Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second[1]

Nevin Heintze
Research, Agere Systems
(formerly Lucent's Microelectronics Division)
nch@agere.com

Olivier Tardieu
Ecole des Mines, Paris
olivier.tardieu@mines.org

ABSTRACT
We describe the design and implementation of a system for very fast points-to analysis. On code bases of about a million lines of unpreprocessed C code, our system performs field-based Andersen-style points-to analysis in less than a second and uses less than 10MB of memory. Our two main contributions are a database-centric analysis architecture called compile-link-analyze (CLA), and a new algorithm for implementing dynamic transitive closure. Our points-to analysis system is built into a forward data-dependence analysis tool that is deployed within Lucent to help with consistent type modifications to large legacy C code bases.

1. INTRODUCTION
The motivation for our work is the following software maintenance/development problem: given a million+ lines of C code, and a proposed change of the form "change the type of this object (e.g. a variable or struct field) from type1 to type2", find all other objects whose type may need to be changed to ensure the "type consistency" of the code base. In particular, we wish to avoid data loss through implicit narrowing conversions. To solve this problem, we need a global data-dependence analysis that in effect performs a forward data-dependence analysis (Section 2 describes this analysis, and how it differs from other more standard dependence analyses in the literature.). A critical part of this dependence analysis is an adequate treatment of pointers: for assignments such as `*p = x` we need to determine what objects `p` could point to. This kind of aliasing analysis is commonly called points-to analysis in the literature [4]. The scalability of points-to analysis has been a subject of intensive study over the last few years [5, 8, 21, 11, 23]. However the feasibility of building interactive tools that employ some form of "sufficiently-accurate" pointer analysis on million line code-bases is still an open question.

The paper has two main contributions. The first is an architecture for analysis systems that utilizes ideas from indexed databases. We call this architecture compile-link-analyze (CLA), in analogy with the standard compilation process. This architecture provides a substrate on which we can build a variety of analyses (we use it to implement a number of different algorithms for Andersen-style points-to analysis, dependence analysis and a unification-style points-to analysis, all using a common database format for representing programs). It scales to large code bases and supports separate and/or parallel compilation of collections of source files. Also, its indexing structures support rapid dynamic loading of just those components of object files that are needed for a specific analysis, and moreover after reading a component we have the choice of keeping it in memory or discarding it and re-reading it if we even need it again (this is used to reduce the memory footprint of an analysis). We describe CLA in detail in Section 4, and discuss how it differs from other approaches in the literature, such as methods where separate files are locally analyzed in isolation and then the individual results are combined to analyze an entire code-base.

The second contribution is a new algorithm for implementing dynamic transitive closure (DTC). Previous algorithms in the literature for Andersen's analysis are based on a transitively closed constraint graph e.g. [4, 10, 11, 21, 23, 22]. In contrast, our algorithm is based on a pre-transitive graph i.e. we maintain the graph in a form that is not transitively closed. When information about a node is requested, we must perform a graph reachability computation (as opposed to just looking up the information at the node itself in the case of a transitively closed constraint graph). A direct implementation of the pre-transitive graph idea is impractical. We show how two optimizations – caching of reachability computations, and cycle elimination – yield an efficient algorithm. Cycle elimination has previously been employed in the context of transitively closed graph and shown to result in significant improvement [11], however in that work the cost of finding cycles is non-trivial and so completeness of cycle detection is sacrificed in order to contain its cost. However, in the pre-transitive setting, cycle detection is essentially free during graph reachability. We describe our algorithm in detail in Section 5.

Section 6 presents various measurements of the performance of our system. For the Lucent code bases for which our system is targeted, runtimes are typically less than a second (800MHz Pentium) and space utilization is about 10MB.

[1] This is a substantially revised version of [16].

These code bases are in excess of a million lines of code (uncommented non-blank lines of source, before pre-processing). On gimp (a publicly available code base of about 440K lines), our system performs field-based Andersen-style points-to analysis in about a second (800MHz Pentium) and uses about 12MB. We also present data to illustrate the space advantages of CLA.

2. MOTIVATING APPLICATION – DEPENDENCE ANALYSIS

Our points-to analysis system is built into a forward data-dependence analysis tool that is deployed within Lucent to help with consistent type modifications to large legacy C code bases. The basic problem is as follows: suppose that the range of values to be stored in a variable must be increased to support additional system functionality. This may require changing the type of a variable, for example from `short` to `int`. To avoid data loss through implicit narrowing conversions, any objects that take values from the changed variable must also have their types appropriately altered. Consider the following program fragment.

```
short x, y, z, *p, v, w;
y = x;
z = y+1;
p = &v;
*p = z;
w = 1;
```

If the type of `x` is changed from `short` to `int`, then we may also have to change the types of `y`, `z`, `v` and probably `p`, but we do not need to change the type of `w`.

Given an object whose type must be changed (the *target*), we wish to find all other objects that can be assigned values from the specified object. This is a forward dependence problem, as opposed to backwards dependence used for example in program slicing [25]. Moreover it only involves data-dependencies, as opposed to *both* data-dependencies and control-dependencies which are needed in program slicing. Our analysis refines forward data-dependence analysis to reflect the importance of a dependency for the purposes of consistent type changes. The most important dependencies are those involving assignments such as `x = y` and `z = y+1`. On the other hand, an assignment such as `z1 = !y` can be ignored, since changing the type of `y` has no effect on the range of values of `z1`, and so the type of `z1` does not need to be changed. Assignments involving operations such as division and multiplication are less clear. We discuss this later in the section.

An important issue for the dependence analysis is how to treat `structs`. Consider the program fragment involving structs in Figure 1. If the target is the variable `target`, then `u`, `w` and `s.x` are all dependent objects. If the type of `target` is changed from `short` to `int`, then the types of `u`, `w` and `s.x` should also be changed. To effect this change to the type of `s.x`, we can either change the type of the `x` field of the struct `S`, or we can introduce a new struct type especially for `s`. The advantage of the former case is that we make minimal changes to the program. The disadvantage is that we also change the type of `t.x`, and this may not be strictly necessary. However, in practice it is likely that if we have to change the type of the `x` field of `s`, then we will have to change the type of the `x` field of `t`. As a result, it is desirable to treat objects that refer to the same field in a uniform way. By "same field", we mean not just that the fields have the same name, but that they are the same field of the same struct type.

Table 1: Classification of operations.

Operations	Argument 1	Argument 2
`+`, `-`, `|`, `&`, `^`	Strong	Strong
`*`	Weak	Weak
`%`, `>>`, `<<`	Weak	None
unary: `+`, `-`	Strong	n/a
`&&`, `||`	None	None
`!`	None	n/a

Since our ultimate use of dependence analysis is to help identify objects whose type must be changed, we are not just interested in the set of dependent objects. Rather, we need to give a user information about *why* one object is dependent on another. To this end, we computes the dependence chains, which identify paths of dependence between one object and another. In general there are many dependence paths between a pair of objects. Moreover, some paths are more important than others. Dependencies arising from direct assignments such as `x=y` are usually the most important; dependencies involving arithmetic operations `x=y+1`, `x=y>>3`, `x=42%y`, `x=1<<y` are increasingly less important. Our metric of importance is biased towards operations that are likely to preserve the shape and size of input data. Table 1 outlines a simple strong/weak/none classification that we have employed. Our analysis computes the most important path, and if there are several paths of the same importance, we compute the shortest path.

Large code bases often generate many dependent objects – typically in the range 1K-100K. To help users sift through these dependent objects and determine if they are objects whose type must be changed, we prioritize them according to the importance of their underlying dependence chain. We also provide a collection of graphic user interface tools for browsing the tree of chains and inspecting the corresponding source code locations. In practice, there are often too many chains to inspect – a common scenario is that a central object that is not relevant to a code change becomes dependent (often due the context- or flow-insensitivity of the underlying analysis), and then everything that is dependent on this central object also becomes dependent. We address this issue with some additional domain knowledge: we allow the user to specify "non-targets", which are objects that the user knows are certainly not dependent on the target object. This has proven to be a very effective mechanism for focusing on the important dependencies.

3. ANDERSEN'S POINTS-TO ANALYSIS

We review Andersen's points-to analysis and introduce some definitions used in the rest of the paper. In the literature, there are two core approaches to points-to analysis, ignoring context-sensitivity and flow-sensitivity. The first approach is

```
1.  short target;
2.  struct S { short x; short y; };
3.  short u, *v, w;
4.  struct S s, t;
5.  v = &w;
6.  u = target;
7.  *v = u;
8.  s.x = w;
```

`w/short` <eg1.c:3> → `u/short` <eg1.c:7> → `target/short` <eg1.c:6> `where` `target/short` <eg1.c:1>

`target/short` <eg1.c:1>

`u/short` <eg1.c:3> → `target/short` <eg1.c:6> `where` `target/short` <eg1.c:1>

`S.x/short` <eg1.c:2> → `w/short` <eg1.c:8> → `u/short` <eg1.c:7> → `target/short` <eg1.c:6> ···

Figure 1: A program fragment involving structs and its dependence results (the target is `target`).

unification-based [24]: an assignment such as `x = y` invokes a unification of the node for `x` and the node for `y` in the points-to graph. The algorithms for the unification-based approach typically involve union/find and have essentially linear-time complexity. The second approach is based on subset relationships: an assignment such as `x = y` gives rise to a subset constraint `x` $\supseteq$ `y` between the nodes `x` and `y` in the points-to graph [4]. The algorithms for the subset-based approach utilize some form of subtyping system, subset constraints or a form of dynamic transitive closure, and have cubic-time complexity.

The unification-based approach is faster and less accurate [22]. There has been considerable work on improving the performance of the subset-based approach [11, 23, 21], although the performance gap is still sizable (c.f. [23, 21] and [8]). As Das very recently observed "In spite of these efforts, Andersen's algorithm does not yet scale to programs beyond 500KLOC." [8] There has also been work on improving the accuracy of the unification-based approach by incorporating some of the directional features of the subset-based approach to produce a hybrid unification-based algorithm [8]: for a small increase in analysis time (and quadratic worst-case complexity), much of the additional accuracy of the subset-based approach can be recovered.

A Deductive Reachability Formulation

We use a context-insensitive, flow-insensitive version of the subset-based approach that is essentially the analysis due to Andersen [4]. One reason for this choice is the better accuracy of the subset-based approach over the unification-based approach. Another reason is that users of our dependence analysis system must be able to inspect the dependence chains produced by our system (Section 2), and understand why they were produced. Subset-based approaches generate easier to understand results; unification-based approaches often introduce hard to understand "backwards" flows of information due to the use of equalities.

Previous presentations of Andersen's algorithm have used some form of non-standard type system. Our presentation uses a simple deductive reachability system. This style of analyses presentation was developed by McAllester [20]. It has also been used to describe control-flow analysis [18]. To simplify our presentation, we consider a tiny language consisting of just the operations $\star$ and &. Expressions e have

$$\frac{x \longrightarrow \&y}{y \longrightarrow e} \quad (\text{if } \star x = e \text{ in } P) \qquad (\textsc{star-1})$$

$$\frac{x \longrightarrow \&y}{e \longrightarrow y} \quad (\text{if } e = \star x \text{ in } P) \qquad (\textsc{star-2})$$

$$\frac{}{e_1 \longrightarrow e_2} \quad (\text{if } e_1 = e_2 \text{ in } P) \qquad (\textsc{assign})$$

$$\frac{e_1 \longrightarrow e_2 \qquad e_2 \longrightarrow e_3}{e_1 \longrightarrow e_3} \qquad (\textsc{trans})$$

Figure 2: Deduction rules for aliasing analysis.

the form:

$$e \;::=\; x \;\mid\; \star x \;\mid\; \&x$$

We shall assume that nested uses of $\star$ and & are removed by a preprocessing phase. Programs are sequences of assignments of the form $e_1 = e_2$ where e_1 cannot be $\&x$.

Given some program P, we construct deduction rules as specified in Figure 2. In the first rule, the side condition "if $\star x = e$ in P" indicates that there is an instance of this rule for each occurrence of an assignment of the form $\star x = e$ in P. The side conditions in the other rules are similarly interpreted. Intuitively, an edge $e_1 \longrightarrow e_2$ indicates that any object pointer that we can derive from e_2 is also derivable from e_1. The first rule deals with expressions of the form $\star x$ on the left-hand-sides of assignments: it states that if there is a transition from x to $\&y$, then add a transition from y to e, where e is the left-hand-side of the assignment. The second rule deals with expressions of the form $\star x$ on the right-hand-sides of assignments: it states that if there is a transition from x to $\&y$, then add a transition from e to y where e is the right-hand-side of the assignment. The third rule adds a transition from e_1 to e_2 for all assignments $e_1 = e_2$ in the program, and finally, the fourth rule is just transitive closure. The core of our points-to analysis can now be stated as follows: x can point to y if we can derive $x \longrightarrow \&y$. Figure 3 contains an example program and shows how $y \rightarrow \&x$ can be derived.

Analysis of Full C

Extending this core analysis to full C presents a number of choices. Adding values such as integers is straightforward. It is also easy to deal with nested uses of $\star$ and & through the addition of new temporary variables (we remark that

```
int x, *y;          z → &y     (ASSIGN)
int **z;            *z → &x    (ASSIGN)
z = &y;             y → &x     (from STAR-1)
*z = &x;
```

Figure 3: Example program and application of deduction rules to show y → &x.

considerable implementation effort is required to avoid introducing *too many* temporary variables). However, treating `structs` and `unions` is more complex. One possibility is, in effect, to ignore them: each declaration of a variable of `struct` or `union` type is treated as an unstructured memory location and any assignment to a field is viewed as an assignment to the entire chunk e.g. $x.f$ is viewed as an assignment to x and the field component f is ignored. We call this the *field-independent* approach and examples include [10, 11, 22]. Another approach is to use a field-based treatment of structs such as that taken by Andersen [4]. In essence, the field-based approach collects information with each field of each struct, and so an assignment to $x.f$ is viewed as an assignment to f and the base object x is ignored. (Note that two fields of different structs that happen to have the same name are treated as separate entities.) The following code illustrates the distinction between field-based and field-independent.

```
struct S { int *x; int *y; } A, B;
int z;
main () {
 int *p, *q, *r, *s;
 A.x = &z;  /* field-based: assigns to "x"
             * field-independent: assigns to "A" */
 p = A.x;   /* p gets &z in both approaches */
 q = A.y;   /* field-independent: q gets &z */
 r = B.x;   /* field-based: r gets &z */
 s = B.y;   /* in neither approach does s get &z */
}
```

In the field-independent approach, the analysis determines that only `p` and `q` can point to $\&z$. In the field-based approach, only `p` and `r` can point to $\&z$. Hence, neither of these approaches strictly dominates the other in terms of accuracy. We note that while the works [10, 11, 22] are based on Andersen's algorithm [4], they in fact differ in their treatment of structs: they are field-independent whereas Andersen's algorithm is field-based[1]. In Section 6, we show this choice has significant implications in practice, especially for large code bases. Our aliasing analysis uses the field-based approach, in large part because our dependence analysis is also field-based.

4. COMPILE-LINK-ANALYZE

A fundamental problem in program analysis is modularity: how do we analyze large code bases consisting of many source files? The simple approach of concatenating all of the source files into one file does not scale beyond a few thousand lines of code. Moreover, if we are to build interactive tools based on an analysis, then it is important to avoid reparsing/reprocessing the entire code base when changes are made to one or two files.

The most basic approach to this problem is to parse compilation units down to an intermediate representation, and then defer analysis to a hybrid link-analyze phase. For example, at the highest level of optimization, DEC's MIPS compiler treats the internal ucode files produced by the frontend as "object files", and then invokes a hybrid linker (uld) on the ucode files [9]. The uld "linker" simply concatenates the ucode files together into a single big ucode file and then performs analysis, optimization and code generation on this file. The advantage of this approach is it modularizes the parsing problem – we don't have to parse the entire program as one unit. Also, we can avoid re-parsing of the entire code base if one source file changes. However, it does not modularize the analysis problem – the analysis proceeds as if presented with the whole program in one file.

One common way to modularize the analysis problem is to analyze program components (at the level of functions or source files), and compute summary information that captures the results of these local analyses. Such summaries are then combined/linked together in a subsequent "global-analysis" phase to generate results for the entire program. This idea is analogous to the construction of principle types in type inference systems. For example, assigning "$\alpha \to \alpha$" to the identity function in a simply typed language is essentially a way of analyzing the identity function in a modular way. Uses of the identity function in other code fragments can utilize $\alpha \to \alpha$ as a summary of the behavior of the identity function, thus avoiding inspection of the original function. (Of course, full polymorphic typing goes well beyond simply analyzing code in a modular way, since it allows different type instantiations for different uses of a function – akin to context-sensitive analysis – which is beyond the scope of the present discussion.)

This modular approach to analysis has a long history. According to folklore, one version of the MIPS compiler employed local analysis of separate files and then combined the local analysis results during a "linking" phase. The idea is also implicit in Aiken et. al.'s set-constraint type systems [3], and is much more explicit in Flanagan and Felleisen's componential analysis for set-based analysis [12]. Recently, the idea has also been applied to points-to analysis. Das [8] describes a hybrid unification-based points-to analysis with the following steps. First, each source file is parsed, and the assignment statements therein are used to construct a points-to graph with flow edges, which is simplified using a propagation step. The points-to graph so computed is then "serialized" and written to disk, along with a table that associates symbols and functions with nodes in the graph. The second phase reads in all of these (object) files, unifies nodes corresponding to the same symbol or function from different object files, and reapplies the propagation step to obtain global points-to information. In other words, the analysis algorithm is first applied to individual files and the internal state of the algorithm (which in this case is a points-to graph, and symbol information) is frozen and written to a file. Then, all of these files are thawed, linked and the algorithm re-applied.

[1]Strictly speaking, while Andersen's core algorithm is field-based, he assumes that a pre-processing phase has duplicated and renamed struct definitions so that structs whose values cannot flow together have distinct names (see Section 2.3.3 and 4.3.1 of [4]).

This means that the object files used are specific not just to a particular class of analysis (points-to analysis), but to a particular analysis algorithm (hybrid unification-based analysis), and arguably even to a particular implementation of that algorithm. The object files are designed with specific knowledge of the internal data-structures of an implementation in such a way that the object file captures sufficient information about the internal state of the implementation that this state can be reconstructed at a later stage.

The CLA Model

Our approach, which we call compile-link-analyze (CLA), also consists of a local computation, a linking and a global analysis phase. However, it represents a different set of tradeoffs from previous approaches, and redraws the boundaries of what kind of work is done in each phase. A key difference is that the first phase simply parses source files and extracts assignment statements – no actual analysis is performed – and the linking phase just links together the assignment statements. One advantage of the CLA architecture is that the first two phases remains unchanged for many different implementations of points-to analysis and even different kinds of analysis (we return to this point later). A follow-on advantage is that we can justify investing resources into optimizing the representation of the collections of assignments, because we can reuse this work in a number of different analysis implementations. In particular, we have developed a database-inspired representation of assignments and function definitions/calls/returns. This representation is compact and heavily indexed. The indexing allows relevant assignments for a specific variable to be identified in just one lookup step, and more generally, it supports a mode where the assignments needed to solve a particular analysis problem can be dynamically loaded from the database on demand.

More concretely, CLA consists of three phases. The *compile* phase parses source files, extracts assignments and function calls/returns/definitions (in what follows we just call these "assignments"), and writes an object file that is basically an indexed database structure of these basic program components. No analysis is performed yet. Complex assignments are broken down into primitive ones by introducing temporary variables. The elements of the database, which we call primitive assignments, involve variables and (typically) at most one operation.

The *link* phase merges all of the database files into one database, using the linking information present in the object files to link global symbols (the same global symbol may be referenced in many files). During this process we must recompute indexing information. The "executable" file produced has the same format as the object files, although its linking information is typically obsolete (and could be stripped).

The *analyze* phase performs the actual analysis: the linked object file is dynamically loaded on demand into the running analysis. Importantly, only those parts of the object file that are required are loaded. An additional benefit of the indexing structure of the object file is that when we have read information from the object file we can simply discard it and re-load it later if necessary (memory-mapped I/O is used to support efficient reading and re-reading of the object file). We use this feature in our implementation of Andersen's analysis to greatly reduce the memory footprint of the analysis. It allows us to maintain only a very small portion of the object file in memory.

An example source file and a partial sketch of its object file representation is given in Figure 4. These object files consist of a header section which provides an index to the remaining sections, followed by sections containing linking information, primitive assignments (including information about function calls/returns/definitions) and string information, as well as indexing information for identifying targets for the dependence analysis. The primitive assignments are contained in the *dynamic section*; it consists of a list of blocks, one for each object in the source program. Each block consists of information about the object (its name, type, source code location and other attributes), followed by a list of primitive assignments where this object is the source. For example, the block for `z` contains two primitive assignments, corresponding to the second and third assignments in the program (a very rough intuition is that whenever `z` changes, the primitive assignments in the block for `z` tell us what we must recompute).

As mentioned before, one of the goals of our work is to build infrastructure that can be used for a variety of different analysis implementations as well as different kinds of analysis. We have used our CLA infrastructure for a number of different subset-based points-to analysis implementations (including an implementation based on bit-vectors, as well as many variations of the graph-based points-to algorithm described later in this paper), and field-independent and field-based points-to analysis. The key point is that our object files do not depend on the internals of our implementation and so we can freely change the implementation details without changing the object file format. We have also used CLA infrastructure for implementing unification-based points-to analysis, and for the dependence analysis described in Section 2. Finally, we note that we can write pre-analysis optimizers as database to database transformers. In fact, we have experimented with context-sensitive analysis by writing a transformation that reads in databases and simulates context-sensitivity by controlled duplication of primitive assignments in the database – this requires no changes to code in the compile, link or analyze components of our system.

We now briefly sketch how the dependence and points-to analyses use object files. Returning to Figure 4, consider performing points-to analysis. The starting point for points-to analysis is primitive assignments such as `q = &y` in the static section. Such an assignment says that `y` should be added to the points-to set for `q`. This means that the points-to set for `q` is now non-empty, and so we must load all primitive assignments where `q` is the source. In this case, we load `p = q`, which imposes the constraint $\mathtt{p} \supseteq \mathtt{q}$. This is all we need to load for points-to analysis in this case. Now consider a dependence analysis. Suppose that the target of the dependence analysis is the variable `z`. We first look up "z" in the hashtable in the target section to find all variables in the object file whose name is "z" (strictly speaking, we find the object file offsets of all such variables). In this case we find just one variable. We build a data-structure to file a.c:

```
int x, y, z, *p, *q;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
```

header section: segment offsets and sizes
global section: linking information
static section: address-of operations; always loaded for points-to analysis `q = &y`
string section: common strings
target section: hashtable for finding targets
dynamic section: elements are loaded on demand, organized by object x @ a.c:1 none y @ a.c:1 x = y @ a.c:2 z @ a.c:1 x = z @ a.c:3 *p =z @ a.c:4 p @ a.c:1 x = *p @ a.c:7 q @ a.c:1 p = q @ a.c:5

Figure 4: Example program and sketch of its object file

say that this variable is a target of the dependence analysis. We then load the block for `z`, which contains the primitive assignments `x = z` and `*p = z`. Using the first assignment, we build a data-structure for `x` and then we load the block for `x`, which is empty. Using the second assignment, we find from the points-to analysis that `p` can point to `&y`, and so we build a data-structure for `y` and load the block for `y`, etc. In the end, we find that both `x` and `y` depend on `z`.

The compilation phase we have implemented includes more information in object files that we have sketched here. Our object files record information about the strength of dependencies (see Section 2), and also information about any operations involved in assignments. For example, corresponding to a program assignment `x = y + z`, we obtain two primitive assignments `x = y` and `x = z` in the database. Each would retain information about the "+" operation. Such information is critical for printing out informative dependence chains; it is also useful for other kinds of analysis that need to know about the underlying operations. We include sections that record information about constants in the program. To support advanced searches and experiment with context-sensitive analysis, we also include information for each local variable that identifies the function in which it is defined. We conjecture that our object file format can be used (or easily adapted) for any flow-insensitive analysis that computes properties about the values of variables i.e. any analysis that focuses entirely on the assignments of the program, and ignores control constructs. Examples include points-to analysis, dependence analysis, constant propagation, binding-time analysis and many variations of set-based analysis. One advantage of organizing object files using sections (much like COFF/ELF), is that new sections can be transparently added to object files in such a way that existing analysis systems do not need to be rewritten.

We conclude with a discussion of functions and function pointers. Function are handled by introducing standardized names for function arguments and returns. For example, corresponding to a function definition `int f(x, y) { ... return(z)}`, we generate primitive assignments $x = f_1$, $y = f_2$, $f_{ret} = z$, where f_1, f_2, f_{ret} are respectively the standardized variables for the two arguments of f and f's return value. Similarly, corresponding to a call of the form $w = f(e_1, e_2)$, we generate primitive assignments $f_1 = e_1$, $f_2 = e_2$ and $w = f_{ret}$. These standardized names are treated as global objects, and are linked together, like other global objects, by the linker. The treatment of indirect function calls uses the same naming convention, however some of the linking of formal and actual parameters happens at analysis time. Specifically, corresponding to a function definition for g, there is an object file entry (in the block for g) that records the argument and return variables for g. Corresponding to an indirect call `(*f)(x, y)`, we mark f as a function pointer as well as adding the primitive assignments $f_1 = x$, $f_2 = y$, etc. During analysis, if a function g is added to the points-to set for f (marked as a function pointer), then we load the record of argument and return variables for both f and g. Using this information, we add new assignments $g_1 = f_1$, $g_2 = f_2$ and $f_{ret} = g_{ret}$.

5. A GRAPH-BASED ALGORITHM FOR ANDERSEN'S ANALYSIS

Scalability of Andersen's context-insensitive flow-insensitive points-to analysis has been a subject of much research over the last five years. One problem with Andersen's analysis is the "join-point" effect of context-insensitive flow-insensitive analysis: results from different execution paths can be joined together and distributed to the points-to sets of many variables. As a result, the points-to sets computed by the analysis can be of size $O(n)$ where n is the size of the program; such growth is commonly encountered in large benchmarks. This can spell scalability disaster if all points-to sets are explicitly enumerated.

Aiken et. al. have addressed a variety of scaling issues for Andersen's analysis in a series of papers. Their work has included techniques for elimination of cycles in the inclusion graph [11], and projection merging to reduce redundancies in the inclusion graph [23]. All of these are in the context of a transitive-closure based algorithm, and their results show very substantial improvements over their base algorithm – with all optimizations enabled, they report analysis times of 1500s for the gimp benchmark on a SPARC Enterprise 5000 with 2GB [23].

Alternatively, context- and flow-sensitivity can be used to reduce the effect of join-points. However the cost of these additional mechanisms can be large, and without other breakthroughs, they are unlikely to scale to millions of lines of code. Also, recent results suggest that this approach may be of little benefit for Andersen's analysis [13].

In principle, ideas from sub-transitive control-flow analysis [18] could also be applied to avoid propagation of the information from join-points. The basic idea of sub-transitive control-flow analysis is that the usual dynamic transitive closure formulation of control-flow analysis is redesigned so that the dynamic edge-adding rules are de-coupled from the transitive closure rules. This approach can lead to linear-time algorithms. However, it is currently only effective on bounded-type programs, an unreasonable restriction for C.

The main focus of our algorithm, much like that of the sub-transitive approach, is on finding a way to avoid the cost of computing the full transitive closure of the inclusion graph. We begin by classifying the assignments used in the deductive reachability system given in Section 3 into three classes: (a) simple assignments, which have the form $x = y$, (b) base assignments, which have the form $x = \&y$, and (c) complex assignments, which have the form $x = \star y$ or $\star x = y$. For simplity, we omit treatment of $\star x = \star y$; it can be split into $\star x = z$ and $z = \star y$ where z is a new variable. In what follows we refer to items of the form $\&y$ as lvals.

The central data-structure of our algorithm is a graph $\mathcal{G}$, which initially contains all information about the simple assignments and base assignments in the program. The nodes of $\mathcal{G}$ are constructed as follows: for each variable x in the program, we introduce nodes n_x and $n_{\star x}$ (strictly speaking, we only need a node $n_{\star x}$ if there is a complex assignment of the form $y = \star x$). The initial edges of $\mathcal{G}$ are constructed as follows: corresponding to each simple assignment $x = y$, there is an edge $n_x \rightarrow n_y$ from n_x to n_y. Corresponding to every node n in $\mathcal{G}$, there is a set of base elements, defined as follows:

$$\texttt{baseElements}(n_x) = \{y : x = \&y \text{ appears in } P\}$$

The complex assignments, which are not represented in $\mathcal{G}$, are collected into a set $\mathcal{C}$. The algorithm proceeds by iterating through the complex assignments in $\mathcal{C}$ and adding edges to $\mathcal{G}$ based on the information currently in $\mathcal{G}$. At any point in the algorithm, $\mathcal{G}$ represents what we explicitly know about the sets of lvals for each program variable. A major departure from previous work is that $\mathcal{G}$ is maintained in *pre-transitive form* i.e. we do not transitively close the graph. As a result, whenever we need to determine the current lvals of a specific variable, we must perform graph reachability: to find the set of lvals for variable x, we find the set of nodes reachable from n_x in zero or more steps, and compute the union of the `baseElements` sets for all of these nodes. We use the function $\texttt{getLvals}(n_x)$ to denote this graph reachability computation for node n_x.

The process of iterating through the complex assignments in $\mathcal{C}$ and adding edges to $\mathcal{G}$ based on the information currently in $\mathcal{G}$ is detailed in Figure 5. Note that line 7 need only be executed once, rather than once for each iteration of the loop.

Before discussing the `getLvals()` function, we give some intuition on the computational tradeoffs involved in maintaining the constraint graph in pre-transitive form and computing lvals on demand. First, during its execution, the algorithm only requires the computation of lvals for some subset of the nodes in the graph. Now, of course, at the end of the algorithm, we may still have to compute all lvals for all graph nodes. However, in the presence of cycle-elimination (discussed shortly), it is typically much cheaper to compute all lvals for all nodes when the algorithm terminates than it is to do so during execution of the algorithm. Second, the pre-transitive algorithm trades off traversal of edges versus flow of lvals along edges. More concretely, consider a complex assignment such as $\star x = y$, and suppose that the set of lvals for x includes $\&x_1$ and $\&x_2$. As a result of this complex assignment, we add edges from x_1 to y and x_2 to y. Now, in an algorithm based on transitive-closure, all lvals associated with y will flow back along the new edges inserted and from there back along any paths that end with x_1 or x_2. In the pre-transitive graph, the edges are added and there is no flow of lvals. Instead, lvals are collected (when needed) by a traversal of edges. In the transitive closure case, there are $O(n.E)$ transitive closure steps, where n is the average number of distinct lvals that flow along an edge, and E is the number of edges, versus $O(E)$ steps per reachability computation This tradeoff favors the pre-transitive graph approach when E is small and n is large. (We remark that this is analysis is for intuition only; it is not a formal analysis, since neither the transitive closure step nor the reachability step are $O(1)$ operations.)

We next describe `getLvals()`, which is the graph reachability component of the algorithm. A key part of the graph reachability algorithm is the treatment of cycles. Not only is cycle detection important for termination, but it has fundamental performance implications. The first argument of `getLvals()` is a graph node and the second is a list of elements that define the path we are currently exploring; top-level calls have the form $\texttt{getLvals}(n, nil)$. Each node in $\mathcal{G}$ has a one-bit field `onPath`.

The function `unifyNodes()` merges two nodes. We implement node merging by introducing an optional `skip` field for each node. Two nodes n_1 and n_2 are then unified by setting $\texttt{skip}(n_1)$ to n_2, and merging edge and `baseElement` information from n_1 into n_2. Subsequently, whenever node n_1 is accessed, we follow its skip pointer. We use an incremental algorithm for updating graph edges to skip-nodes to their de-skipped counter-parts.

Cycle elimination was first used for points-to analysis by Fähndrich et. al [11]. In their work, the cost of finding cycles was non-trivial and so completeness of cycle detection was sacrificed in order to contain its cost. In contrast, cycle detection is essentially free in our setting during graph reachability computations. Moreover, we find almost all cycles – more precisely, we find all cycles in the parts of the graphs we traverse during graph reachability. In essence, we find the costly cycles – those that are not detected are in parts of the graph that we ignore. In other words, one of the benefits of our algorithm is that it finds more of the important cycles and it does so more cheaply.

```
    /* The Iteration Algorithm */
1.  do {
2.    nochange = true;
3.    for each complex assignment *x = y in  C
4.      for each &z in getLvals(n_x)
5.        add an edge n_z -> n_y to  G;
6.    for each complex assignment x = *y in  C
7.      add an edge n_x -> n_*y;
8.      for each &z in getLvals(n_y)
9.        add an edge n_*y -> n_z
10. } until nochange
```

```
1.   getLvals(n, path) {
2.     if(onPath(n)) { /* we have a cycle */
3.       foreach n' in path, unifyNode(n', n);
4.       return(emptySet);
5.     } else { /* explore new node n */
6.       onPath(n) = 1;
7.       lvals = emptySet;
8.       path = cons(n, path);
10.        lvals = union(lvals, baseElements(n);
11.      foreach n' such that there is an edge from n to n'
12.        lvals = union(lvals, getLvals(n, path));
13.      onPath(n) = 0;
14.      return(lvals);
15.  }
```

Figure 5: The Pre-Transitive Graph Algorithm for Points-to Analysis

This completes the basic description of our algorithm. We conclude with a number of enhancements to this basic algorithm. First, and most important, is a caching of reachability computations. Each call to *getLvals()* first checks to see if the lvals have been computed for the node during the current iteration of the iteration algorithm; if so, then the previous lvals are returned, and if not, then they are recomputed (and stored in the node). Note that this means we might use "stale" information; however if the information is indeed stale, the `nochange` flag in the iteration algorithm will ensure we compute another iteration of the algorithm using fresh information. Second, the graph edges are maintained in both a hash table and a per-node list so that it is fast to determine whether an edge has been previously added and also to iterate through all of the outgoing edges from a node. Third, since many lval sets are identical, a mechanism is implemented to share common lvals set. Such sets are implemented as ordered lists, and are linked into a hash table, based on set size. When a new lval set is created, we check to see if it has been previously created. This table is flushed at the beginning of each pass through the complex assignments. Fourth, lines 4-5 and lines 8-9 of Figure 5 are changed so that instead of iterating over all lvals in `getLvals(`n_y`)`, we iterate over all nodes in `getLvalsNodes(`n_y`)`. Conceptually, the function `getLvalsNodes()` returns all of the de-skipped nodes corresponding to the lvals computed by `getLvals()`; however it can be implemented more efficiently.

From the viewpoint of performance, the two most significant elements of our algorithm are cycle elimination and the caching of reachability computations. We have observed a slow down by a factor in excess of $> 50K$ for *gimp* (45,000s c.f. 0.8s user time) when both of these components of the algorithm are turned off.

6. RESULTS

Our analysis system is implemented using a mix of ML and C. The compile phase is implemented in ML using the ckit frontend[6]. The linker and the analyzer are implemented in C. Our implementation deals with full C including structs, unions, arrays and function call/return (including indirect calls). Support for many of these features is based on simple syntactic transformations in the compile phase. The field-based treatment of structs is implemented as follows: we generate a new variable for each field f of a struct definition, and then map each access of that field to the variable. Our treatment of arrays is index-independent (we essentially ignore the index component of sub expressions). The benchmarks we use are described in Table 2. The first six benchmarks were obtained from the authors of [21], and the lines of code reported for these are the number of lines of non-blank, non-# lines in each case. We do not currently have accurate source line counts for these benchmarks. The seventh benchmark was obtained from the authors of [23]. The last benchmark is the Lucent code base that is the main target of our system (for proprietary reasons, we have not included all informations on this benchmark). For each benchmark, we also measure the size of the preprocessed code in bytes, the size of the object files produced by the analysis (compiler + linker, also in bytes), and the number of primitive assignments in the object files – the five kinds of assignments allowed in our intermediate language.

We remark that line counts are only a very rough guide to program size. Source code is misleading for many reasons. For instance, macro expansion can considerably increase the amount of work that must be performed by an analysis. Preprocessed code is misleading because many extraneous extern declarations are included as the result of generic system include files. Moreover, these system include files can vary considerably in size from system to system. AST node counts of preprocessed code are a better measure of complexity because they de-emphasize the effects of coding style; however there is no agreed upon notion of AST nodes, and AST nodes might still be inflated by unnecessary declarations generated by rampant include files. Counts of primitive assignments may be a more robust measure.

Results from these benchmarks are included in Table 3. These results measure analysis where (a) each static occurrence of a memory allocation primitive (`malloc`, `calloc`, etc.) is treated as a fresh location, and (b) we ignore constant strings. This is the default setup we use for points-to and dependence analysis. The first column of Table 3 represents the count of program objects (variables and fields) for which we compute non-empty pointer sets; it does not include any temporary variables introduced by the analysis. The second column represents the total sizes of the points-to sets for all program objects. The third and fourth columns give wall-clock time and user time in seconds respectively, as reported by `/bin/time` using a single processor of a two processor Pentium 800MHz machine with 2GB of memory running Linux[2]. The fifth column represents space utiliza-

[2]Red Hat Linux release 6.2 (Piglet)
VA Linux release 6.2.3 07/28/00 b1.1 P2
Kernel 2.2.14-VA.5.1smp on a 2-processor i686.

	LOC (source)	LOC (preproc.)	preproc. size	object size	program variables	assignments				
						$x = y$	$x = \&y$	$*x = y$	$*x = *y$	$x = *y$
nethack	-	44.1K	1.4MB	0.7MB	3856	9118	1115	30	34	105
burlap	-	74.6K	2.4MB	1.4MB	6859	14202	1049	1160	714	1897
vortex	-	170.3K	7.7MB	2.6MB	11395	24218	7458	353	231	1866
emacs	-	93.5K	40.2MB	2.6MB	12587	31345	3461	614	154	1029
povray	-	175.5K	68.1MB	3.1MB	12570	29565	4009	2431	1190	3085
gcc	-	199.8K	69.0MB	4.4MB	18749	62556	3434	1673	585	1467
gimp	440K	7486.7K	201.6MB	27.2MB	131552	303810	25578	5943	2397	6428
lucent	1.3M	-	-	20.1MB	96509	270148	72355	1562	991	3989

Table 2: Benchmarks

tion in MB, obtained by summing static data and text sizes (reported by `/bin/size`), and dynamic allocation (as reported by `malloc_stats()`). We note that for the lucent benchmark – the target code base of our system – we see total wall-clock times of about half a second, and space utilization of under 10MB.

The last three columns explain why the space utilizations are so low: these columns respectively show the number of primitive assignments maintained "in-core", the number loaded during the analysis, and the total number of primitive assignments in the object file. Note that only primitive assignments relevant to aliasing analysis are loaded (e.g. non-pointer arithmetic assignments are usually ignored). Recall that once we have loaded a primitive assignment from an object file and used it, we can discard it, or keep it in memory for future use. Our discard strategy is: discard assignments $x = y$ and $x = \&y$, but maintain all others. These numbers demonstrate the effectiveness of the load-on-demand and load-and-throw-away strategies supported by the CLA architecture.

Table 4 studies the effect of changing the baseline system. The first group of columns represents the baseline and is just a repeat of information from Table 3. The second group shows the effect of changing the underlying treatment of structs from field-based to field-independent. We caution that these results are very preliminary, and should not be interpreted as a conclusive comparison of field-based and field-independent. In particular, there are a number of opportunities of optimization that appear to be especially important in the field-independent case that have not implemented in our current system. We expect that these optimizations could significantly close the time and space gap between the two approaches. However, it is clear that the choice between field-based and field-independent has significant implications in practice. Most points-to systems in the literature use the field-independent approach. Our results suggest that the field-based might in fact represent a better tradeoff. The question of the relative accuracy of the two approaches is open – even the metric for measuring their relative accuracy is open to debate.

We conclude by briefly discussing empirical results from related systems in the literature. Since early implementations of Andersen's analysis [22], much progress has been made [11, 23, 21]. Currently, the best results for Andersen's are analysis times of about 430 seconds for about 500K lines of code (using a single 195 MHz processor on a multi-processor SGI Origin machine with 1.5GB) [21]. The main limiting factor in these results is that space utilization (as measured by the amount of live data after GC) is 150MB and up – in fact the largest benchmark in [21] ran out of memory. Results from [23] report analysis times of 1500s for gimp (on a SPARC Enterprise 5000 with 2GB). We note that both of these implementations of Andersen's analysis employ a field-independent treatment of structs, and so these results are not directly comparable to ours (see the caveats above about the preliminary nature of results in Table 4).

Implementations of Steensgaard's algorithm are faster and use less memory. Das reports that Word97 (about 2.2 million lines of code) runs in about 60s on a 450MHz Intel Xeon running Windows NT [8]. Das also reports that modifications to Steensgaard's algorithm to improve accuracy yield analysis times of about 130s, and memory usage of "less than 200MB" for the same benchmark. We again note that Das uses a field-independent treatment of structs.

7. CONCLUSION

We have introduced CLA, a database-centric analysis architecture, and described how we have utilized it to implement a variety of high-performance analysis systems for points-to analysis and dependence analysis. Central to the performance of these systems are CLA's indexing schemes and support for demand-driven loading of database components. We have also described a new algorithm for implementing dynamic transitive closure that is based on maintaining a pre-transitive graph, and computing reachability on demand using caching and cycle elimination techniques.

The original motivation for this work was dependence analysis to help identify potential narrowing bugs that may be introduced during type modifications to large legacy C code bases. The points-to analysis system described in this paper has been built into a forward data-dependence analysis tool that is deployed within Lucent. Our system has uncovered many serious new errors not found by code inspections and other tools.

Future work includes exploration of context-sensitivity, and a more accurate treatment of structs that goes beyond field-based and field-independent (e.g. modeling of the layout of C structs in memory[7], so that an expression $x.f$ is treated as an offset "f" from some base object x)

Acknowledgements: Thanks to Satish Chandra and Jeff Foster for access to their respective systems and bench-

	pointer variables	points-to relations	real time	user time	process size	assignments in core	loaded	in file
nethack	1018	7K	0.03s	0.01s	5.2MB	114	5933	10402
burlap	3332	201K	0.08s	0.03s	5.4MB	3201	12907	19022
vortex	4359	392K	0.15s	0.11s	5.7MB	1792	15411	34126
emacs	8246	11232K	0.54s	0.51s	6.0MB	1560	28445	36603
povray	6126	141K	0.11s	0.09s	5.7MB	5886	27566	40280
gcc	11289	123K	0.20s	0.17s	6.0MB	2732	53805	69715
gimp	45091	15298K	1.05s	1.00s	12.1MB	8377	144534	344156
lucent	22360	3865K	0.46s	0.38s	8.8MB	4281	101856	349045

Table 3: Results

	field-based				field-independent (preliminary)			
	pointers	relations	utime	size	pointers	relations	utime	size
nethack	1018	7K	0.01s	5.2MB	1714	97K	0.03s	5.2MB
burlap	3332	201K	0.03s	5.4MB	2903	323K	0.21s	5.9MB
vortex	4359	392K	0.11s	5.7MB	4655	164K	0.09s	5.7MB
emacs	8246	11232K	0.51s	6.0MB	8314	14643K	1.05s	6.7MB
povray	6126	141K	0.09s	5.7MB	5759	1375K	0.39s	6.6MB
gcc	11289	123K	0.17s	6.0MB	10984	408K	0.65s	8.8MB
gimp	45091	15298K	1.00s	12.1MB	39888	79603K	30.12s	18.1MB
lucent	22360	3865K	0.46s	8.8MB	26085	19665K	137.20s	59.0MB

Table 4: Effect of a field-independent treatment of structs.

marks.

8. REFERENCES

[1] A. Aiken, M. Fähndrich, J. Foster, and Z. Su, "A Toolkit for Constructing Type- and Constraint-Based Program Analyses", *TIC'98*.

[2] A. Aiken and E. Wimmers, "Solving Systems of Set Constraints", *LICS*, 1992.

[3] A. Aiken and E. Wimmers, "Type Inclusion Constraints and Type Inference", *ICFP*, 1993.

[4] L. Andersen, "Program Analysis and Specialization for the C Programming Language", PhD. thesis, DIKU report 94/19, 1994,

[5] D. Atkinson and W. Griswold, "Effective Whole-Program Analysis in the Presence of Pointers", 1998 Symp. on the Foundations of Soft. Eng..

[6] S. Chandra, N. Heintze, D. MacQueen, D. Oliva and M. Siff, "ckit: an extensible C frontend in ML", to be released as an SML/NJ library.

[7] S. Chandra and T. Reps, "Physical Type Checking for C" *PASTE*, 1999.

[8] M. Das, "Unification-Based Pointer Analysis with Directional Assignments" *PLDI*, 2000.

[9] "Appendix D: Optimizing Techniques (MIPS-Based C Compiler)", *Programmer's Guide: Digital UNIX Version 4.0*, Digital Equipment Corporation, March 1996.

[10] J. Foster, M. Fähndrich and A. Aiken, "Flow-Insensitive Points-to Analysis with Term and Set Constraints" U. of California, Berkeley, UCB//CSD97964, 1997.

[11] M. Fähndrich, J. Foster, Z. Su and A. Aiken, "Partial Online Cycle Elimination in Inclusion Constraint Graphs" *PLDI*, 1998.

[12] C. Flanagan and M. Felleisen, "Componential Set-Based Analysis" *PLDI*, 1997.

[13] J. Foster, M. Fähndrich and A. Aiken, "Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C", *SAS* 2000.

[14] N. Heintze, "Set Based Program Analysis", PhD thesis, Carnegie Mellon University, 1992.

[15] N. Heintze, "Set-Based Analysis of ML Programs", *LFP*, 1994.

[16] N. Heintze, "Analysis of Large Code Bases: The Compile-Link-Analyze Model" unpublished report, November 1999.

[17] N. Heintze and J. Jaffar, "A decision procedure for a class of Herbrand set constraints" *LICS*, 1990.

[18] N. Heintze and D. McAllester, "On the Cubic-Bottleneck of Subtyping and Flow Analysis" *LICS*, 1997.

[19] "Programming Languages - C", *ISO/IEC* 9899:1990, Internation Standard, 1990.

[20] D. McAllester, "On the Complexity Analysis of Static Analysis", *SAS*, 1999.

[21] A. Rountev and S. Chandra, "Off-line Variable Substitution for Scaling Points-to Analysis", *PLDI*, 2000.

[22] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-To Analysis", *POPL*, 1997.

[23] Z. Su, M. Fähndrich, and A. Aiken, "Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs", *POPL*, 2000.

[24] B. Steensgaard, "Points-to Analysis in Almost Linear Time", *POPL*, 1996.

[25] F. Tip, "Generation of Program Analysis Tools", Institute for Logic Language and Computation dissertation series, 1995-5, 1995.