Luke Robison for their help in conducting experiments and Dan Berlin for his help with the GCC compiler internals. Kathryn McKinley, Ben Wiedermann, and Adam Brown provided valuable comments on earlier drafts. This work was supported by NSF grant ACI-0313263 and a grant from the Intel Research Council.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
2. Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Programming Language Design and Implementation (PLDI)*, pages 103–114, 2003.
3. Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, 1997.
4. Manuvir Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation (PLDI)*, pages 35–46, 2000.
5. Manuel Faehndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Programming Language Design and Implementation (PLDI)*, pages 85–96, 1998.
6. Ben Hardekopf and Calvin Lin. The Ant and the Grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Programming Language Design and Implementation (PLDI)*, 2007.
7. 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)*, pages 24–34, 2001.
8. Donglin Liang and Mary Jean Harrold. Equivalence analysis and its application in improving the efficiency of program slicing. *ACM Trans. Softw. Eng. Methodol.*, 11(3):347–383, 2002.
9. 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.
10. David Pearce, Paul Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 37–42, 2004.
11. 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)*, pages 3–12, 2003.
12. Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Programming Language Design and Implementation (PLDI)*, pages 47–56, 2000.
13. Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–14, 1997.
14. Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.
15. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.
16. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis. In *Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.