Open access • Proceedings Article • DOI:10.1145/1250734.1250766

Making context-sensitive points-to analysis with heap cloning practical for the real world — Source link

Chris Lattner, Andrew Lenharth, Vikram Adve

Institutions: Apple Inc., University of Illinois at Urbana–Champaign

Published on: 10 Jun 2007 - Programming Language Design and Implementation

Topics: Pointer analysis, Heap (data structure), Call graph, Algorithmics and Pointer (computer programming)

Related papers:

- Cloning-based context-sensitive pointer alias analysis using binary decision diagrams
- Points-to analysis in almost linear time
- LLVM: a compilation framework for lifelong program analysis & transformation
- Program Analysis and Specialization for the C Programming Language
- The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code

Share this paper:

View more about this paper here: https://typeset.io/papers/making-context-sensitive-points-to-analysis-with-heap-3mtbe9p6zx

Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World *

Chris Lattner
Apple Inc.
clattner@apple.com

Andrew Lenharth
University of Illinois at Urbana-Champaign
alenhar2@cs.uiuc.edu

Vikram Adve
University of Illinois at Urbana-Champaign
vadve@cs.uiuc.edu

Abstract

Context-sensitive pointer analysis algorithms with full "heap cloning" are powerful but are widely considered to be too expensive to include in production compilers. This paper shows, for the first time, that a context-sensitive, field-sensitive algorithm with full heap cloning (by acyclic call paths) can indeed be both scalable and extremely fast in practice. Overall, the algorithm is able to analyze programs in the range of 100K-200K lines of C code in 1-3 seconds, takes less than 5% of the time it takes for GCC to compile the code (which includes no whole-program analysis), and scales well across five orders of magnitude of code size. It is also able to analyze the Linux kernel (about 355K lines of code) in 3.1 seconds. The paper describes the major algorithmic and engineering design choices that are required to achieve these results, including (a) using flow-insensitive and unification-based analysis, which are essential to avoid exponential behavior in practice; (b) sacrificing context-sensitivity within strongly connected components of the call graph; and (c) carefully eliminating several kinds of $O(N^2)$ behaviors (largely without affecting precision). The techniques used for (b) and (c) eliminated several major bottlenecks to scalability, and both are generalizable to other context-sensitive algorithms. We show that the engineering choices collectively reduce analysis time by factors of up to 3x-21x in our ten largest programs, and that the savings grow strongly with program size. Finally, we briefly summarize results demonstrating the precision of the analysis.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Algorithms

Keywords Pointer analysis, context-sensitive, field-sensitive, interprocedural, static analysis, recursive data structure

* This work is supported in part by NSF under grant numbers EIA-0093426, EIA-0103756, CCR-9988482 and CCF-0429561, and in part by the University of Illinois.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006. . . $5.00

1. Introduction

Context-sensitive alias analysis algorithms have been studied intensively over the last two decades, with impressive improvements in algorithmic scalability [7, 10, 35, 14, 13, 22, 11, 12, 23, 34]. To achieve true context-sensitivity, such algorithms must distinguish heap objects by (acyclic) call paths, not just by allocation site; this property is sometimes referred to as "heap cloning" [26]. Heap cloning is important because it allows analyses to distinguish different *instances* of a logical data structure created at different places in a program, even if the data structure is implemented with a common set of functions (e.g., a data structure library) that allocate memory internally. Such programming patterns are increasingly prevalent particularly in object-oriented programs, where reusable libraries are an important feature. For example, in the LLVM compiler system [19], which is written in C++, there are no less than 25 static occurrences of a single class (`vector<unsigned>`). Heap cloning also allows analyses to handle allocations that occur through one or more levels of wrappers in a simple, general manner instead of handling them as a special case (e.g., single-level wrappers [15]). More quantitatively, Nystrom et al. [26] show that for many programs, naming heap objects only by allocation site (the most common alternative) significantly reduces analysis precision compared with heap specialization by call paths, up to some threshold.

Unfortunately, there is widespread skepticism that algorithms with heap cloning can be scalable and fast enough to be included in production compilers. To date, this skepticism is arguably justified: *we know of no previous paper that demonstrates that an alias analysis that uses heap cloning is scalable and fast enough for use in production compilers*. Section 6 discusses the current state of the art in more detail. Briefly, two recent algorithms that use different forms of cloning with a subset-based (rather than a unification-based) analysis have been to shown to be scalable but are still quite slow in absolute running times [34, 25]. At least for now, such algorithms appear too slow to be used in a production compiler, although they may be reasonable for static analysis tools. On the other hand, there are several algorithms that use unification to control the exponential blow-up that can occur with context-sensitivity (with or without heap cloning). The MoPPA algorithm by Liang and Harrold [23] uses heap cloning and is quite fast but exhausts available memory on a workstation with 640MB of memory for their largest program, `povray`. Their results (and many others) justify the common concern that, in practice, memory consumption can be a limiting factor in scaling context-sensitive analyses. Many other context-sensitive alias analysis papers either do not use heap cloning [22, 11, 12] or (particularly in earlier papers) only report results for relatively small programs of a few thousand lines of code or less and typically make limited or no claims about scalability [7, 10, 35, 14, 13, 27].

In this paper, we describe an algorithm named **Data Structure Analysis** (DSA), and use it to present strong evidence that a context-sensitive, field-sensitive algorithm with full heap cloning (by acyclic call paths) can indeed be both scalable and extremely fast in practice. DSA is able to analyze programs in the range of 100K-220K lines of C code in 1-3 seconds, taking less than 5% of the time it takes GCC to compile the program at -O3 (which does no whole-program analysis), and scaling well across five orders of magnitude of code size in terms of both analysis time and memory usage. It is also able to analyze the Linux kernel (about 355K lines of compiled code in our configuration) in 3.1 seconds. These analysis times and scaling behavior appear to be very reasonable for production compilers.

More specifically, DSA is a points-to analysis algorithm that is field-sensitive and context-sensitive with full heap cloning (by acyclic call paths). Like several previous papers, we combine these with a flow-insensitive and unification-based approach to improve scalability [22, 11, 12, 23]. There are three novel features in our algorithm itself:

- First, DSA uses a new extension of Tarjan's SCC finding algorithm to incrementally construct a call graph during the analysis *without any iteration*, even in the presence of function pointers and recursion. An algorithm by Fähndrich et al. [11] is the only previous work we know that achieves the same property, but they do it by using a constraint instantiation method that appears difficult to extend to incomplete programs, which is essential for production compilers in practice.
- Second, it uses a fine-grain "completeness" tracking technique for points-to sets as a unified solution for several difficult problems that have usually been solved via different techniques before (or ignored): (a) supporting field-sensitivity for non-type-safe languages; (b) correctly supporting incomplete programs, i.e., with *unknown* external functions; and (c) constructing a call graph incrementally during the analysis. This technique can be used in any alias analysis that explicitly tracks reachability of objects (typically, analyses that use an explicit representation of memory rather than just "aliasing pairs" of references [17]).
- Third, DSA includes several essential engineering choices developed over time to avoid $O(N^2)$ behaviors that were a major bottleneck to scalability in practice. Two of these have been described previously (the globals graph [28, 23] and efficient inlining [28]) but no experimental results on their benefits have been reported. These choices are generalizable to other context-sensitive algorithms. Our experiments show that each choice contributes substantial speedups, and they collectively achieve 3x–21x reduction in analysis time in our ten largest programs. furthermore this reduction increases strongly with program size, demonstrating that they eliminate significant scalability bottlenecks.

In addition to speed and scalability, the algorithm has several *practical* strengths that we consider valuable for real-world compilers. Perhaps most important, DSA correctly handles incomplete programs (i.e., programs with *unknown* external functions): it produces conservative results while still trying to provide aliasing information for as much of a program as possible. To our knowledge, *none* of the previous context-sensitive, unification-based algorithms [22, 11, 12, 23] can correctly handle incomplete programs. The algorithm supports the full generality of C/C++ programs, including type-unsafe code, function pointers, recursion, `setjmp/longjmp`, C++ exceptions, etc. The algorithm does not require a call graph as input, as noted earlier (even though it is non-iterative).

We compare the precision of our algorithm for alias analysis to Andersen's algorithm [2] (a context-insensitive subset-based algorithm), showing that DSA is about as precise Andersen's for many cases, is significantly more precise for some programs, and is only worse in rare cases. Further, other work [18] shows that the mod-/ref information captured by DSA is significantly better than that computed by non-context-sensitive algorithms.

In addition to alias analysis, DSA can also be used to extract limited information about entire linked data structures: identifying *instances* of these structures, bounding lifetimes of each instance, and extracting available (flow-insensitive) structural and type information for each identified instance (this capability is the source of the name Data Structure Analysis).

Although this information is significantly more limited than full-blown shape analysis [29, 4, 16], it is sufficient for many interesting applications. One such application is the *Automatic Pool Allocation* transformation [20], which segregates heap objects into distinct pools if the points-to graph can distinguish subsets. This can improve spatial locality significantly for recursive data structure traversals, and can enable other compiler and run-time techniques that wish to modify (or measure) per-data-structure behavior [21]. Another example is the SAFECode compiler for C [8], which enforces safety properties (memory safety, control-flow integrity, type safety for a subset of objects, and analysis soundness) automatically and with relatively low overhead for unmodified C programs. Type homogeneity of points-to sets enables elimination of run-time checks in SAFECode, reducing its overhead.

The full source code for DSA can be downloaded via anonymous CVS at `llvm.org`. Because only the first ("local") phase of the algorithm directly inspects the program representation, the implementation should be relatively straightforward to incorporate into other mid-level or low-level compiler systems written in C++.

The next section precisely defines the points-to graph representation computed by our analysis. Section 3 describes the algorithm and analyzes its complexity. Section 4 describes several important engineering choices required for scalability. Section 5 describes our experimental results for algorithm cost and very briefly summarizes the results of two studies of algorithm precision. Section 6 discusses related work and Section 7 briefly summarizes the major conclusions.

```
typedef struct list { struct list *Next;
                      int Data; } list;
int Global = 10;
void do_all(list *L, void (*FP)(int*)) {
  do { FP(&L->Data);
       L = L->Next;
  } while(L);
}
void addG(int *X) { (*X) += Global; }
void addGToList(list *L) { do_all(L, addG); }
list *makeList(int Num) {
  list *New = malloc(sizeof(list));
  New->Next = Num ? makeList(Num-1) : 0;
  New->Data = Num; return New;
}
int main() {    /* X & Y lists are disjoint */
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  Global = 20;
  addGToList(Y);
}
```

Figure 1. C code for running example

2. The Data Structure Graph

Data Structure Analysis computes a graph we call the Data Structure Graph (DS graph) for each function in a program, summarizing the memory objects accessible within the function along with their connectivity patterns. Each DS graph node represents a (potentially infinite) set of dynamic memory objects, and distinct nodes represent disjoint sets of objects, i.e., the graph is a finite, static partitioning of the memory objects. All dynamic objects which may be pointed to by a single pointer variable or field are represented as a single node in a graph.

In defining the DS graph, we assume that input programs have a simple type system with structural equivalence, having primitive integer and floating point types of predefined sizes, plus four derived types: pointers, structures, arrays, and functions. The analysis explicitly tracks points-to properties only for pointer types and integer types of the same size or larger; we call these *pointer-compatible* types (other values are treated very conservatively if converted into a *pointer-compatible* type). For any type τ, $fields(\tau)$ returns a set of field names of τ. This is a single degenerate field name if τ is a scalar type or function type. An array type of known size k may be represented either as a structure with k fields (if all index expressions into the array are compile-time constants) or by a single field; an unknown-size array is always represented as the latter. We also assume a load/store program representation in which virtual registers and memory locations are distinct, it is not possible to take the address of a virtual register, and virtual registers can only represent scalar variables (i.e., integer, floating point, or pointer). Specific operations in the input program representation are described in Section 3.2.

2.1 Graph Definition

The DS graph for a function is a finite directed graph represented as a tuple $DSG(F) = \langle N, E, E_V, N_{call} \rangle$, where:

- N is a set of nodes, called "DS nodes". DS nodes have several attributes described in Section 2.2 below.
- E is a set of edges in the graph. Formally, E is a function of type $\langle n_s, f_s \rangle \rightarrow \langle n_d, f_d \rangle$, where $n_s, n_d \in N$, $f_s \in fields(T(n_s))$ and $f_d \in fields(T(n_d))$, and $T(n)$ denotes type information computed for the objects of n as explained below. We refer to a $\langle$node,field$\rangle$ pair as a "*cell*". E is a function because a source field can have only a single outgoing edge. Non-pointer-compatible fields (and virtual registers) are mapped to $\langle null, 0 \rangle$.
- E_V is a partial function of type $vars(f) \rightarrow \langle n, f \rangle$, where $vars(f)$ is the set of virtual registers in scope in function f. This includes global variables, which are treated as virtual registers of pointer type with global scope, pointing to an unnamed global memory object. Conceptually, $E_V(v)$ is an edge from register v to the target field $\langle n, f \rangle$ pointed to by v, if v is of pointer-compatible type.
- $N_{call} \subset N$ is a set of "call nodes" in the graph, which represent unresolved call sites within the current function or one of its (immediate or transitive) callees. Each call node $c \in N_{call}$ is a $k+2$ tuple: $(r, f, a_1, \ldots, a_k)$, where every element of the tuple is a node-field pair $\langle n, f \rangle$. r denotes the value returned by the call (if it is pointer-compatible), and f the set of possible callee functions. $a_1 \ldots a_k$ denote the values passed as arguments to the call. Conceptually, each tuple element can also be regarded as a points-to edge in the graph.

To illustrate the DS graphs and the analysis algorithm, we use the code in Figure 1 as a running example. This example creates and traverses two disjoint linked lists, using iteration, recursion, function pointers, a pointer to a subobject, and a global variable reference. Despite the complexity of the example, Data Structure Analysis is able to prove that the two lists X and Y are disjoint. *Naming heap objects by allocation site would prevent the two lists from being disambiguated* because nodes of both are allocated at the same site. The final DS graph computed for `main` is shown in Figure 10.

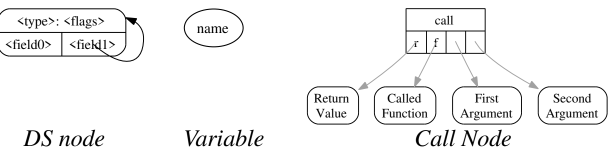


Figure 2. Graph Notation

To illustrate the DS graphs computed by various stages of our algorithm, we render DS graphs using the graphical notation shown in Figure 2. Figure 3 shows the initial ("local") graphs computed for the `do_all` and `addG` functions, before any interprocedural information is applied. The figure includes an example of a call node, which (in this case) calls the function pointed to by `FP`, passing the address of the field pointed to by `L->data` as an argument, and ignores the return value of the call.

2.2 Graph Nodes and Fields

Each DS node n has three pieces of information describing the memory objects corresponding to that node:

- $T(n)$: a language-specific type for the memory objects represented by n. This type determines the number of fields and outgoing edges in a node. Note that fields are tracked separately only for "type-homogeneous" nodes, as explained below.
- $G(n)$: a set of global objects represented by n. Includes functions, which represent the targets of function pointers and of the f field in call nodes.
- $flags(n)$: set of flags associated with $n \subseteq$ { **H**,**S**,**G**,**U**, **A**, **M**,**R**, **C**, **O** }. These flags are defined below.

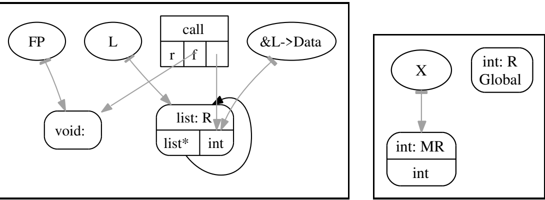


Figure 3. Local DSGraphs for `do_all` and `addG`

Storage class flags (H, S, G, U): The **'H'**, **'S'**, **'G'** and **'U'** flags in $flags(n)$ are used to distinguish **H**eap, **S**tack, **G**lobal (including functions), and **U**nknown objects. Multiple flags may be present in a single DS node. The **U**nknown flag is added to a DS node when a constant integer value is cast to a pointer or when unanalyzable address arithmetic is seen: the flag signifies that the instruction creating the target object was not found (these cases are infrequent in portable programs). A node marked **'U'** must be treated as potentially overlapping with (i.e., representing common memory objects as) *any* other node in the graph. Nodes representing objects created in an external, unanalyzed function are *not* marked **'U'**, but are treated as "incomplete," as described below.

Mod/Ref flags (M, R): These flags simply mark whether store or load operations have been detected on a memory object at the node. This directly provides context-sensitive Mod/Ref information for memory objects, including global, stack and heap objects.

Completeness flag (C): The **C**omplete flag denotes that all operations on objects at a node have been processed. For example, the

list node in the local graph for function `do_all` (Figure 3) has no **C** flag because the list is accessible in an unprocessed callee, but both lists and the global in function `main` are marked **C**omplete[1]. A node may have no **C** flag even at the end of analysis if it is reachable from unavailable external functions (we find it is common to have a few such nodes in large programs, because of unavailable libraries). At any point, if a node is not marked complete, the information calculated for the DS node represents partial information and must be treated conservatively. In particular, the node may later be assigned extra edges, extra flags, a different type, or may even end up merged with another incomplete node in the graph. Formally, two nodes with no **C** flag (e.g., `L` and `FP`) may represent common objects, i.e., a client (e.g., alias analysis queries) must assume that pointers to the two nodes may alias. Nevertheless, other nodes in such a graph may be complete and such nodes will never be merged with any other node, providing useful partial information for incomplete programs. (Of course, a pointer to node with the **U** flag may alias a pointer to *any* other node, even those with **C** set). The algorithm for inferring **C** flags is described in Section 3.1.

An important benefit of the **C**omplete flag within the analysis itself is that it allows DS analysis to assume speculatively that all accesses to a node are type-safe, until an access to the node is found which conflicts with the other accesses. Because a node is not marked complete as long as there are potentially unprocessed accesses, this is safe. DSA uses this to provide field-sensitive information *for the type-safe subsets of programs*, while collapsing fields for type-unsafe structure types because tracking fields for such types can be expensive [30].

Collapsed flag (O): The collapsed flag (O) marks nodes representing multiple, incompatible types of objects. More precisely, if all *uses* of objects in a node (or to a field of the node) follow a consistent type τ (or the field type within τ), then DSA assigns $T(n) = \tau$; we refer to such a node as "type-homogeneous." Here *uses* are defined as operations on pointers to the node that interpret the type, viz., loads, stores, or structure and array indexing. In particular, pointer casts (e.g., from `void*`) are not counted as uses. If uses with incompatible types (as defined in Section 3) are found, we no longer track the type or fields of objects represented by the node. We mark the node with the **O** flag (c**O**llapsed), set $T(n)$ = `char[]`, i.e., an unsized array of bytes, and merge all outgoing edges into a single edge. We do this using the following algorithm:

```
collapse(dsnode n)
  cell e = ⟨null, 0⟩                          // null target
  ∀f ∈ fields(T(n))
    e = mergeCells(e, E(⟨n, f⟩))              // merge old target with e
    remove field f                            // remove old edge
  T(n) = char*                                // reset type information
  E(⟨n, 0⟩) = e                               // new edge from field 0
  flags(n) = flags(n) ∪ {'O', A}              // node is cOllapsed, Array
```

The function "*mergeCells*(c_1, c_2)" (described in the next section) merges the cells c_1 and c_2 and therefore the nodes pointed to by those cells. This ensures that the targets of the two cells are now exactly equal. The result is the same as if the type information was *never speculated* for node n.

Array flag (A): This flag is added to a node if any array indexing expresssion is applied to a pointer targeting that node. Note that this does not eliminate type homogeneity for non-collapsed nodes: the node may represent singleton objects and arrays of objects of type τ. The **A** flag is always set for Collapsed nodes since the degenerate type they use is an array of bytes.

[1] This is somewhat similar to the "inside nodes" of [33].

3. Construction Algorithm

DS graphs are created and refined in a three step process. The first phase constructs a DS graph for each function in the program, using only intraprocedural information (a "local" graph). This is the only phase that inspects the actual program representation; the next two phases operate solely on DS graphs. Second, a "Bottom-Up" analysis phase is used to eliminate incomplete information due to callees in a function, by incorporating information from callee graphs into the caller's graph (creating a "BU" graph). By the end of the BU phase, the call graph construction is also complete. The final "Top-Down" phase eliminates incomplete information due to incoming arguments by merging caller graphs into callees (creating a "TD" graph). Both, the BU and TD phases operate on the known (i.e., partial or completed) Strongly Connected Components (SCCs) in the call graph.

Two properties are important for understanding how the analysis works in the presence of incomplete programs, and how it can incrementally construct the call graph even though it operates on the SCCs of the graph. First, the DS graph for a function is correct even if only a subset of its potential callers and potential callees have been incorporated into the graph (i.e., the information in the graph can be used safely so long as the limitations on nodes without **'C'** flags are respected, as described in Section 2). Intuitively, the key to this property simply is that a node must not be marked complete until it is known that all callers and callees potentially affecting that node have been incorporated into the graph. Second, the result of two graph inlining operations at one or two call sites is independent of the order of those operations. This follows from a more basic property that the order in which a set of nodes is merged does not affect the final result.

3.1 Primitive Graph Operations

Data Structure Analysis is a flow-insensitive algorithm which uses a unification-based memory model, similar to Steensgaard's algorithm [31]. The algorithm uses several primitive operations on DS graphs, shown in Figure 4. These operations are used in the algorithm to merge two cells (`mergeCells`), merge a callee's graph into a caller's graph at a particular call site (`resolveCallee`) and vice versa (`resolveCaller`), and compute the completeness property ('C' flag) for DS nodes (`markComplete`). The two graph-merging operations are described later in this section.

The fundamental operation in the algorithm is $mergeCells$, which merges two target nodes by merging the type information, flags, globals and outgoing edges of the two nodes, and moving the incoming edges to the resulting node. If the two fields have incompatible types (e.g., $T(n_1)$ = `int`, $f_1 = 0$, $T(n_2) = \{$`int`, `short`$\}$, $f_2 = 1$), or if the two node types are compatible but the fields are misaligned (e.g., $T(n_1) = T(n_2) = \{$`int`, `short`$\}$, $f_1 = 0$, $f_2 = 1$), the resulting node is first collapsed as described earlier before the other information is merged. Merging outgoing edges causes the target node of the edges to be merged as well; if the node is collapsed, the resulting node for n_2 will have only one outgoing edge which is merged with all the out-edges of n_1. We use Tarjan's Union-Find algorithm [32] to make the merging efficient.

The routine `markComplete` uses an efficient traversal of a DS graph, starting at formal arguments, the return node (π), and globals, though the efficient traversal is now shown in Figure 4. In the Top-Down phase, if a function is not visible to external code (i.e., all its callers have been identified), nodes reachable from formal arguments, the return value, and globals that are not externally visible are marked Complete. Identifying which functions and globals may be externally visible is done by the LLVM linker, which can link both LLVM and native code files and can be much more aggressive about marking symbols internal when linking complete programs than libraries [1].

(*Merge two cells of same or different nodes; update n_2, discard n_1*)
Cell **mergeCells**(Cell $\langle n_1, f_1\rangle$, Cell $\langle n_2, f_2\rangle$,)
  if (IncompatibleForMerge($T(n_1), T(n_2), f_1, f_2$))
    collapse n_2 (i.e., merge fields and out-edges)
  union flags of n_1 into flags of n_2
  union globals of n_1 into globals of n_2
  merge target of each out-edge of $\langle n_1, f_j\rangle$ with
      target of corresponding field of n_2
  move in-edges of n_1 to corresponding fields of n_2
  destroy n_1
  return $\langle n_2, 0\rangle$ (if collapsed) or $\langle n_2, f_2\rangle$ (otherwise)

(*Clone G_1 into G_2; merge corresponding nodes for each global*)
cloneGraphInto(G_1, G_2)
  G_{1c} = make a copy of graph G_1
  Add nodes and edges of G_{1c} to G_2
  for (each node $N \in G_{1c}$)
    for (each global $g \in G(N)$)
      merge N with the node containing g in G_2

(*Clone callee graph into caller and merge arguments and return*)
resolveCallee(Graph G_{callee}, Graph G_{caller},
      Function F_{callee}, CallSite CS)
  cloneGraphInto(G_{callee}, G_{caller})
  clear **'S'** flags on cloned nodes
  resolveArguments($G_{caller}, F_{callee}, CS$)

(*Clone caller graph into callee and merge arguments and return*)
resolveCaller(Graph G_{caller}, Graph G_{callee},
      Function F_{callee}, CallSite CS)
  cloneGraphInto(G_{caller}, G_{callee})
  resolveArguments($G_{callee}, F_{callee}, CS$)

(*Merge arguments and return value for resolving a call site*)
resolveArguments(Graph G_{merged}, Function F_C, CallSite CS)
  mergeCells(target of $CS[1]$, target of return value of F_C)
  for ($1 \le i \le min$(Numformals(F_C), NumActualArgs(CS))
    mergeCells(target of arg i at CS, target of formal i of F_C)

(*Mark nodes* **C***omplete if safe in Local, BU phases; see text for TD*)
markComplete(Graph G)
  for (each node $N \in G$), where **C** $\notin flags(N)$
    if N is reachable from call nodes or other incomplete nodes, skip it
    if N is reachable from formal arguments or $E_V(\pi)$, skip it
    otherwise, $flags(N) \cup =$ '**C**'

Figure 4. Primitive operations used in the algorithm

3.2 Local Analysis Phase

The goal of the local analysis phase is to compute a *Local DS graph* for each function, without information about callers and callees (see Figure 5). We present this analysis in terms of a minimal language which is still as powerful as C. The assumptions about the type system and memory model in this language were described in Section 2. We assume that the functions $E(X)$ and $E_V(X)$ return a new, empty node with the type of X (by invoking makeNode($typeof(X)$)), if no previous target node existed.

The "*LocalAnalysis*" first creates empty nodes for pointer-compatible virtual registers and for globals, and then does a linear scan to process each instruction of the function. We assume that operand types in all instructions are strictly checked; any non-matching operand in an operation must first be converted with an explicit cast. Also, operations that produce non-pointer-compatible values in variables or fields are simply ignored because those locations are always mapped to $\langle null, 0\rangle$ in E_V and E respectively.

We focus on a few, less obvious, cases of the local analysis here. First, note that the type of a cell, $E_V(Y)$, is updated only when Y is used in a load, store, or indexing operation. No type is inferred at `malloc`, `alloca` and cast operations. `return` instructions are handled by creating a special π virtual register which is used to capture pointer-compatible return values. Function calls result in a new call node being added to the DS graph, such as the call node for the call to `do_all` in `addGToList`, in Figure 7(a). The node gets entries for the value returned, the function pointer (for both direct and indirect calls), and each pointer-compatible argument.

(*Compute the local DS Graph for function F*)
LocalAnalysis(function F)
  Create an empty graph
  $\forall$ virtual registers R, $E_V(R)$ = makeNode($T(R)$)
  $\forall$ globals X (variables and functions) used in F
    N = makeNode($T(X)$)); $G(N) \cup = X$; $flags(N) \cup = '\mathbf{G}'$
  $\forall$ instruction $I \in F$: case I in:

  `X = malloc ...:` (*heap allocation*)
    $E_V(X) = makeNode($`void`$)$
    $flags(node(E_V(X))) \cup =$ **'H'**

  `X = alloca ...:` (*stack allocation*)
    $E_V(X) = makeNode($`void`$)$
    $flags(node(E_V(X))) \cup =$ **'S'**

  `X = *Y:` (*load*)
    mergeCells($E_V(X),\ E(E_V(Y))$)
    $flags(node(E_V(X)) \cup =$ **'R'**

  `*Y = X:` (*store*)
    mergeCells($E_V(X),\ E(E_V(Y))$)
    $flags(node(E_V(X)) \cup =$ **'M'**

  `X = &Y->Z:` (*address of struct field*)
    $\langle n, f\rangle =$ updateType($E_V(Y),\ typeof(*Y)$)
    $f' = 0$, if n is collapsed; $field(field(n, f), Z)$ otherwise
    mergeCells($E_V(X), \langle n, f'\rangle$)

  `X = &Y[idx]:` (*address of array element*)
    $\langle n, f\rangle =$ updateType($E_V(Y),\ typeof(*Y)$)
    mergeCells($E_V(X), \langle n, f\rangle$)
    $flags(node(E_V(X)) \cup =$ **'A'**

  `return X:` (*return pointer-compatible value*)
    mergeCells($E_V(\pi), E_V(X)$)

  `X = (τ) Y:` (*value-preserving cast*)
    mergeCells($E_V(X), E_V(Y)$)

  `X = Y(Z1, Z2, ... Zn):` (*function call*)
    callnode c = new callnode
    $N_{calls} \cup = c$
    mergeCells($E_V(X), c[1]$) (*return value*)
    mergeCells($E_V(Y), c[2]$) (*callee function*)
    $\forall i \in \{1...n\}$: mergeCells($E_V(Z_i), c[i+2]$)

  (Otherwise) `X = Y op Z:` (*all other instructions*)
    mergeCells($E_V(X), E_V(Y)$)
    mergeCells($E_V(X), E_V(Z)$)
    $flags(node(E_V(X))) \cup =$ **'U'**
    collapse($node(E_V(X))$)

  MarkCompleteNodes()

Figure 5. The LocalAnalysis function

Using *mergeCells* for each entry correctly merges type information in the case when the argument type does not match the type of the formal Finally, if any other instruction is applied to a pointer-compatible value, or used to compute such a value (e.g., a cast from a pointer to an integer smaller than the pointer and vice versa), any nodes pointed to by operands and the result are collapsed and the **U**nknown flag is set on the node.

The final step in the Local graph construction is to calculate which DS nodes are **C**omplete, which is done as described in Section 3.1. For a Local graph, nodes reachable from a formal argument, a global, passed as an argument to a call site, or returned by a function call may not be marked complete. For example, in Figure 7(a), neither of the nodes for the arguments to `do_all` are marked **'C'**.

3.3 Bottom-Up Analysis Phase

The Bottom-Up (BU) analysis phase refines the local graph for each function by incorporating interprocedural information from the callees of each function. The result of the BU analysis is a graph for each function summarizing the total effect of calling that function (imposed aliases and mod/ref information) without any calling context information. It computes this graph by cloning the BU graphs of all *known* callees into the caller's Local graph, merging

nodes pointed to by corresponding formal and actual arguments and by common globals. We first describe a single graph inlining operation, then explain how the call graph is discovered and traversed.

Consider a call to a function F with formal arguments $f_1,\ldots,f_n$, where the actual arguments passed are $a_1,\ldots,a_n$. The procedure *resolveCallee* in Figure 4 shows how such a call is processed in the BU phase. We describe a simple, naïve, version here; a better approach is described in Section 4. We first copy the BU graph for F using *cloneGraphInto*, which also merges targets of common globals in the caller's graph with those in the cloned graph. We then clear all **S**tack flags since stack objects of a callee are not legally accessible in a caller. Note that we cannot delete reachable nodes with **S**tack flags: the nodes may escape (making them incomplete), so we cannot tell whether other flags will be included in such a node later. We then merge the node pointed to by each actual argument a_i of pointer-compatible type with the copy of the node pointed to by f_i. If applicable, we also merge the return value in the call node with the copy of the return value node from the callee. Note that any unresolved call nodes in F's BU graph are copied into the caller's graph, and all the objects representing arguments of the unresolved call in the callee's graph are now represented in the caller as well.

(*Create a new, empty node of type τ*)
makeNode(type τ)
  n = new Node(type = τ, flags = ϕ, globals = ϕ)
  $\forall f \in fields(\tau), E(n, f) = < null, 0 >$
  return n

(*Merge type of field $\langle n, f\rangle$ with type τ*. This may
collapse fields and update in/out edges via mergeCells())
updateType(cell $\langle n, f\rangle$, type τ)
  if ($\tau \neq$ `void` $\wedge\ \tau \neq typeof(\langle n, f\rangle)$)
    m = makeNode(τ)
    return mergeCells($\langle m, 0\rangle, \langle n, f\rangle$)
  else return $\langle n, f\rangle$

Figure 6. makeNode and updateType operations

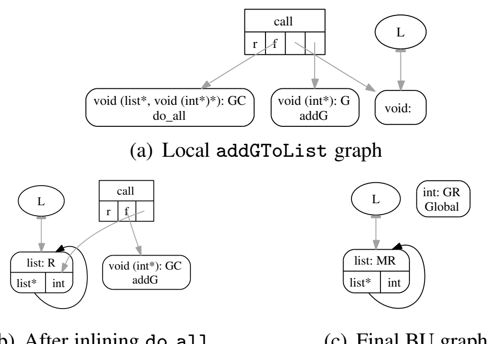
3.3.1 Basic Analysis Without Recursion

The complete Bottom-Up algorithm for traversing calls is shown in Figure 8. We explain it for four different cases. In the simplest case of a program with only direct calls to non-external functions, no recursion, and no function pointers, the call nodes in each DS graph implicitly define the entire call graph. The BU phase simply has to traverse this acyclic call graph in post-order (visiting callees before callers), cloning and inlining graphs as described above.

To support programs that have function pointers and external functions (but no recursion), we restrict our post-order traversal to only process a call-site if its function pointer targets a **C**omplete node (i.e, its targets are fully resolved, as explained in §2.2), *and* all potential callees are non-external functions (Line (1) in Figure 8).

Such a call site may become resolved if the function passed to a function pointer argument becomes known (typically, in some caller of the function containing the indirect call). For example, the call to *FP* cannot be resolved within the function `do_all`, but will be resolved in the BU graph for the function `addGToList`, where we conclude that it is a call to *addG*. Then, we clone and merge the indirect callee's BU graph into the graph of the function where the call site became resolved, using *resolveCallee* just as before (Line (2) in Figure 8). This technique of resolving call nodes as their function pointer targets are completed effectively discovers the call-graph on the fly, and preserves context-sensitivity of the analysis because the different function pointer may resolve to different callees in different contexts. We record the call graph as it is discovered for use in the TD pass.

Note that the function containing the original call still has the unresolved call node in its BU graph (and so do intervening functions into which the call node was inlined). We do not re-visit these



(a) Local `addGToList` graph

(b) After inlining `do_all` (c) Final BU graph

Figure 7. Construction of the BU DS graph for `addGToList`

functions to resolve the call node because that would lose context-sensitivity of the BU graph information; those call nodes will eventually be resolved in the top-down phase. Conceptually, the BU graph for a function acts like a procedure-summary that is used to resolve the effects of the function in different calling contexts. The BU graph for the function where the call was resolved now fully incorporates the effect of the call. For example, inlining the BU graph of `addG` into that of `addGToList` yields the finished graph shown in Figure 7(c). The **M**odified flag in the node pointed to by L is obtained from the node $E_V(X)$ from `addG` (Figure 3). This graph for `addGToList` is identical to that which would have been obtained if `addG` was first inlined into `do_all` (eliminating the call node) and the resulting graph was then inlined into `addGToList`.

After the cloning and merging for a function is done, we identify newly complete nodes (Line (5)) and remove unreachable nodes from the graph (Line (6)).

3.3.2 Recursion without Function Pointers

To handle recursion, we essentially apply the bottom-up process described above but on Strongly Connected Components (SCCs) of the call graph, handling each multi-node SCC separately. The overall Bottom-Up analysis algorithm is shown in Figure 8. DSA uses Tarjan's linear-time algorithm to find and visit Strongly Connected Components (SCCs) in the call graph in postorder.

For each SCC, all calls to functions outside the SCC are first cloned and resolved as before, as shown on lines (1) and (2) in Figure 8 (these functions will already have been visited because of the postorder traversal over SCCs). Once this step is complete, the only call nodes in the functions in the SCC are for intra-SCC calls and calls to external functions (the latter are ignored throughout, because they can never be resolved). Within an SCC, each function will eventually need to inline the graphs of all other functions in the SCC at least once (either directly or through the graph of a callee). A naïve algorithm can produce an $O(n^2)$ and even exponential number of inlining operations. To avoid this cost, we build a single BU DS Graph for an SCC (instead of each function), giving up context sensitivity within an SCC. This is accomplished by lines (3) and (4) of Figure 8, which merges BU graphs, then resolves all intra-SCC call sites (exactly once each) in the context of this single merged graph. The speed benefits of this approach are evaluated in Section 5.2.

3.3.3 Recursion with Function Pointers

The final case is a recursive program with indirect calls. The key difficulty here is that call edges are not known before-hand because they are discovered incrementally by the algorithm, but some of these call edges may induce new cycles, and hence new SCCs, in the call graph. We make a key observation, based on the properties described earlier, that yields a simple strategy to handle such situations: some call edges of an SCC can be resolved *before dis-*

```
        BottomUpAnalysis(Program P)
          ∀ Function F ∈ P
.             BUGraph{F} = LocalGraph{F}
.             Val[F] = 0; NextID = 0
          while (∃ unvisited functions F ∈ P)   (visit main first if available)
             TarjanVisitNode(F, new Stack)

        TarjanVisitNode(Function F, Stack Stk)
          NextID++; Val[F] = NextID; MinVisit = NextID; Stk.push(F)
          ∀ call sites C ∈ BUGraph{F}
             ∀ known non-external callees F_C at C
                if (Val[F_C] == 0)      (F_C unvisited)
                   TarjanVisitNode(F_C, S)
                else MinVisit = min(MinVisit, Val[F_C])
             if (MinVisit == Val[F])     (entire SCC is on the Stack)
                SCC S = { N: N = F ∨ N appears above F on stack }
                ∀ F ∈ S: Val[F] = MAXINT; Stk.pop(F)
                ProcessSCC(S, Stk)

        ProcessSCC(SCC S, Stack Stk)
          ∀ Function F ∈ S
(1)          ∀ resolvable call sites C ∈ BUGraph{F}      (see text)
                ∀ known callees F_C at C
                   if (F_C ∉ S)       (Process funcs not in SCC)
(2)                   ResolveCallee(BUGraph{F_C}, BUGraph{F}, F_C, CS)
(3)       SCCGraph = BUGraph{F_0}, for some F_0 ∈ S
          ∀ Function F ∈ S, F ≠ F_0    (Merge all BUGraphs of SCC)
             cloneGraphInto(BUGraph{F}, SCCGraph)
             BUGraph{F} = SCCGraph
(4)       ∀ resolvable call sites C ∈ SCCGraph       (see text)
             ∀ known callees F_C at C    (Note: F_C ∈ S)
                ResolveArguments(SCCGraph, F_C, CS)
(5)       MarkCompleteNodes() - Section 3.2
(6)       remove unreachable nodes
(7)       if (SCCGraph contains new resolvable call sites)
             ∀ F ∈ S: Val[F] = 0      (mark unvisited)
             TarjanVisitNode(F_0, Stk), for some F_0 ∈ S   (Re-visit SCC)
```

Figure 8. Bottom-Up Closure Algorithm

covering that they form part of an SCC. When the call site "closing the cycle" is discovered (say in the context of a function F_0), the effect of the complete SCC will be incorporated into the BU graph for F_0 though not the graphs for functions in the SCC that were handled earlier.

Based on this observation, we extended Tarjan's algorithm to revisit the functions in an SCC when it is discovered (but visiting only unresolved call sites in it). After the current SCC is fully processed (i.e., after step (6) in Figure 8), we check whether the SCC graph contains any newly inlined call nodes that are now resolvable. If so, we reset the *Val* entries for all functions in the SCC, which are used in *TarjanVisitNode* to check if a node has been visited. This causes all the functions in the *current* SCC to be revisited, but only the new call sites are processed (since other resolvable call sites have already been resolved, and will not be included in steps (1) and (4)).

For example, consider the recursive call graph shown in Figure 9(a), where the call from E to C is an indirect call. Assume this call is resolved in function D, e.g., because D passes C explicitly to E as a function pointer argument. Since the edge $E \to C$ is unknown when visiting E, Tarjan's algorithm will first discover the SCCs { F }, { E }, and then { D } (Figure 9(c)). Now, it will find a new call node in the graph for D, find it is resolvable as a call to C, and mark D as unvisited (Figure 9(b)). This causes Tarjan's algorithm to visit the "phantom" edge $D \to C$, and therefore to discover the partial SCC { B, D, C }. After processing this SCC, no new call nodes are discovered. At this point, the BU graphs for B, D and C will all correctly reflect the effect of the call from E to C, but the graph for E will not[2]. The TD pass will resolve the call from E to C (within E) by merging the graph for D into E. Note that even in this case, the algorithm only resolves each callee at each call site once: no iteration is required, even for SCCs induced by indirect calls.

Figure 10 shows the BU graph calculated for the `main` function of our example. This graph has disjoint subgraphs for the lists pointed to by X and Y. These were proved disjoint because we cloned and then inlined the BU graph for each call to `addGToList()`. This shows how context sensitivity with heap cloning can identify disjoint data structures, even when complex pointer manipulation, indirect calls and recursion are involved (and despite unification and flow-insensitivity).

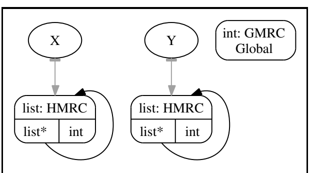


Figure 10. Finished BU graph for `main`

3.4 Top-Down Analysis Phase

The Top-Down construction phase is very similar to the Bottom-Up construction phase, and the detailed pseudo-code is omitted here but shown in [18]. The BU phase has already identified the call graph, so the TD phase can traverse the SCCs of the call graph directly using Tarjan's algorithm; it does not need to "re-visit" SCCs as the BU phase does. Note that some SCCs may have been visited only partially in the BU phase, so the TD phase is responsible for merging their graphs.

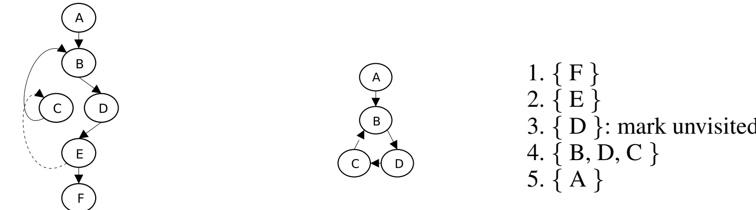
Overall, the TD phase differs from the BU phase in only 4 ways: First, the TD phase never marks an SCC as unvisited as explained above: it uses the call edges discovered and recorded by the BU phase. Second, the TD phase visits SCCs of the call graph in reverse postorder instead of postorder. Third, the Top-Down pass merges each function's graph into that of each of its callees (rather than the reverse), and it does so directly: it never needs to "defer" this inlining operation since the potential callees at each call site are known. The final difference is that formal argument nodes are marked complete if all callers of a function have been identified by the analysis, i.e., the function is not accessible to any external functions. Similarly, global variables are marked complete unless they are accessible to external functions.

3.5 Bounding Graph Size

In the common case, the merging behavior of the unification algorithm we use prevents individual data structure graphs from blowing up, and in fact, keeps them very compact. This occurs whenever a data structure is processed by a loop or recursion because, in either case, a common variable must point to instances of objects in successive iterations or successive calls. Unification then forces these objects to be merged. In contrast, subset-based analyses can easily generate exponentially large graphs [34].

Nevertheless, the combination of field sensitivity and cloning makes it theoretically possible for our algorithm to build data structure graphs that are exponential in the size of the input program. Such cases can only occur if the program builds and processes a data structure using *non-loop, non-recursive code*, and are thus *extremely* unlikely to occur in practice.

[2] Nor should it. A different caller of E may cause the edge to be resolved to a different function, thus the BU graph for E does not include information about a call edge which is not necessarily present in all calling contexts.



(a) Recursive Call Graph (indirect call $E \rightarrow C$ is dotted) (b) Call Node Edges, After inlining F & E (c) SCC visitation order

Figure 9. Handling recursion due to an indirect call in the Bottom-Up phase

Using a technique like k-limiting [17] to guard against such unlikely cases is unattractive because it could reduce precision for reasonable data structures with paths more than k nodes long. Instead, we propose that implementations either simply impose a hard limit on graph size (e.g., 10,000 nodes, which is larger than real programs are likely to need), or use a smarter heuristic which monitors and limits the growth of graphs due to cloning. If this limit is exceeded, node merging can be used to reduce the size of the graph. Our results in Section 5 show that the maximum function graph size we have observed in practice across a wide range of programs is only 3196 nodes, and this maximum is only weakly correlated to program size (in fact, in our benchmarks, it is solely determined by one large call-graph SCC, because the graphs for functions in the SCC are merged). Maximum DS graph size is only 278 in the Linux kernel and only 401 in `perlbmk` which had the next-largest call-graph SCC after the two versions of `gcc`.

3.6 Complexity Analysis

The local phase adds at most one new node, E_V entry, and/or edge for each instruction in a procedure (before node merging). Furthermore, node merging or collapsing only reduces the number of nodes and edges in the graphs. We have implemented node merging using Tarjan's union-find algorithm [32], which ensures that this phase requires $O(n\alpha(n,n))$ time and $O(n)$ space for a program containing n instructions in all [31]. $\alpha(n,n)$ is the inverse Ackerman's function.

The BU and TD phases operate on DS graphs directly, so their performance depends on the size of the graphs being cloned and the time to clone and merge each graph. We denote these by K and l respectively, where l is $O(K\alpha(K,K))$ in the worst case. They also depend on the average number of out-edges in the call graph per function, denoted c. If there are f functions, then $e = fc$ is simply the total number of edges in the call graph.

For the BU phase, each function must inline the graphs for c callee functions, on average. Because each inlining operation requires l time, the time required is $fcl = K\alpha(K)e$. The call sites within an SCC do not introduce additional complexity, since every potential callee is again inlined only once into its caller within or outside the SCC (in fact, these are slightly faster because only a single graph is built, causing common nodes to be merged). Thus, the time to compute the BU graph is $\Theta(K\alpha(K)e)$. The space required to represent the Bottom-Up graphs is $\Theta(fK)$. The TD phase is identical in complexity to the BU phase.

Putting these together, the worst case time and memory complexity are $\Theta(n\alpha(n) + K\alpha(K)e)$, and $\Theta(fK)$,

4. Major Engineering Choices

Through our experience scaling to increasingly larger programs over time, we have repeatedly found that several O(N^2) aspects of the algorithm were the main bottlenecks to scalability. In contrast, we have never seen any significant combinatorial growth from cloning: unification has been successful at preventing this problem. To scale DSA to large programs, we have devised engineering solutions to improve these N^2 problems, many of which should be applicable to other interprocedural heap or pointer analysis algorithms. The speedups achieved by these techniques are evaluated in Section 5.2.

The Globals Graph: In the algorithm so far, global variables accessed anywhere in the program would propagate bottom-up to `main`, then top-down to all functions in the program, ballooning graph size by a factor that grows as $O(N^2)$. A key optimization we add to DSA is to use a separate "Globals Graph" to hold information about global nodes and all nodes reachable from global nodes. We can then remove global variables from a function's graph if they are not used in the current function (even though they may be used in callers or callees of that function). For example, this eliminates the `G` nodes in graphs of all functions except `addG` (and `main`). Both Ruf [28] and Liang and Harrold [23] use a somewhat similar technique, but they do not motivate it primarily as an optimization and do not evaluate its impact on speed. We refer the reader to [18] for the detailed steps we use.

Shrinking E_V with Global Equivalence Classes: Even with the above refinement, programs that use large global arrays of pointers to globals can be problematic (e.g. an array of pointers to strings). The E_V entries for the target globals are replicated in every function that accesses any one of those globals. Since DSA can never disambiguate these globals, we solved this by keeping only one representative global in each DS node (and removing the rest from each graph's E_V as well). In programs with many globals, this replaces $O(N^2)$ entries with $O(N)$ in E_V.

Efficient Graph Inlining: The version of function "cloneGraphInto" shown in Figure 4 sometimes proved very slow in practice because it allocates and copies many nodes, only to discard them soon after creation. To solve this issue, we now inline graphs using a recursive traversal (of both graphs) from the common pointers, only visiting nodes that will actually be reflected into the target graph. This is possible because unification ensures a 1-1 mapping of paths in the two graphs, though there may be a many-to-one mapping of nodes from source to target. Ruf used a similar technique, although the benefit wasn't evaluated [28].

5. Experimental Results

We implemented the DSA algorithm in the LLVM Compiler Infrastructure [19]. The analysis is performed at link-time, using stubs for C library functions while treating unknown external functions conservatively. We have successfully used DSA for optimizations and/or memory safety for a wide range of programs and for the Linux kernel [18, 20, 21, 9, 8], and LLVM is used by several commercial organizations.

We present results evaluating DSA on several benchmark suites: the "ptrdist" 1.1 benchmarks, the SPEC 1995 and SPEC 2000 integer and floating point benchmarks (for those Fortran programs that we were able to convert successful to C using f2c), a few unbundled programs, and the Linux 2.4.22 kernel. `povray31` includes sources for the `zlib` and `libpng` libraries. The Linux kernel includes only a few drivers, but includes many other modules that would normally be separately compiled and loaded, including many file system and

	Code Size			Analysis Time (sec)				Mem (KB)		# of Nodes		
Benchmark	**LOC**	**MInsts**	**max |SCC|**	**Local**	**BU**	**TD**	**Total**	**BU**	**TD**	**Total**	**Max**	**Coll-apsed**
ptrdistanagram	647	271	1	0.00	0.00	0.00	0.01	111	89	163	18	11
ptrdistks	684	546	1	0.00	0.00	0.00	0.01	97	68	207	24	0
ptrdistft	1301	433	1	0.00	0.00	0.00	0.01	112	86	150	14	0
ptrdistyacr2	3212	1621	1	0.01	0.01	0.01	0.02	222	212	369	17	0
ptrdistbc	6627	3729	1	0.01	0.02	0.02	0.05	591	408	738	29	16
130.li	7598	7894	24	0.03	0.09	0.04	0.16	1948	933	806	33	328
124.m88ksim	19233	7951	2	0.03	0.03	0.02	0.08	1334	774	1796	56	195
132.ijpeg	28178	12507	1	0.03	0.02	0.02	0.07	1453	935	1531	65	62
099.go	29246	20543	1	0.04	0.02	0.04	0.10	1299	906	2298	131	0
134.perl	26870	29940	19	0.08	0.12	0.06	0.26	2038	1201	1463	232	136
147.vortex	67211	37632	23	0.09	0.14	0.07	0.30	2785	1678	3529	355	242
126.gcc	205085	129083	255	0.44	1.87	0.37	2.68	10982	6586	12226	3046	1109
145.fpppp	2784	4447	2	0.01	0.01	0.01	0.03	472	261	623	48	43
104.hydro2d	4292	5773	2	0.02	0.02	0.01	0.05	672	384	688	48	88
110.applu	3868	5854	2	0.01	0.02	0.00	0.03	518	289	583	57	19
103.su2cor	2332	6450	2	0.02	0.02	0.01	0.05	759	437	1080	160	49
146.wave5	7764	11333	2	0.03	0.01	0.01	0.05	908	521	1171	70	164
181.mcf	2412	991	1	0.00	0.00	0.01	0.01	76	53	103	49	0
256.bzip2	4647	1315	1	0.01	0.00	0.00	0.01	131	80	205	76	3
164.gzip	8616	1785	1	0.00	0.01	0.00	0.01	217	127	290	60	1
175.vpr	17728	8972	1	0.03	0.02	0.01	0.06	969	614	2106	366	118
197.parser	11391	10086	3	0.04	0.03	0.03	0.1	1301	758	1291	109	121
186.crafty	20650	14035	2	0.05	0.03	0.04	0.12	1344	817	2890	701	45
300.twolf	20459	19686	1	0.05	0.01	0.03	0.09	1179	807	2022	411	37
255.vortex	67220	37601	23	0.07	0.10	0.09	0.26	2765	1669	3515	392	241
254.gap	71363	47389	9	0.17	0.41	0.19	0.77	8040	3837	5889	370	728
252.eon	35819	51897	6	0.24	0.18	0.14	0.56	7818	4450	6936	411	511
253.perlbmk	85055	98386	250	0.31	2.11	0.24	2.65	8785	3517	2038	401	510
176.gcc	222208	139790	337	0.44	2.38	0.42	3.24	11628	7101	12736	3196	1000
168.wupwise	2184	5087	2	0.01	0.01	0.01	0.03	523	302	608	64	33
173.applu	3980	5966	2	0.02	0.01	0.01	0.04	523	291	593	68	19
188.ammp	13483	10551	1	0.00	0.00	0.01	0.01	865	546	897	281	69
177.mesa	58724	43352	1	0.10	0.06	0.07	0.23	4115	2476	3038	98	857
fpgrowth	634	544	1	0.01	0.00	0.01	0.02	58	37	108	49	0
boxed-sim	11641	12287	1	0.02	0.03	0.01	0.06	693	442	480	65	61
NAMD	5312	19002	1	0.05	0.02	0.02	0.09	1042	761	1539	224	276
povray31	108273	62734	56	0.18	0.23	0.13	0.54	5582	3389	5278	318	732
linux	355384	305073	28	1.35	1.00	0.76	3.10	14498	31232	47348	278	9666

Table 1. Program information, analysis time, memory consumption, and graph statistics.

networking modules. It is complete enough for us to use it as a standard configuration on a modern workstation.

Table 1 describes relevant properties of the benchmarks, as well as the experimental results. "LOC" is the raw number of lines of C code for each benchmark, "MInsts" is the number of memory instructions[3] for each program in the LLVM internal representation, and "SCC" is the size of the largest SCC in the call-graph for the program. We omitted eight SPEC programs of less than 2000 lines (101, 102, 107, 129, 171, 172, 179, 183) for lack of space, but their results are available in [18].

5.1 Analysis Time & Memory Consumption

We evaluated the time and space usage of our analysis on a Linux workstation with an AMD Athlon MP 2100+ processor, which runs at 1733MHz. We compiled LLVM with GCC 3.4.2 at the `-O3` level of optimization. Table 1 shows the running times and memory usage, and graph node statistics for DSA (the Local phase uses little memory and is not shown).

The six largest programs in our study, `vmlinux`, `176.gcc`, `126.gcc`, `povray31`, `253.perlbmk` and `254.gap` are both fairly large and contain non-trivial SCCs in the call graph. Nevertheless, it takes only between 0.54 and 3.24 seconds to perform the complete algorithm on these programs. To put this into perspective, we compiled the 176.gcc, 253.perlbmk, and povray31 benchmarks with our system GCC compiler at the -O3 level of optimization. GCC takes 94.7s, 47.4s, and 38.5s to compile and link these programs, even though it does not perform *any cross-file optimizations*. DSA's times are only **3.4%**, **5.6%**, and **1.4%** of the total GCC compile times for these cases, which we consider low enough to be practical for production optimizing compilers.

The table shows that memory consumption of DSA is also quite small. The peak memory (which is almost exactly equal to BU+TD) consumed for the largest code is **less than 46MB**, which seems very reasonable for a modern compiler. These numbers are noteworthy because the algorithm is performing a context-sensitive whole-program analysis *with cloning*, and memory use can often be the bottleneck in scaling such analyses to large programs.

The "# of Nodes" columns show information about the DS graph nodes. We exclude E_V entries since those are simple map entries, not actual nodes. "Total" is the *aggregate* number of nodes in the TD graphs for all functions and "Max" is the maximum size of any particular function's graph (the term K in the notation of Section 3.6). Most importantly, the table shows that Total as well as K both grow quite slowly with total program size, indicating that context-sensitive analysis and heap cloning do not cause any blowup in the points-to graphs; this would not have been the case with a subset-based analysis [34]. "Collapsed" is the total number of TD graph nodes collapsed, which happens due to incompatible types on merged nodes. In most of the largest programs, roughly about 7-15% of the nodes are collapsed, and the most common rea-

[3] Memory instructions are `load`, `store`, `malloc`, `alloca`, `call`, and structure or array indexing instructions. Analysis time and memory usage correlate better with this number than with LOC, as shown in [18].

son appears to be merging of Global nodes, which in some cases causes other nodes to be merged. In some cases, however, *unrecognized* custom memory allocators produce significantly higher fractions of Collapsed nodes, e.g., 25% in 253.perlbmk and 20% in linux.

We have also examined the scaling behavior of the analysis (these graphs are omitted here for lack of space but are available in [18]). Across programs spanning five orders of magnitude of program size, the Local and TD passes take $O(n)$ time, where n is the number of memory operations in the program (column $MInsts$ in the table). The BU phase is slightly worse, but mainly for programs with large numbers of globals such as 254.gap, 253.perlbmk and 176.gcc (in other words, number of memory operations does not fully capture the analysis complexity). We believe this behavior can be further improved with more sophisticated handling of global equivalence classes.

5.2 Importance of Optimizations

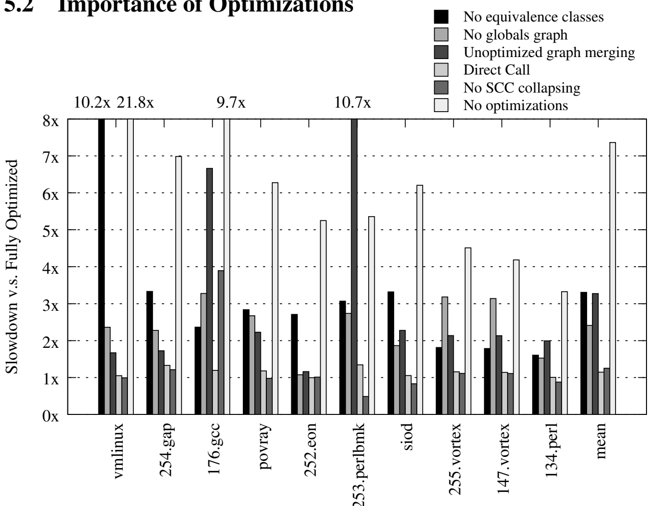


Figure 11. Relative Performance Impact of Optimizations

To investigate the effectiveness of the major optimizations, including the engineering ones described in Section 4 and the collapsing of SCCs described in Section 3.3.2, we measured the analysis times with each optimization turned off individually and also with all these optimizations off. This shows the incremental speedup each optimization provides *in the presence of all the others*, as well as the overall speedup.

In Figure 11, we show results for the 10 longest-running benchmarks in terms of unoptimized analysis time, and the average across these 10 benchmarks. The data for each program are normalized to the runtime of the fully optimized analysis, i.e, a number larger than 1.0 is a slowdown.

Overall, the speedups range from 2x to almost 16x in most cases. The two most important optimizations, on average, are equivalence classes of globals, and inlining only nodes reachable in the result graph. The former reduces the size of the nodes and the size of the lookup tables mapping IR variables to nodes. The latter reduces the number of nodes allocated during inlining as well as nodes walked during unreachable node elimination. Each of these optimizations individually produces speedups of about 1.5x-10x, and on average just over 3x.

Using a separate globals graph is second best for several programs; it reduces the size of each graph, and makes graph inlining faster. It gives speedups ranging from 1.0x to 3.5x, and on average over 2x. The two optimizations related to globals reduce memory usage considerably, e.g., by 67% and 55% respectively for 176.gcc.

Note that the Linux kernel is substantially different from other large programs. It has a very large number of globals, which explains the over 10x improvement with global equivalence classes. The globals graph gives a smaller speedup of about 2.3x, perhaps because of shallow call depths (but we have not verified that). Linux also has significantly smaller graphs compared with other programs of similar size. This helps explain the speed of the analysis on the kernel, and also the unusually small improvement from efficient inlining.

Perhaps most importantly, these results show that the benefit of the optimizations is correlated with program size. To show this trend, the Table 2 below lists the benefits for the 12 largest programs by LOC, averaged across groups of four. The benefits grow strongly with program size, reflecting the $O(N^2)$ nature of the bottlenecks eliminated. Both the use of many globals and having to inline large graphs (due to large SCCs and more indirect calls) are more common in complex, larger programs.

	Avg. LOC	Total opt. speedup
Largest 4 programs	280k	10.8x
Second largest 4	72k	4.37x
Third largest 4	52k	2.74x

Table 2. Speedup trend for 12 largest programs

5.3 Analysis of DSA Precision

DSA is powerful in some respects (context sensitivity with heap cloning and field-sensitivity) but weak in others (unification and flow insensitivity). We compared the precision of DSA with several other standard pointer analysis algorithms, for several clients, including alias analysis, interprocedural mod/ref analysis, and a few dataflow optimizations [18]. We present results for an alias analysis client here and briefly summarize the mod/ref results.

Figure 12 compares the alias analysis precision of DSA against four other pointer analysis algorithms: *local*: a simple/fast local analysis that traverses SSA edges intraprocedurally and includes base-offset analysis; *steens-fi* and *steens-fs*: implementations of Steensgaard's [31] unification-based, context-insensitive algorithm, in field-insensitive and field-sensitive versions; and *anders*: Andersen's [2] context-insensitive, field-insensitive algorithm. Every other algorithm "falls back" (if it answers "may alias") to *local*, ensuring that any query resolvable by the local algorithm will be resolved by all analyses. This means that comparing against the results for **local** alone isolates the actual benefit each of the more aggressive analyses would provide in a realistic compiler (such chaining of alias analyses has been shown to be important in practice [15]). Note that *local* can provide more precise answers than the interprocedural queries in some cases, because it benefits from simple flow sensitivity by traversing SSA edges within a procedure. In fact, *local* can return "must alias" on some queries whereas the flow-insensitive algorithms (all the others) cannot.

The evaluation performs a set of alias queries of all pairs of pointer variables (which include intermediate pointers obtained by indexing into a struct or array) within each function in a program and counts the number of queries that return "may alias" (i.e., cannot be proven to be "no alias" or "must alias"). A lower percentage therefore corresponds to a more precise analysis. Because this evaluates alias pairs within a function, it approximates the use of an abstract intraprocedural optimization client. Figure 12 focuses on the C/C++ SPEC CPU2000 benchmarks, which have been widely studied by other work, including [15].

Briefly, the major findings are:

- The alias disambiguation precision of DSA is comparable to Andersens algorithm in many cases (256.bzip2, 164.gzip, 183.equake, 176.gcc, etc). As expected, there are cases where unification reduces precision (197.parser, 255.vortex), but the

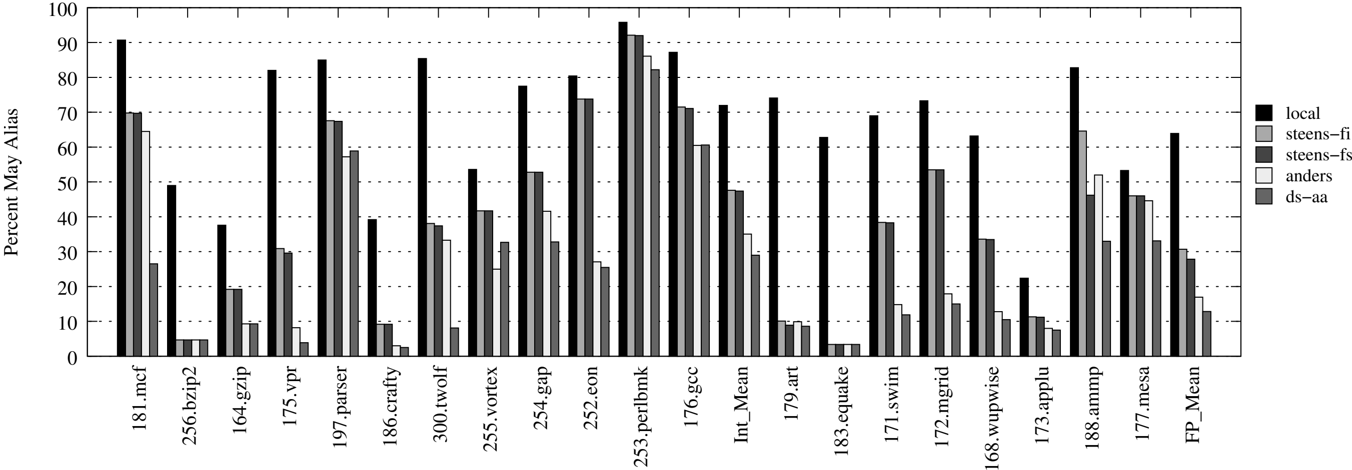


Figure 12. Percentage of alias queries that return "May Alias" (lower is more precise)

difference is quite small in both cases[4]. There are many more cases (181.mcf, 175.vpr, 186.crafty, 254.gap, 300.twolf, 188.ammp, 177.mesa, etc) where the combination of context sensitivity and unification are significantly more precise than non-context-sensitive subset-based analysis. This is partly because (as other researchers [5, 11] have shown) bidirectional argument binding is the leading cause of precision loss in a unification-based analysis. This problem can be solved by using either context sensitivity or a subset-based analysis. Conversely, the context sensitivity can help precision significantly in some cases that are not handled well by subset-based analyses.

- Using a cloning-based context-sensitive analysis can yield far more accurate points-to results than using a static naming scheme for heap and stack objects. The effect is most pronounced in programs that use a large amount of heap allocated data and have few static allocation sites. For example, 175.vpr, 300.twolf, and 252.eon have simple wrapper functions around malloc that would prevent ordinary context-insensitive algorithms from disambiguating many pairs of heap references. Further experiments are needed to determine how deep the context-sensitivity must go to achieve these results [26].
- Comparing **steens-fi** to **steens-fs**, we see that field sensitivity substantially improves the precision of unification-based analysis only in rare cases (188.ammp), but generally has only a very small effect. The most important reason is that the **local** analysis captures many common cases of field sensitivity with simple base/offset disambiguation, e.g., two references to different fields off the same pointer. Field sensitivity may be more important when combined with context-sensitivity, because there is greater likelihood of disambiguating pointers to heap objects, but we have not evaluated that hypothesis.

In our study of precision for interprocedural mod/ref analysis, the results show that DSA (like any context-sensitive algorithm) produces significantly better mod/ref information than Andersen's algorithm. For example, in 24 of the 40 programs in that study, DSA returns NoModRef 40% more often than Andersen's [18].

Finally, we have also evaluated how effective DSA is in identifying linked data structure instances and their properties. We did this by manually inspecting the DS graphs and correlating them with source code information for a few of our benchmarks [24]. The results showed that the algorithm is usually successful in identifying different kinds of linked data structures as well as their type and lifetime information, and is sometimes successful at distinguishing instances of such structures. The most common reason for failure in the latter is due to lack of flow-sensitivity.

[4] Note that this compares a field-sensitive algorithm (DSA) with a field-insensitive one (Andersen's) and so we cannot draw any definite conclusions comparing context-sensitive with subset-based analysis. Our goal here is simply to show that DSA is reasonably precise compared with state-of-the-art interprocedural analyses.

6. Related Work

There is a vast literature on pointer analysis (e.g., see the survey by Hind [17]), but the majority of that work focuses on context-insensitive analyses. We focus here on context-sensitive techniques.

Several algorithms are context-sensitive in the analysis but not the heap naming: they name heap objects by allocation site and not by call path. Liang and Harrold's FICS algorithm processes programs up to 25K lines of code in a few seconds each [22]. Fahndrich et al [11] analyze programs as large as 200K lines of code (the Spec95 version of gcc) in about 3 minutes. The Whaley-Lam algorithm takes up to 19 minutes for one program and over 10 minutes for four out of twenty programs [34]. It also potentially considers all possible acyclic call paths (e.g., they report 10^{24} paths in one program). Both of these raise concerns for production compilers. Conversely, the GOLF algorithm [6], which is both fast and scalable, uses only one level of context-sensitivity. The key question is whether full cloning of heap objects, which we term "full context sensitivity," can achieve similar speed and scalability.

The most similar algorithm to ours is called MoPPA [23]. It is also flow-insensitive and context-sensitive, uses unification, and its structure is similar to our algorithm, including Local, Bottom-Up, and Top-Down phases, and uses a separate Globals Graph. MoPPA seems to require much higher memory than our algorithm: it runs out of memory analyzing `povray3` with field-sensitivity on a machine with 640M of memory. It has several other practical limitations: it can only use field-sensitivity for *completely* type-safe programs, requires a complete program, and requires a precomputed call-graph. We avoid these limitations via fine-grain completeness tracking and the use of call nodes for incompletely resolved calls.

Ruf's synchronization removal algorithm for Java [28] also shares several important properties with ours and with MoPPA, including combining context-sensitivity with unification, a non-iterative analysis with local, bottom-up and top-down phases, and node flags to mark global nodes. Unlike our algorithm, his work requires a call graph to be specified, it is limited to type-safe programs, and does not appear to handle incomplete programs.

A few papers describe context-sensitive algorithms with full heap cloning *without* using unification [3, 25]. Although the latter [25] has been shown to scale to quite large programs, which is an impressive result for a non-unification-based context-sensitive analysis, it does not provide convincing evidence that the technique could be used in production compilers. The work of Nystrom et al. is about 2 orders of magnitude slower than ours in absolute compile time, e.g., 192 s. for 176.gcc compared with 3.24 seconds for ours, *measured on similar systems*.

7. Conclusion

The experimental results in this paper provide strong evidence that a context-sensitive points-to analysis with full heap cloning (by acyclic call paths) can be made efficient and practical for use in production compilers. Our algorithm requires only a few seconds for programs of 100-350K lines of code and scales well across five orders of magnitude of program size. Some of the key technical ideas emerging from this work are the careful design choices to construct the call graph incrementally during the analysis, to eliminate $O(N^2)$ behaviors, and to track fine-grained completeness information as a unified solution for several difficult algorithmic problems.

Despite important simplifications for scalability (unification and flow-insensitivity), the analysis is powerful enough to provide better alias analysis precision than Andersen's algorithm in several programs. DSA can distinguish instances of logical data structures in programs and provide useful compile-time information about such data structures, without an expensive shape analysis. In fact, DSA, in combination with Automatic Pool Allocation [20], enables novel analyses and transformations that can operate *on entire recursive data structures* instead of individual loads and stores or data elements [18].

References

[1] LLVM Link Time Optimization: Design and Implementation. http://llvm.org/docs/LinkTimeOptimization.html.

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

[3] B.-C. Cheng and W. mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *PLDI*, Vancouver, British Columbia, Canada, June 2000.

[4] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.

[5] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.

[6] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, 2001.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241, June 1994.

[8] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *PLDI*, June 2006.

[9] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Trans. on Embedded Computing Systems*, Feb. 2005.

[10] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, Orlando, FL, June 1994.

[11] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.

[12] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *Proc. Int'l Symp. on Static Analysis (SAS)*, London, UK, 2000.

[13] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[14] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[15] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *PLDI*, 2001.

[16] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, New York, NY, USA, 2005.

[17] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, 2001.

[18] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Comp. Sci. Dept., Univ. of Illinois, Urbana, IL, May 2005.

[19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Int'l Symp. on Code Generation and Optimization*, Mar 2004.

[20] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI*, Chicago, IL, Jun 2005.

[21] C. Lattner and V. Adve. Transparent Pointer Compression for Linked Data Structures. In *MSP*, Chicago, IL, Jun 2005.

[22] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *ESEC*, 1999.

[23] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. In *SAS 2001*, July 2001.

[24] P. Meredith, B. Pankaj, S. Sahoo, C. Lattner, and V. Adve. How successful is data structure analysis in isolating and analyzing linked data structures? Tech. Report UIUCDCS-R-2005-2658, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Nov 2005.

[25] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS 2004*, 2004.

[26] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 43–48, New York, NY, USA, 2004.

[27] R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 338–348, New York, NY, USA, 1997. ACM Press.

[28] E. Ruf. Effective synchronization removal for java. In *PLDI*, pages 208–218, 2000.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1), Jan. 1998.

[30] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Compiler Construction*, pages 136–150, London, UK, 1996.

[31] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[32] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[33] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *PLDI*, pages 35–46, 2001.

[34] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[35] R. P. Wilson and M. S. Lam. Effective context sensitive pointer analysis for C programs. In *PLDI*, pages 1–12, June 1995.