Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Structure aliasing in GCC

Daniel Berlin
IBM T.J. Watson Research Center
dberlin@us.ibm.com

Abstract

GCC has a well known deficiency in the ability to detect and optimize structure accesses (`a.b`) and pointer-to structure accesses (`a->b`). We present a structure overlap determination algorithm and a structure access disambiguation algorithm for the SSA middle-end of GCC. We also present an algorithm for field sensitive points-to analysis.

This paper also describes the changes necessary to make this information available to the tree optimizations.

These algorithms can be applied both intraprocedurally and interprocedurally. They are efficient enough to be applied to whole programs.

With these algorithms implemented, the GCC tree-SSA optimizers automatically take advantage of the new information and produce significantly better code when faced with structure accesses, and pointer to structure accesses, which are heavily used in both C and C++.

1 Introduction

GCC's middle-end is entirely SSA based. Variables that cannot be directly renamed due to aliasing, call clobbering, or being non-scalar variables, have a "virtual" SSA form that represents the aliasing properties. An example is given in Program 1. Virtual SSA form is shown by the V-USE and V-MAY-DEF for non-scalar operations. Like regular SSA form, use-def and def-use chains are created and kept up to date for the virtual form.

Program 1 Program with virtual SSA form for non-scalar accesses. Arrows represent def-use and use-def chains

```
struct foo
{
  int a;
  int b;
} temp;

int main(void)
{
  int t1, t2;

  test_1 = V_MAY_DEF <test_0>
  temp.a = 5;

  test_2 = V_MAY_DEF <test_1>
  temp.b = 6;

  VUSE <test_2>
  t1_1 = temp.a;

  VUSE <test_2>
  t2_2 = temp.b;

  return t1_1 * t2_2;
}
```

Structural aliasing information was not present in the initial implementation of the SSA middle-end. In particular, there are three forms of structural aliasing GCC's middle end currently lacks[1], each demonstrated by a motivating example[2].

1. In Program 1, we'd like the ability to disambiguate between regular references to structure fields, and determine that they do not interfere with each other.

2. In Program 2, we'd like the ability to differentiate between dereferences of structure pointers, so we can

   (a) Determine that the loads and stores of `t2->a` only affects what happens to structure `a` (and in particular, field a of structure `astruct`),

   (b) Determine that the loads and stores of `t2->b` only affect what happens to structure `b` (and in particular, field `b` of structure `bstruct`).

   Note that if Program 2 changes slightly, we may not be able to determine this, for reasons explained later.

3. In Program 3, we'd like the ability to discover what `t3.afield` points to, and what `t3.b` points to, so that we can determine that they do not alias each other, and the loads and stores to each field are independent.

The second two problems are actually part of the same problem (field-sensitive pointer analysis), and will be treated together.

[1] It should be noted that the backend of GCC can often disambiguate these by luck, because they are usually simple base + offset addresses in registers by that time.

[2] These examples are, of course, quite contrived, although the idioms occur often in real programs.

Program 2 Motivating example for the second issue

```
struct foo
{
  int a;
  int b;
};

int main(void)
{
  struct foo astruct, bstruct;
  struct foo *t2;
  int temp1, temp2;
  t2 = &astruct;
  t2->a = 5;
  temp1 = t2->a;
  t2 = &bstruct;
  t2->b = 6;
  temp2 = t2->b;
  return temp1 * temp2;
}
```

2 Disambiguating references to structure fields

2.1 Finding structure overlaps

In order to resolve the issue of structure overlaps, we first must discover where the overlaps are. To discover overlapping fields, we walk the structure type recursively and record the offset and size of each field. All nested structures are effectively inlined into the main structure. In other words, we consider fields of nested structures to be fields of the containing structure, at a different offset.

The basic algorithm for getting the list of fields from the type is given in Algorithm 1 for reference. The check for whether anything was added to fieldlist is used to handle empty structures. The actual GCC implementation uses a

Program 3 Motivating example for the third issue

```
struct foo
{
  int *afield;
  int *bfield;
};

int main(void)
{
  int a, b;
  int temp1, temp2;
  struct foo t3;
  t3.afield = &a;
  t3.bfield = &b;
  a = 5;
  b = 6;
  temp1 = *(t3.afield);
  temp2 = *(t3.bfield);
  return temp1 * temp2;
}
```

stack, and checks whether anything was pushed onto the stack by the recursion.

2.2 Representing structure overlaps

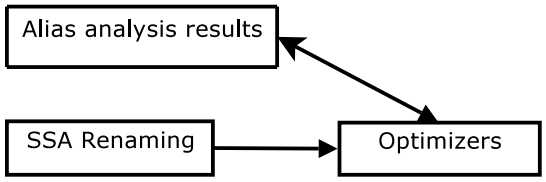

Figure 1: Alias Oracle approach to aliasing

There are two commonly used mechanisms for providing aliasing information to the optimization passes: calling an oracle or encoding the information into the intermediate representation. The oracle method (which looks like Figure 1), used at the RTL level, places the implementation burden on the optimization to ask about the memory references involved in a

Algorithm 1 Algorithm for generating a list of fields into *fieldlist* from a structure type *s*

for all fields in *s* **do**
  if *field* is a structure type **then**
    $temp \leftarrow fieldlist$
    recurse on field
    if $temp = fieldist$ **then**
      $fieldlist \leftarrow fieldlist \cup field$
    end if
  else
    $fieldlist \leftarrow fieldlist \cup field$
  end if
end for

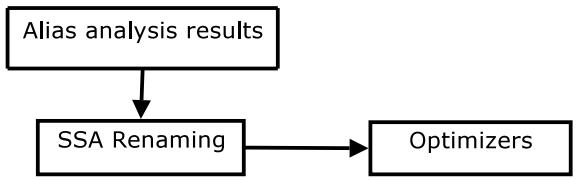

Figure 2: IL Encoding approach to aliasing

transformation before the transformation is applied. The representation method (which looks like Figure 2), places the burden on the analysis phase to represent the output in the encoding used for the intermediate code. With this, one can determine whether two statements may-alias (may access the same memory location) directly from the virtual SSA form versions on each statement.

For low quality alias information, the encoding mechanism works fine because little or no work is required to get the transformation to work in the presence of aliasing information. However, as the aliasing gets more sophisticated, the representation method either becomes bulky, or fails to be able to represent the aliasing information.

For now we have chosen to work within the representation framework but the quality of information available with this analysis begs that this decision be revisited in the near future.

Virtual SSA form makes regular structural aliasing a bit trickier to do than the oracle method. We can't just say "these things have different base + offset, therefore they are different," we must actually encode this into the variables used in the virtual form so that the various use-def and def-use chains are kept up to date as the SSA form is changed and incrementally updated.

GCC currently has a notion of memory tags that represent abstract memory locations, and these are are attached to pointers and aggregates to represent what the variable can alias. These memory tags are, in effect, virtual variables that represent type based aliasing and points-to aliasing results.

In order to represent structure fields, we create a memory tag for each variable that is a structure.

The first field of `foo` at offset 0, with a size of 32 bits becomes variable `SFT.0`. Anywhere then, when scanning the statements, and for aliasing purposes, a component access is considered to must-use/must-def the structure field tags for the fields it would access. For accesses where we cannot determine what structure field is being accessed, we consider the access to may-use/may-def the structure field tags for all the fields of that structure.

After encoding the structure overlap information into the variables, the SSA renamer takes care of linking everything together. All of the SSA optimizations will see that the loads and stores from one field don't interfere with the other. This results in Program 4, given the original input from Program 1.

This encoding has the unfortunate problem that it requires a new variable for each section of a structure field type **for each variable that is a structure**. Ideally, we would want to simply encode the overlap information for each type, and be able to use that directly. Unfortunately, this is not possible with the current virtual SSA representation, because it requires real variables. An example of this is shown in Program 4. For each structure field in `temp`, there is a variable named `SFT`. If another variable of type `struct foo` were created and used, there would be more `SFT` variable for each of its fields as well. This is all because the virtual SSA form depends on accesses to the same location being represented with the same name.

As a result of this, the current scheme for expressing structure overlaps can become expensive in terms of the number of virtual operands, especially when things like call clobbering and pointers to structures, are taken into account. If two structures are call clobbered, each with 10 fields, you end up with 20 virtual operands at every clobber site. In contrast to all of these

Program 4 After structural aliasing

```
struct foo
{
  int a;
  int b;
} temp;

int main(void)
{
  int t1, t2;

  SFT.0_1 = V_MUST_DEF <SFT.0_0>
  temp.a = 5;

  SFT.1_1 = V_MUST_DEF <SFT.1_0>
  temp.b = 6;

  VUSE <SFT.0_1>
  t1_1 = temp.a;

  VUSE <SFT.1_1>
  t2_2 = temp.b;

  return t1_1 * t2_2;
}
```

variables and operands, if we were using the oracle method, the oracle could simply look at a prebuilt map of where the overlaps are for each type, and determine whether two component accesses to the same structure overlapped. The cost of such a map is $O(n)$ space where n is the number of fields in the type. For the represenation method we currently use, the cost is at least $O(ni)$ space, where i is the number of instances of that type. The cost is actually much higher than that, since you have the space of virtual definitions and uses, for each definition, use, and ϕ of a variable representing a field.

To help keep the number of extra virtual operands down, we perform analysis to determine which parts of the structure are actually used in a given function, and only create the virtual variables for those portions. Variables whose only uses are being passed by address to another function have no virtual variables created for them, at least at the moment. This significantly reduces the number of virtual operands that need to be created to represent call clobbering and aliasing.

Early prototypes of this work attempted to introduce partial may/must-def (IE include an offset and size in the operation) and partial use operations into the virtual form, but this did not mesh well with the incremental SSA updating that is performed. In particular is algorithmically more complex to handle, because you end up with reaching definitions and ϕ nodes on a per-offset basis. If you end up encoding that information into the name, you get the result that the standard SSA algorithms will rename it all properly for you.

3 Disambiguating dereferences of structure pointers and structure fields

The problem of determining whether t->a and t->b can alias at all (as opposed to determining what they end up pointing to) can be formulated as a base + offset problem, and solved in the same way as the structure overlaps (by creating virtual variables).

However, determining what the structure pointers actually point to, and the structure fields point to, in Program 2 and Program 3 are actually much harder problems, as they involve statically determining what pointers can point to.

This is generally known as points-to analysis, and is a well studied problem [4].

3.1 Introduction to pointer analysis

The general way compilers implement pointer analysis is either using a unification approach, such as [9], or a set constraint based approach, such as [1], [3].

The unification approach is very fast, and can be made somewhat precise (see [2]), but unfortunately patents prevent its use in GCC. As such, GCC has avoided using unification based pointer analysis, and stuck with set constraint based pointer analysis.

The set constraint approach is considered more precise, but more expensive. However, with efficient implementation techniques, these techniques run fast enough for most code bases (See [3]).

Set constraints are a way of modeling program analysis problems that involve sets. They consist of an inclusion constraint language, describing the variables (each variable is a set)

and operations that are involved on the variables, and a set of rules that derive facts from these operations. To solve a system of set constraints, you derive all possible facts under the rules, which gives you the correct sets as a consequence.

3.2 Simple pointer analysis

The most basic constraint system for pointer analysis has a constraint language consisting of the following:

$$p \supseteq q \,|\, p \supseteq \{q\} \,|\, p \supseteq *q \,|\, *p \supseteq q \,|\, *p \supseteq \{q\}$$

Here, $p \supseteq q$ means that set p includes all the members of set q, and $p \supseteq \{q\}$ means that p contains the member q itself. $*$ is the normal dereference operator.

The basic rule system for pointer analysis consists of 4 rules, shown below, and originally described in [1]:

$$[copy]\ \frac{p \supseteq \{q\} \quad r \supseteq p}{r \supseteq \{q\}}$$

$$[load]\ \frac{p \supseteq *q \quad q \supseteq \{r\}}{p \supseteq r}$$

$$[store_1]\ \frac{*p \supseteq q \quad p \supseteq \{r\}}{r \supseteq q}$$

$$[store_2]\ \frac{*p \supseteq \{q\} \quad p \supseteq \{r\}}{r \supseteq \{q\}}$$

The top portion of the rule describes the facts required for the rule to apply. The bottom portion of the rule describes the fact derived from the rule.

In order to generate points-to sets for a program, we transform the program into set constraints, then solve the system of constraints by deriving all the facts. An example program and the set constraints generated from it is given in Program 5.

Program 5 Example program and set constraints generated from them

int * p,**pp,*b,q,c,f	
p = &q	$p \supseteq \{q\}$
b = p	$b \supseteq p$
pp = &f	$pp \supseteq \{f\}$
pp = b	$*pp \supseteq b$
pp = &c	$*pp \supseteq \{c\}$

3.3 Solving

In order to solve the system of set constraints, the following is done:

1. Each variable x has a solution set associated with it, $Sol(x)$.
2. Constraints are separated into direct, copy, and complex. Direct constraints are direct addressof constraints that require no extra processing, i.e. $p \supseteq \{q\}$. Copy constraints are those of the form $p \supseteq q$. Complex constraints are all the constraints involving dereferences. The reason for this partitioning will become clear in a few moments.
3. All direct constraints of the form $p \supseteq \{q\}$ are processed, such that q is added to $Sol(p)$
4. All complex constraints for a given variable are stored in a linked list attached to that variable.
5. A directed graph is built out of the copy constraints. Each variable is a node in the graph, and an edge from q to p is added for each copy constraint of the form $p \supseteq q$
6. The graph is then walked, and solution sets are propagated along the copy edges, such that an edge from q to p causes $Sol(p) \leftarrow Sol(p) \cup Sol(q)$.

7. As we visit each node, all complex constraints associated with that node are processed by adding approriate copy edges to the graph, or the approriate variables to the solution set.

8. The process of walking the graph is iterated until no solution sets change.

As an example, we will use the constraints from program 5. The solution set for each node is displayed under the node label in the brackets.

Before the first iteration of solving, the direct constraints have been processed, and we have an edge from p to b due to the constraint $p \supseteq b$. The result is the graph in figure 3.

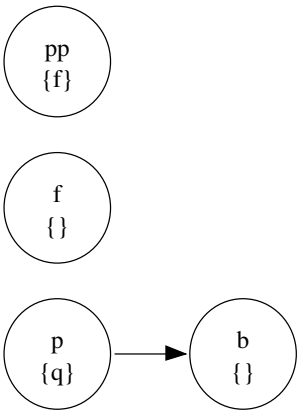

Figure 3: Result at start of iteration 1

During iteration 1, processing of the constraint $*pp \supseteq f$, causes us to add an edge from b to everything in $Sol(pp)$ (which consists of f), giving us the graph in Figure 4. Processing the constraint $*pp \supseteq \{c\}$ causes us to add c to $Sol(x)$ for all x in $Sol(pp)$, which adds c to $Sol(f)$. In addition, we have propagated $Sol(p)$ to $Sol(b)$, causing q to be added to $Sol(b)$.

During iteration 2, $Sol(pp)$ has not changed, so we don't have to add any more edges to the graph due to the $*pp \supseteq f$ constraint. We propagate $Sol(b)$ to $Sol(f)$, adding q to $Sol(f)$.

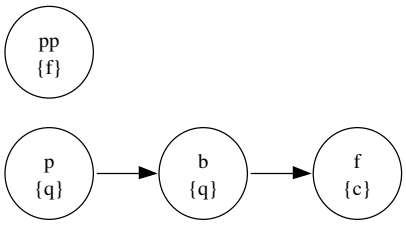

Figure 4: Result at start of iteration 2

Because iteration 3 doesn't change anything, we now have our final result, shown in Figure 5.

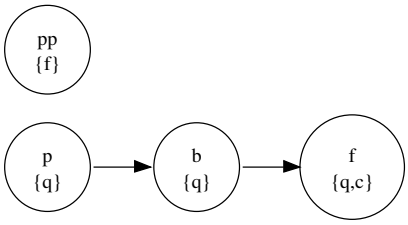

Figure 5: Result at start of iteration 3

Solving this simple set of constraints gives the following points-to sets:

$$p = \{q\}$$
$$pp = \{f\}$$
$$f = \{q, c\}$$
$$b = \{q\}$$

3.4 Handling fields

While the constraint system given before is enough to handle non-aggregates, aggregates present a problem for it in some languages. In a language like Java, there is no way to take the address of a field, so one can simply treat every field like a separate variable, and achieve field-sensitivity that way. However, in C and C++, you run into the issue that one can take the address of a field. This means we need some way to express a constraint that loads the current solution of a variables and adds the appropriate offset to it (the equivalent of `&((*p) + 32`

`bits)`, in order to model taking the address of a field.

In [7], a method is described to both handle structures, and taking the address of a structure field. The method is as follows

1. The constraint graph is turned from a directed graph into a weighted directed graph.
2. All fields in a structure are assigned consecutive blocks of constraint variables, so that if field `a` of `foostruct` in Program 6 was associated with a constraint variable with id 1, field `b` of `foostruct` would be a constraint variable with id 2.
3. The edge weight is used as an offset during constraint processing, and is added to the id of each member of the solution set before the solution set is propagated along the edge. Thus, if the weight of an edge was 1, and the solution set contained field `a`'s variable, the solution set we propagate would now contain field `b`'s variable.

Unfortunately, relying solely on the "index" of a structure field is not easy when generating the constraint variables for a structure (it requires lookup and normalization, and won't work when you allow structures whose first field is not at offset 0), so GCC takes a slightly different approach. We do not assume the constraint variables for a field are consecutively allocated, nor do we use the simple indexes to describe offsets. All constraint variables of an aggregate are linked together in a linked list along with the offset, in bits, from the beginning of the structure, and the size of the field, in bits. Our multigraph weights are also offsets in bits, rather than simple integers.

New rules are also added that deal with this offset, just like those in [7], as follows:

$$[fieldload]\ \frac{p \supseteq (*q + offset) \quad q \supseteq \{r\}}{p \supseteq (r + offset)}$$

$$[fieldstore_1]\ \frac{(*p + offset) \supseteq q \quad p \supseteq \{r\}}{(r + offset) \supseteq q}$$

$$[fieldstore_1]\ \frac{(*p + offset) \supseteq \{q\} \quad p \supseteq \{r\}}{(r + offset) \supseteq \{q\}}$$

$$[fieldaddr]\ \frac{p \supseteq (q + offset) \quad q \supseteq \{r\}}{p \supseteq \{(r + offset)\}}$$

An example program using these new constraints is shown in Program 6.

Program 6 Example program for field based set constraints

struct foo { int *a; int *b }	
int * p,*q, r	
struct foo foostruct	
struct foo *foop	
foop = &foostruct	$foop \supseteq \{foostruct\}$
p = &r	$p \supseteq \{r\}$
foop->b = p	$(*foop + 32) \supseteq p$
q = &foop->b	$q \supseteq (foop + 32)$

Solving this set of constraints gives the following points-to sets:

$$foostruct.a = \{\}$$
$$foostruct.b = \{r\}$$
$$p = \{r\}$$
$$foop = \{foostruct\}$$
$$q = \{foostruct.b\}$$

4 GCC Implementation

There are two implementations currently in progress, one interprocedural, one intraprocedural. The reason for two implementations is that at the time of writing, the interprocedural framework for GCC is still being built; we

wanted to take advantage of whatever field-based alias analysis we could do in the meanwhile.

The intraprocedural version makes conservative assumptions around calls. In particular, we assume globals can point to anything, and that pointers that escape the function can also point to anything.

4.1 Making it fast

Without using some techniques to speed up the constraint solving, set constraint based pointer analysis can become very slow, as has been pointed out over the years. GCC's solver is based on the solver presented in [7], which also covers most of these techniques. The ideas and why they are used are redescribed for the convenience of the reader.

4.1.1 Static Cycle Elimination

Cycle elimination is a very important part of an efficient Andersen's implementation. Cycles in the constraint graph are usually formed because of recursive procedure calls, or pointer variables modified during loops. All members of a cycle will necessarily have the same points to set, because the solution will continually propagate around the cycle. As a result, each cycle will cause additional iterations of the solver, but none of the 0-weight cycles are necessary for correctness or precision of the solutions.

In order to perform this static cycle elimination, we use Tarjan's strongly connected component algorithm [10], as modified by Nuutila to keep only non-root nodes on the stack [5]. All members of a cycle in a graph will be in the same strongly connected component. Thus, once we have found the SCC's of the graph, all nodes in each component are then collapsed to a single node in the graph.

The algorithm to find and collapse static cycles is reprinted as Algorithm 2. Unifying nodes also allows us to store the points-to sets for all the nodes in the cycle once, saving memory. The only caveat is that when querying the points to sets, we have to look at the representative for a node, instead of the original node itself.

Algorithm 2 Collapsing cycles in the graph

```
Compute the strongly connected components
of the graph
for all SCC (c) in the graph do
  for all nodes (n) in c do
    Merge all edges in the graph for n into
    Root(c)
    Sol(n) ← Sol(Root(c)) ∪ Sol(n)
    Rep(n) ← Root(c)
  end for
end for
```

4.1.2 Variable substitution

Prior to solving the constraint graph, off-line variable substitution, as described in [8], is performed, in order to collapse nodes which will end up with equivalent points-to sets. No nodes with 0-weight edges between them are collapsed, in order to preserve correctness of the constraint graph. See [7] for more details on why this is necessary.

4.1.3 Ordering

The order in which nodes and edges are processed is also important to the speed of solving. We process the constraint graph in weak topographic order. [6]

4.1.4 Removing Cycles Dynamically

Processing complex constraints may cause new edges to be added to our constraint graph. These new edges may, in turn, causes additional cycles in graph. In order to remove these cycles that occur during solving, we run cycle elimination at the start of each graph iteration. The cost of doing so is far outweighed by the amount of work it saves.

4.2 Timings

The majority of the time spent in our points-to analysis is spent in the constraint solving, mainly because constraint generation can be done in linear time in a single walk over the program. In order to show that the solving algorithm is scalable enough for GCC's immediate future (next few years), we need to show that the solver can scale to whole programs,or at least large portions of programs, that gcc is likely to see in the next few years.

David Pearce was kind enough to provide his constraint files he generated from several significant size whole programs, including emacs and gcc 2.95. These file consists of a list of variables, and the set constraints from the entire program processed at once.

These constraint files were read into the GCC implementation solver implementation, and the constraint graph solved.

Field insensitive versions are simply the same programs, where seperate variables for seperate fields are not generated (instead, a single variable represents the entire structure). This is a useful comparison for purposes of determining whether we pay too high a cost for field sensitivity.

Timings were as follows (fs = field sensitive, fi = field insensitive):

Program	type	vars	time (s)
make 3.79	fs	6920	0.05
make 3.78	fi	4773	0.03
gcc 2.95	fs	75279	20.1
gcc 2.95	fi	42822	3.1
emacs 20.7	fs	38170	0.178
emacs 20.7	fi	17961	0.313

The reason that emacs is faster in field sensitive mode than field insensitive mode is that field sensitive mode has the constraints split up in a way that causes less iterations of the worklist solver to be needed.

Even when processing an entire program at once (which is something GCC is still building the infrastructure to do), the algorithm still is fast enough to be useful, given that a compilation taking 20 minutes (all of gcc 2.95), 20 seconds is not a long time. In addition, timings when no structure fields are used show that degrading sensitivity if necessary would make it fast enough to useful for even very large programs.

Note that the timings here are significantly faster than those reported in [7], in part because we use a sparse bitmap implementation, so our set union operations are $O(E)$, where E is the number of set bits in the bitmap.

5 Conclusion

An enhanced representation for structure fields offers GCC the ability to transform and optimize a significant amount of code at the tree level that it could not before. Improving GCC's pointer analysis so that it can take into account structure fields also provides significant improvements in what can be optimized by the tree level, and provides a framework for building an interprocedural pointer analysis that can give even more.

References

[1] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[2] M. Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, May 2000.

[3] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[4] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[5] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, Jan. 1994.

[6] D. J. Pearce and P. H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the Workshop on Efficient and experimental Algorithms*, volume 3059 of *Lecture Notes in Computer Science*, pages 383–398. Springer-Verlag, 2004.

[7] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering*. ACM Press, 2004.

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

[9] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[10] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.