Pointer Analysis for Programs with Structures and Casting *

Suan Hsi Yong, Susan Horwitz, and Thomas Reps
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
Electronic mail: {suan, horwitz, reps}@cs.wisc.edu

Abstract

Type casting allows a program to access an object as if it had a type different from its declared type. This complicates the design of a pointer-analysis algorithm that treats structure fields as separate objects; therefore, some previous pointer-analysis algorithms "collapse" a structure into a single variable. The disadvantage of this approach is that it can lead to very imprecise points-to information. Other algorithms treat each field as a separate object based on its offset and size. While this approach leads to more precise results, the results are not portable because the memory layout of structures is implementation dependent.

This paper first describes the complications introduced by type casting, then presents a tunable pointer-analysis framework for handling structures in the presence of casting. Different instances of this framework produce algorithms with different levels of precision, portability, and efficiency. Experimental results from running our implementations of four instances of this framework show that (i) it is important to distinguish fields of structures in pointer analysis, but (ii) making conservative approximations when casting is involved usually does not cost much in terms of time, space, or the precision of the results.

1 Introduction

Static analyses like live variables and constant propagation are useful for optimization and program understanding. When dealing with a programmming language with pointers, pointer analysis must first be used to obtain information about the locations to which a pointer points. This information is often provided in the form of a *points-to set*, which is a safe approximation (superset) of the set of locations to which a pointer *may* point. The sizes of the points-to sets generated by a given pointer-analysis algorithm is a measure of its precision: smaller points-to sets represent more precise results. In many cases, the precision of pointer analysis significantly affects the precision of subsequent static-analysis phases [SH97a].

In a programming language like C, the ability to cast variables and pointers allows a program to access a given object as if it had a type different from its declared type. This makes the treatment of fields of structures in program analysis a non-trivial task, further complicated by the fact that the memory layout of structure objects is implementation dependent. For this reason, some pointer-analysis algorithms "collapse" structures into a single variable [Ste96b, SH97b]. For example, consider the following code fragment:

```
struct S { int *s1; int *s2; } s;
int x, y, *p;
s.s1 = &x;
s.s2 = &y;
p = s.s1;
```

An analysis that collapses structures would ignore types, and would treat these three assignment statements as if they were:

```
s = &x;
s = &y;
p = s;
```

Consequently, a points-to fact of the form s **points-to** a would be interpreted as *"any field of s may point to any field of a"*.

A drawback to this approach is that it can produce very imprecise points-to information. For example, for the code fragment given above, this approach would identify p's points-to set as $\{x, y\}$, while in fact p only points to x. This loss of precision can have an important negative impact on subsequent analysis phases. For instance, a code-slicing experiment that our research group carried out produced disappointing results, and the imprecise treatment of structures in the pointer-analysis phase was identified as a cause. A group at Microsoft Research also found that being able to distinguish between fields of structures was crucial to the accuracy of subsequent analysis phases [Ste96a].

Some other pointer-analysis algorithms handle structures by using exact offsets and sizes [WL95, Ryd98]. For example, assuming that every pointer takes four bytes of storage, and that there is no padding between the fields of an object of type `struct S`, the assignment `s.s2 = &y` would cause a points-to fact of the form "bytes 4 to 7 of `s` contain the address of byte 0 of `y`" to be inferred. This approach

*This work was supported in part by the National Science Foundation under grants CCR-9625656, CCR-9625667 and CCR-9619219, by the U.S.-Israel BSF under grant 96-00337, by grants from IBM and Microsoft, and by a Vilas Associate Award from the Univ. of Wisconsin.

To appear in: Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI).
(c) 1999 ACM

provides very precise points-to information. However, since the layout of structure fields is implementation dependent, the results of a pointer-analysis algorithm that uses this approach are not portable. This is not an issue when pointer analysis is one phase of an optimizing compiler (since the compiler's particular layout strategy can be used both for analysis and for code generation). However, when pointer analysis is used as part of a programming tool, (e.g., for code understanding or restructuring), the analysis results must be consistent with all possible run-time behaviors of the program, which may differ from one compiler to the next, depending on which layout strategy is used for structures. In this case, the use of offsets and sizes is not a safe approach.

This paper provides three main contributions in the area of pointer analysis:

- We identify the problems specific to structures and casting that are relevant to pointer analysis.
- We present a tunable framework for pointer analysis that handles programs with structures and casting. (As presented here, the framework produces flow-insensitive pointer-analysis algorithms; however, as discussed below, the ideas are equally relevant to flow-sensitive analysis.[1]) This work generalizes certain previous algorithms, which can be expressed as instances of our framework.
- To evaluate the relative efficacies of different approaches to handling structures in the presence of casting, we implemented four pointer-analysis algorithms as instances of our framework. Each algorithm was run on 20 C programs, ranging in size from about 650 to 29,000 lines. In each case, we measured the size of the points-to set of each instance of a dereferenced pointer in the program. We found that:
  - The sets computed by an algorithm that distinguishes individual fields of structures are often much smaller than those computed by an algorithm that collapses structures.
  - Using an analysis algorithm that computes portable results (as opposed to results that are safe for only one platform) usually is not too costly, either in time, space, or precision of the results.

Our pointer-analysis framework is presented as an intraprocedural analysis, using rules of inference to express the effect that each kind of assignment statement has on the set of points-to facts. The important issues addressed here (how to handle structures in the presence of casting) are orthogonal to the issues of how to handle function calls and pointers to functions. Our techniques can be used in conjunction with any of the well-known techniques that address those issues to provide an interprocedural analysis. Such techniques are used in our implementation, which performs (context-insensitive) interprocedural points-to analysis.

Although our rules of inference define a flow-*insensitive* analysis, it should be clear that the ideas presented here are also applicable to a flow-sensitive analysis. In particular, the inference rule for each kind of assignment statement is closely related to the dataflow function that would be used for that kind of statement in a flow-sensitive analysis.

The remainder of the paper is organized as follows: Section 2 presents our basic assumptions and notation. Section 3 describes an approach that distinguishes fields of structures, but does not handle casting. Section 4 explains how casting introduces non-trivial problems into pointer analysis, and presents our solution to these problems: an analysis framework parameterized by three functions named *normalize*, *lookup*, and *resolve*. Section 4 includes four different definitions of the three functions, which produce instances of the framework with a wide range of precisions. Experimental results appear in Section 5. Related work is discussed in Section 6.

2 Background: Source Language, Notation, and Terminology

In this paper, we assume that assignment statements have been normalized via the introduction of temporary variables to have one of the following forms (the same forms were used in [Ste96b], and are similar to the ones used in the SUIF intermediate representation [WFW+94]):

1. $s = (\tau_s)\&t.\beta$
2. $s = (\tau_s)\&((*p).\alpha)$
3. $s = (\tau_s)t.\beta$
4. $s = (\tau_s) * q$
5. $*p = (\tau_{*p})t$

Optional casts are denoted by (τ_s) and (τ_{*p}), using τ_s to mean the type of variable `s`, and τ_{*p} to mean the type to which variable `p` is declared to point. Single lowercase letters (`s`, `t`, `p`, ...) are used to represent "top level" objects, that is, objects that are not contained within a structure, and lowercase Greek letters $(\alpha, \beta, \gamma, \ldots)$ are used to represent (possibly empty) sequences of field names.

Note that the only operators on the right-hand sides of the assignments are pointer dereferences and field selections. How to handle other operators (e.g., arithmetic operators) is discussed in Section 4.2.1.

Also note that the direct casting of non-scalar types like structures is not permitted in ANSI C. For example, in the following code fragment:

```
struct A { int *a1; } a,;
struct B { int *b1; } b, *p;
b = (struct B)a;
```

the assignment `b = (struct B)a` is not legal C. However, we can always achieve the effect of this assignment indirectly by using the following code fragment:

```
p = (struct B *)&a;
b = *p;
```

Therefore, to facilitate a cleaner presentation of examples, we allow direct casting of structures in this paper.

Heap-allocated memory locations are assumed to be handled by introducing allocation-site-specific pseudo-variables to represent all memory blocks that are allocated at a given allocation site. For example, the statement `p = malloc(...)` at allocation site 1 would be treated as `p = &malloc_1;` where `malloc_1` is a newly created variable.

All array elements are assumed to be treated as a single location (i.e., an assignment to any element of an array `A` is considered to be an update of `A` itself).

[1] A flow-insensitive analysis abstracts away from the actual order in which statements appear in the program, and, in effect, assumes that any statement in the program can follow any other statement in the program. The major advantage of a flow-insensitive pointer analysis is efficiency (albeit at the cost of obtaining less precise points-to information): only a single copy of the abstract store needs to be maintained during the analysis.

Handling unions is similar to handling structures, but introduces additional complications, which we do not discuss here due to space constraints. However, our implementation does handle unions safely.

3 Handling Structures Without Casting

Figure 1 presents rules of inference for flow-insensitive pointer analysis in which fields of structures are distinguished, but casting is not permitted. There is one inference rule for each of the five kinds of assignment statements listed in Section 2. An inferred points-to fact of the form $pointsTo(s.\alpha,\ t.\beta)$ means that $\mathtt{s}.\alpha$ may point to $\mathtt{t}.\beta$ (recall that α and β are used to represent possibly empty sequences of field names). Concatenation of field names α and β is denoted by $\alpha\|\beta$.

1. $$\frac{s\ =\ \&t.\beta \in Prog}{pointsTo(s,\ t.\beta)}$$

2. $$\frac{s\ =\ \&((*p).\alpha) \in Prog,\quad pointsTo(p,\ t.\beta)}{pointsTo(s,\ t.\beta\|\alpha)}$$

3. $$\frac{s\ =\ t.\beta \in Prog,\quad pointsTo(t.\beta\|\gamma,\ u.\delta)}{pointsTo(s.\gamma,\ u.\delta)}$$

4. $$\frac{s\ =\ *q \in Prog,\quad pointsTo(q,\ t.\beta),\ \ pointsTo(t.\beta\|\gamma,\ u.\delta)}{pointsTo(s.\gamma,\ u.\delta)}$$

5. $$\frac{*p\ =\ t \in Prog,\quad pointsTo(p,\ s.\alpha),\ \ pointsTo(t.\beta,\ u.\gamma)}{pointsTo(s.\alpha\|\beta,\ u.\gamma)}$$

Figure 1: Rules of inference for flow-insensitive pointer analysis, assuming no casting.

These rules can be used to compute a safe set of points-to facts for individual fields of structures in programs that do not use casting. To illustrate this, consider again the code fragment used in the Introduction, normalized to use only the five kinds of assignments presented in Section 2 and with line numbers added for reference (the original assignments are provided to clarify the relationship of the normalized assignments to the original code):

```
1: struct S { int *s1; int *s2; } s;
2: int x, y, *p, **tmp1, *tmp2, **tmp3, *tmp4;
3: tmp1 = &s.s1;  }
4: tmp2 = &x;     } s.s1 = &x;
5: *tmp1 = tmp2;  }
6: tmp3 = &s.s2;  }
7: tmp4 = &y;     } s.s2 = &y;
8: *tmp3 = tmp4;  }
9: p = s.s1;
```

Recall that this example was used in the Introduction to illustrate the imprecision of the approach that collapses all fields of a structure. Using that approach, it would be inferred that variable p's points-to set is {x, y}. However, using the rules in Figure 1, the more precise information that p's points-to set is {x} is inferred in three steps from Statements 3, 4, 5, and 9, as described below; Statements 6 – 8 have no effect on p's points-to set:

Step 1: Rule 1 applied to Statements 3 and 4 produces the points-to facts $pointsTo(tmp1,\ s.s1)$ and $pointsTo(tmp2,\ x)$.

Step 2: Rule 5 applied to Statement 5 (using the two points-to facts from Step 1) produces the new points-to fact $pointsTo(s.s1,\ x)$.

Step 3: Rule 3 applied to Statement 9 (using the points-to fact from Step 2) produces the new points-to fact: $pointsTo(p,\ x)$.

For programs that do use casting, however, these rules can fail to infer certain points-to facts. For example, assume that variables a, b, and x have been defined as follows:

```
struct A { int *a1; } a;
struct B { int *b1; } b;
int x;
```

and that a.a1 points to x. Consider the assignment:

```
b = (struct B)a;
```

We might try to handle this assignment by extending Rule 3 to permit a cast on the right-hand side of the assignment (and making no other changes to Rule 3):

3. $$\frac{s\ =\ (\tau_s)t.\beta \in Prog,\quad pointsTo(t.\beta\|\gamma,\ u.\delta)}{pointsTo(s.\gamma,\ u.\delta)}$$

However, an application of this rule produces $pointsTo(b.a1,\ x)$, which makes no sense, since variable b has no a1 field. Furthermore, the desired fact $pointsTo(b.b1,\ x)$ cannot be inferred using only the rules in Figure 1.

4 Handling Structures with Casting

4.1 Problems

Casting allows an object to be accessed as if it had a type different from its declared type. For structures, casting allows a different "layout pattern" to be substituted for the structure's declared "layout pattern". Therefore, for an analysis to be safe and portable, it must assume only those correspondences between the two layout patterns that are guaranteed by the rules of C that govern the layout of the fields of structures. For ANSI C, the relevant guarantees are as follows [ISO90, Sections 6.3.2.3 and 6.5.2.1]:

- The first field of a structure is guaranteed to be at offset 0.
- If two structures contain an initial sequence of fields, all of which have compatible types[2] (and, for bit fields,

[2] The purpose of *compatible types* is to allow type declarations that are similar, but not identical, to match. This is needed to handle types declared in different translation units (including types declared in a single include file but included in multiple translation units). An int is compatible with an enum, and a variable that is volatile is only compatible with another variable of the same type if it is also volatile (likewise for const). Two pointers have compatible types only if the types of the objects they point to are compatible [ISO90].

the same widths), then the offsets of the corresponding fields in the initial sequence are guaranteed to be the same.

Note that there are no guarantees based on the sizes of the fields of a structure; for example, just because the first fields of two structures have types with the same size does not guarantee that the offsets of the second fields will be identical (that is only true if the types of the first fields are compatible).

With these restrictions in mind, we turn now to the problems that can arise when there is casting in a program with structures.

Problem 1

The first problem has to do with the guarantee mentioned above that the first field of a structure is always at offset 0. This means that a pointer that points to a structure object also points to the first field of that structure. The following example illustrates this problem:

```
1:  struct S { int *s1; } s, *p;
2:  int x, *q, *r;
3:  p = &s;
4:  q = &x;
5:  *p = (struct S)q;
6:  r = s.s1;
```

Rule 1 from Figure 1 can be used with Statements 3 and 4 to infer that p points to s and q points to x. If Rule 5 from Figure 1 is extended to permit a cast on the right-hand side, it can be used with Statement 5 to infer that s points to x. However, the rules from Figure 1 do not permit the inference that s.s1 also points to x, and therefore it is not possible to infer from Statement 6 that r also points to x.

Similarly, a pointer that points to the first field of a structure can (with a suitable cast) be treated as a pointer to the structure object itself, and a structure object whose first field is a pointer can (with a suitable cast) be used as if it were that pointer.

Problem 2

The second problem arises when a pointer is dereferenced whose declared type does not match the type of the structure to which it actually points. For example, consider the following code fragment:

```
1:  struct S { int *s1; int  s2; char *s3; } *p;
2:  struct T { int *t1, int *t2; char *t3; } t;
3:  char **c;
4:  p = (struct S *)&t;
5:  c = &((*p).s3);
```

In Statement 5, when p (a pointer of type struct S *) is dereferenced, it is actually pointing to t (an object of type struct T). A pointer-analysis algorithm must be able to identify safely which field(s) of t are actually referenced by the expression (*p).s3. Because the second fields of struct S and struct T have non-compatible types, (*p).s3 may or may not correspond to t.t3.

Problem 3

The third problem occurs when a memory block of one type is copied into a block of a different type. For example, consider again the two structures declared above, with a different assignment:

```
1:  struct S { int *s1; int  s2; char *s3; } s;
2:  struct T { int *t1, int *t2; char *t3; } t;
3:  s = (struct S)t;
```

In this example, Statement 3 copies the structure object t into s. Here, a pointer-analysis algorithm must be able to identify which field or fields of the source object (t) are (or may be) copied into each field of the destination object (s).

4.2 Solution: Functions *normalize*, *lookup*, and *resolve*

Our solution to the problems introduced by casting involves using three auxiliary functions: *normalize* (for Problem 1), *lookup* (for Problem 2), and *resolve* (for Problem 3). It is the use of these functions that gives us a *framework* for pointer analysis rather than a single algorithm. Different definitions of these three functions produce pointer-analysis algorithms of varying complexities and varying levels of precision.

- The *normalize* function is used to ensure that all subfields of a structure that have the same offset within the structure are mapped to the same canonical representative. For example, given the assignment s = (struct S)&x (where s is a struct S whose first field, s1, is a pointer), the result of *normalize*(s) determines whether it is inferred that s itself points to x, or that s.s1 points to x. (Similarly, *normalize* is used to determine what is inferred as being pointed to when the address of a structure object or one of its fields is used on the right-hand side of an assignment.)
- The *lookup* function is used to identify the field that is referenced by a dereferenced pointer. When the declared type of the pointer does not match the type of an object to which it might point, *lookup* returns a safe approximation to the set of fields that are actually referenced. For example, for the expression (*p).s3 (where p is declared to be a pointer to a struct S, but pointer analysis has determined that p might actually point to t, which is a struct T), *lookup*(struct S, s3, t) returns the field or fields of t that correspond to the s3 field of struct S.
- The *resolve* function is used to match each field of a structure of one type with the corresponding field(s) of a structure of another type. For example, for the assignment s = (struct S)t (where s is a struct S and t is a struct T), *resolve*(s, t, struct S) returns the set of pairs $\langle s.\gamma,\ t.\gamma' \rangle$, such that γ is a field of s, and γ' is a corresponding field in t. (The reason for the third argument to *resolve* is explained below.)

Figure 2 presents new definitions of the inference rules from Section 3, modified to use the functions *normalize*, *lookup*, and *resolve* to handle casting. In Figure 2, τ_s denotes the type of s, and τ_{*p} denotes the type that p is declared to point to. A variable followed by a Greek letter with no hat (e.g., t.β) represents a non-normalized structure reference (i.e., in this case, the Greek letter represents a possibly empty sequence of field names, as before). A variable followed by a Greek letter with a hat over it (e.g., $\widehat{\texttt{t}.\beta}$) represents a normalized structure reference. (In the definitions of *normalize* that follow, the β of $\widehat{t.\beta}$ will either be an integer offset, or a possibly empty sequence of field names.) Note that the values returned by functions *lookup* and *resolve*, as well as those in the *pointsTo* relation are always in normalized form. Note also that in Rules 2, 4, and 5, some

1.
$$\frac{s \;=\; (\tau_s)\ \&t.\beta \quad \in Prog}{pointsTo(normalize(s),\ normalize(t.\beta))}$$

2.
$$\frac{\begin{array}{c} s \;=\; (\tau_s)\ \&((*p).\alpha) \quad \in Prog, \\ pointsTo(normalize(p),\ \widehat{t.\beta}), \\ \widehat{t.\gamma} \in lookup(\tau_{*p},\ \alpha,\ \widehat{t.\beta}), \end{array}}{pointsTo(normalize(s),\ \widehat{t.\gamma})}$$

3.
$$\frac{\begin{array}{c} s \;=\; (\tau_s)\ t.\beta \quad \in Prog, \\ \langle \widehat{s.\gamma}, \widehat{t.\gamma'} \rangle \in \\ resolve(normalize(s),\ normalize(t.\beta),\ \tau_s), \\ pointsTo(\widehat{t.\gamma'},\ \widehat{u.\delta}) \end{array}}{pointsTo(\widehat{s.\gamma},\ \widehat{u.\delta})}$$

4.
$$\frac{\begin{array}{c} s \;=\; (\tau_s) * q \quad \in Prog, \\ pointsTo(normalize(q),\ \widehat{t.\beta}), \\ \langle \widehat{s.\gamma}, \widehat{t.\gamma'} \rangle \in resolve(normalize(s),\ \widehat{t.\beta},\ \tau_s), \\ pointsTo(\widehat{t.\gamma'},\ \widehat{u.\delta}) \end{array}}{pointsTo(\widehat{s.\gamma},\ \widehat{u.\delta})}$$

5.
$$\frac{\begin{array}{c} *p \;=\; (\tau_{*p})t \quad \in Prog, \\ pointsTo(normalize(p),\ \widehat{s.\alpha}), \\ \langle \widehat{s.\gamma}, \widehat{t.\gamma'} \rangle \in resolve(\widehat{s.\alpha},\ normalize(t),\ \tau_{*p}), \\ pointsTo(\widehat{t.\gamma'},\ \widehat{u.\delta}) \end{array}}{pointsTo(\widehat{s.\gamma},\ \widehat{u.\delta})}$$

Figure 2: Rules of inference for flow-insensitive pointer analysis in the presence of casting.

of the arguments passed to *lookup* and *resolve* come from *pointsTo* facts, and thus are in normalized form. To be consistent with these cases, we have defined the inference rules so that *lookup*'s 3rd argument, as well as *resolve*'s 1st and 2nd arguments are always in normalized form. Therefore, the example calls to *lookup* and *resolve* given above were not quite right; the actual calls would be: *lookup*(`struct S, s3,` *normalize*(`t`)), and *resolve*(*normalize*(`s`), *normalize*(`t`), `struct S`).

In Section 4.2.2, we define versions of *normalize*, *lookup*, and *resolve* that are specific to a particular layout strategy (i.e., the definitions are in terms of the offsets and sizes of structure fields, and thus are not portable). Section 4.3 presents three alternative definitions of these functions that produce safe pointer-analysis algorithms that are less precise than the one defined in Section 4.2.2, but are portable (i.e., are safe for any layout strategy).

However, before we define functions *normalize*, *lookup*, and *resolve*, we first discuss a few of the strange situations that can arise in the presence of casting that complicate the definitions of *lookup* and *resolve*.

4.2.1 Complicating Factors

The problems identified in Section 4.1 can be exacerbated by some legal (but not very nice) coding practices. These in turn complicate the definitions of *lookup* and *resolve*. In this section, we describe four complicating factors. Sections 4.2.2 and 4.3 give definitions of *normalize*, *lookup*, and *resolve* that handle these complications.

In what follows, we illustrate each complicating factor using a statement that performs a copy between structures of different types; i.e., we illustrate the impact on the definition of function *resolve*. Similar examples can easily be created for function *lookup*.

Complication 1

Through casting, it is possible to access fields beyond the bounds of a nested structure object. This is illustrated in the following example:

```
struct V {           struct W {
   int  *v1;            int *w1;
   char *v2;            struct R {
   int  *v3;               int  *r1;
} v;                       char *r2;
                        } r;
                        int *w3;
                     } w;
```

Given these definitions (and the assumptions that every field takes 4 bytes and there is no padding between fields), the assignment `v = (struct V)w.r`, in addition to accessing `w.r.r1` and `w.r.r2`, will access the field `w.w3`, which is outside the bounds of `w.r`. This complicates the definition of function *resolve*: when matching the fields of `v` and `w.r`, *resolve* must take into account the other fields of the structure *enclosing* `w.r` as well as the fields of `w.r` itself.

Complication 2

A single non-structure object may be large enough to hold more than one pointer value.[3] This is illustrated by the following example:

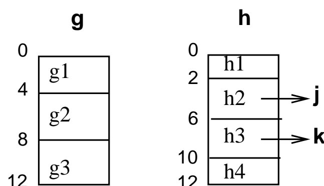
```
struct R { int *r1; int *r2; } r;
double d;
int x, y;
d = (double)r;
```

Assume that `r.r1` points to `x`, and `r.r2` points to `y`. If the size of a `double` is large enough to hold the entire structure `r`, then we must keep track of the fact that variable `d` (or two artificially introduced "sub-fields" of `d`) point to `x` and `y`, since both of those addresses can be recovered from `d` (e.g., by casting `d` back to `struct R`). This means that function *resolve* must sometimes either "break up" an atomic object (such as a `double`) into artificial sub-fields, or must match one object with multiple fields of another object.

Complication 3

Complication 2 can be generalized to partially overlapping objects. For example, consider the following two structures:

[3] Note that for a pointer-analysis algorithm to be safe in the presence of casting, it must track the points-to sets of *all* variables, even those not declared as pointers.



The numbers denote the offsets of the fields within structures `g` and `h`. What points-to information should be inferred from the assignment `g = h`? An analysis specific to the particular layout strategy pictured above can infer that bytes 2 – 5 of `g` contain the address of `j`, and bytes 6 – 9 of `g` contain the address of `k`. Because the values of fields `g1`, `g2`, and `g3` are defined by splicing together pieces of different values from `h`, an analysis algorithm that does *not* depend on specific offset and size information might infer that fields `g1`, `g2`, and `g3` may point to *any* location. This information could be recorded using a special value *Unknown* to represent a pointer that may have been corrupted. The use of *Unknown* could be useful for flagging potential "misuses" of memory in a program.

However, such a strategy may be overly pessimistic, leading to poor results for both pointer analysis and subsequent static analyses. In practice, it may be reasonable to adopt a more optimistic approach by making the following assumption:

Assumption 1 *Whenever a pointer is dereferenced, its value is a valid address that was generated by an allocation operation, or by an application of the address-of* (`&`) *operator (including implicit address-of operations such as* `p = f`*, where* `f` *is the name of a function;* `p = A`*, where* `A` *is the name of an array; or* `p = A+2`*, where* `A` *is the name of an array of size* ≥ 3*).*

We feel this is a reasonable assumption to make, since violations should be rare, and may be considered inconsistent uses of memory. Therefore, for the remainder of this paper we will assume that Assumption 1 holds.

Given Assumption 1, a portable pointer-analysis algorithm should not conclude that `g1`, `g2`, and `g3` point to arbitrary locations. Instead, it should conclude that `g1` and `g2` point to `j`, and that `g2` and `g3` point to `k`, since the pointers to `j` and `k` can be recovered from those fields of `g`. (In this approach, "`g1` points to `j`" means "`g1` may hold part of the address of `j`".) As in the case of Complication 2, this means that function *resolve* must sometimes either break up an object into sub-fields, or must match a single field of one structure with multiple fields of another structure.

Note that a related issue is how to handle pointer arithmetic, as well as operations on pointer values that would normally be considered erroneous (for example, adding two addresses, shifting an address, multiplying an address by an integer value). The results of such operations could be considered to point to arbitrary locations. However, under Assumption 1, better information can be inferred. Pointer arithmetic can be used to move a pointer from one field of a structure to another. For example, if variable `g` pictured above is of type `struct G`, and a pointer `p` is pointing to its `g1` field, then the two assignments

```
k = offsetof(struct G, g2);
p = (char *)p + k;
```

set `p` to point to `g`'s `g2` field. In general, the actual value added to or subtracted from a pointer will not be known statically; therefore, to be safe (given Assumption 1), a pointer-analysis algorithm should treat the result of any arithmetic operation on a pointer that points to some field of a structure `s` as possibly pointing to any sub-field of `s`, or of any structure containing `s`.

Complication 4

When an assignment is performed, it is the type of the left-hand side that determines how many bytes are actually copied. In particular, in the case of our fifth kind of assignment statement:

$$5.\ *p = (\tau_{*p})t$$

it is the type to which `p` is declared to point that determines how many bytes are copied, not the type of the object to which `p` actually does point. This complicates the definition of function *resolve*; in particular, this is why we have defined *resolve* to have three arguments — the destination of the copy, the source of the copy, and the declared type of the destination — instead of just the destination and source of the copy.

The reason it is important for *resolve* to know the type of the left-hand side of an assignment is illustrated by the following example:

```
1:  struct R { int *r1; int *r2; char *r3; } r;
2:  struct S { int *s1; int *s2; int *s3; } s;
3:  struct T { int *t1, int *t2; } *p;
4:  p = (struct T *)&r;
5:  *p = (struct T)s;
```

When Statement 5 is processed, *resolve* is called to match the fields of `r` (the destination of the copy) with the corresponding fields of `s` (the source of the copy). Because the declared type of the left-hand side of the assignment is `struct T`, only the first two fields of `s` are copied into `r`; there is no effect on the value of `r.r3`. Therefore, ideally, *resolve* should only return the pairs $\langle$`r.r1`, `s.s1`$\rangle$ and $\langle$`r.r2`, `s.s2`$\rangle$. Neither the type of the destination of the copy (`struct R`) nor the type of the source of the copy (`struct S`) provides information to function *resolve* about how much memory is copied. Therefore, the declared type of the left-hand side of the assignment is passed to *resolve* as a third argument.

4.2.2 Versions of *normalize*, *lookup*, and *resolve* for a specific layout strategy: "Offsets" Approach

In this section, we give definitions of *normalize*, *lookup*, and *resolve* that are specific to a particular layout strategy (i.e., they assume that offsets and sizes are known for every field of a structure, and that sizes are known for every object). These definitions provide the most precise instance of our pointer-analysis framework; however, the results of the analysis may not be portable. Section 4.3 presents three different instantiations of the framework that are portable and provide a range of precisions.

As discussed in Section 4.2.1, Complication 1 may make it necessary for *lookup* and *resolve* to return offsets in an outermost containing structure (e.g., variable `w` in the motivating example), rather than in the substructure that appears in the assignment statement being analyzed (e.g., `w.r` in the same example). Also, Complications 2 and 3 may make it necessary to keep track of offsets in objects that are not structures (such as the `double` used to motivate Complication 2), as well as offsets that do not correspond to declared fields of structures. Therefore, the versions of *normalize*, *lookup*, and *resolve* given below return values that are pairs

of the form ⟨*outermost containing object, offset*⟩ (where *offset* is an integer value, not a sequence of field names). We use `s`.j and `t`.k as formal parameters of *lookup* and *resolve* to emphasize that the corresponding actual parameters consist of an object with an integer offset. Finally, note that because the inference rules define points-to facts in terms of the values returned by *normalize*, *lookup*, and *resolve*, in the context of the definitions of the three functions given below, the points-to fact $pointsTo(s.j,\ t.k)$ should be understood to mean "the value stored at offset j in variable `s` may be the address of variable `t` plus k."

Function *normalize*

Given specific offset and size information, the *normalize* function simply needs to map an object's (possibly empty) sequence of field names to the corresponding offset. The function is defined as follows, using `offsetof` to mean the same thing as the C `offsetof` macro (defined in `stddef.h`), and using τ_s to denote the type of `s`:

$$normalize(s.\alpha) = \begin{array}{l}\textbf{if } \alpha \textit{ is empty } \textbf{then } s.0 \\ \textbf{else let } j = offsetof(\tau_s,\ \alpha) \textbf{ in } s.j\end{array}$$

To illustrate this definition of *normalize*, consider the following code fragment:

```
1:  struct S { int s1; } s;
2:  int *p;
3:  p = (int *)&s;
4:  p = &s.s1;
```

Rule 1 from Figure 2 is applicable to both Statements 3 and 4. Using this rule involves applying *normalize* to the left- and right-hand sides of the assignment. Both *normalize*(`s`) and *normalize*(`s.s1`) return `s.0` (and *normalize*(`p`) returns `p.0`). Therefore, both statements lead to the inference of the points-to fact $pointsTo(p.0,\ s.0)$.

Function *lookup*

Function *lookup* is defined as follows:[4]

$$lookup(\tau,\ \alpha,\ t.k) = \{t.n \mid n = k + offsetof(\tau, \alpha)\}$$

Recall that function *lookup* is used in the context of a dereference such as `(*p).s3`, where `p` is declared to be a pointer to a `struct S`, but actually points to `t`, which is a `struct T`. In this case, the expression *lookup*(`struct S`, `s3`, *normalize*(`t`)) is used to determine which field of `t` is actually referenced. In other words, *lookup*'s arguments and return value should be understood as follows:

τ	the type that the dereferenced pointer is declared to point to
α	the field name that appears in the expression (a valid field of τ)
$t.k$	what `p` actually points to (offset k in object `t`)
$t.n$	the offset in `t` that is actually referenced

[4]Recall that we have assumed that every array is represented as if it were of size 1. Function *lookup* as described above does not account for this assumption. To handle arrays safely, *lookup* must be modified so that if $t.n$ is within any element of an array, n is adjusted to be the corresponding offset within the array's (single) representative element.

Note that this version of *lookup* always returns a single value. However, some of the portable versions of *lookup* need to return a set of values; therefore, to avoid having different versions of our rules of inference, we define *lookup* to return a set here.

Function *resolve*

Function *resolve* is defined as follows:[5]

$$resolve(s.j,\ t.k,\ \tau) = \left\{ \langle s.m, t.n\rangle \;\middle|\; \begin{array}{l} i \textit{ is an integer in the range} \\ [0..sizeof(\tau)-1], \textit{ and } m = j + i, \\ \textit{and } n = k + i \end{array} \right\}$$

Recall that function *resolve* is used in the context of a copy between structures of different types, such as `s = (struct S)t`, where `s` is a `struct S` and `t` is a `struct T`. In this case, the expression *resolve*(*normalize*(`s`), *normalize*(`t`), `struct S`) is used to match each field of `s` with the corresponding field in `t`. In other words, *resolves*'s arguments and return values should be understood as follows:

$s.j$	the starting point in the destination of the copy (offset j in object `s`)
$t.k$	the starting point in the source of the copy (offset k in object `t`)
τ	the declared type of the left-hand side of the assignment
$s.m$	an offset in the destination object
$t.n$	the corresponding offset in the source object

Note that, because of Complications 2 and 3, we must essentially treat each byte of a structure as a sub-field. Therefore, *resolve* must match *every* byte in the destination object with the corresponding byte in the source object; that is why it returns a pair $\langle s.m,\ t.n\rangle$ for every value of i in the range $0..sizeof(\tau){-}1$.

4.3 Alternative Function Definitions

In practice, precise offset and size information may not be available when static analysis is carried out, or it may be desirable to perform a static analysis whose results are safe for all possible layout strategies that conform to the ANSI C standard. To address this issue, we can modify the definitions of *normalize*, *lookup*, and *resolve* so that they do not rely on offsets and sizes, but still return safe results. (Of course, these results will in general be less precise than those obtained using exact offset and size information.)

Three new sets of definitions of *normalize*, *resolve*, and *lookup* are presented below. These definitions provide instances of our pointer-analysis framework that are safe for all layout strategies, and span an interesting range of precisions.

4.3.1 "Collapse Always" Approach

Given Assumption 1, the most general and least precise strategy is to "collapse" all structures into a single variable; i.e., to treat every read or write of a field of a structure as if it were a read or write of the structure itself. Consequently, a points-to fact of the form $pointsTo(s,\ t)$ would be interpreted as "any field of `s` may point to any field of `t`". (An

[5]As in the case of *lookup*, this definition of *resolve* must be modified to account for arrays: if $s.m$ or $s.n$ is within an array, m or n must be adjusted accordingly.

example of this approach was given in the Introduction.) This approach can be implemented using the following definitions of *normalize*, *lookup*, and *resolve*:

$$normalize(s.\alpha) = \quad s$$

$$lookup(\tau,\ \alpha,\ t.\beta) = \quad \{\ t\ \}$$

$$resolve(s.\alpha,\ t.\beta,\ \tau) = \quad \{\ \langle s,\ t\rangle\ \}$$

The "Collapse Always" approach can be illustrated using the first part of the example from the Introduction and Section 3:

```
1 : struct S { int *s1; int *s2; } s;
2 : int x, **tmp1, *tmp2;
3 : tmp1 = &s.s1;  \
4 : tmp2 = &x;      > s.s1 = &x;
5 : *tmp1 = tmp2;  /
```

Statements 3 and 4 illustrate the use of *normalize* in Rule 1. For Statement 3, the calls to *normalize* and the return values would be:

call	returned value
normalize(`tmp1`)	`tmp1`
normalize(`s.s1`)	`s`

which would lead to the inferred points-to fact $pointsTo(tmp1,\ s)$. A similar use of Rule 1 for Statement 4 would cause the points-to fact $pointsTo(tmp2,\ x)$ to be inferred. Finally, Rule 5 would be applied to Statement 5, leading to the call *resolve*(`s, tmp2, int *`), which would return { ⟨*s, tmp2*⟩ }. This pair, in conjunction with $pointsTo(tmp2,\ x)$, would cause $pointsTo(s,\ x)$ to be inferred.

4.3.2 "Collapse on Cast" Approach

A strategy that leads to more precise results than the "Collapse Always" approach described above is to collapse a structure only when it is accessed as if it had a type different from its declared type. This approach can be implemented using the definitions of *normalize*, *lookup*, and *resolve* given below.

normalize($s.\alpha$) =
  if *$s.\alpha$ is a structure object with first field s1*
  then *normalize*($s.\alpha \| s1$)
  else $s.\alpha$

Function *normalize* maps every structure object $\mathtt{s}.\alpha$ to its innermost first field (normalization is recursive because the first field of $\mathtt{s}.\alpha$ may itself be a structure).

Function *lookup*, defined below, uses an auxiliary function named *followingFields*, which takes two arguments: a structure type τ and field β of τ, and returns the set of fields that come after β in τ.[6]

lookup($\tau, \alpha, t.\beta$) =
  if $\exists\ \delta$ *such that* ($normalize(t.\delta) = t.\beta$) *and* ($\tau_{t.\delta} = \tau$)
  then { $normalize(t.\delta\|\alpha)$ }
  else $\left\{ normalize(t.\gamma) \;\middle|\; \begin{array}{l} \gamma = \beta,\ or \\ \gamma \in followingFields(\tau_t, \beta) \end{array} \right\}$

[6] To handle arrays safely, the *followingFields* of a field within an array must include *all* fields within that array.

Function *lookup* checks whether τ (the type that the dereferenced pointer is declared to point to) is the same as the type of an enclosing structure δ of which β is the innermost first field (recall that a pointer that points to the first field of a structure also points to the structure itself). If the types are the same, it means that $\mathtt{t}.\delta$ must have an α field, so *lookup* returns that field (normalized). If the types are not the same, *lookup* returns all fields of `t` starting with $\mathtt{t}.\beta$. (Because of Complication 1, the α field of τ may correspond to a field of `t` that is not a sub-field of $\mathtt{t}.\beta$. Therefore, *lookup* returns all fields of `t` starting with β, rather than just the sub-fields of $\mathtt{t}.\beta$).

This definition of *lookup* can be illustrated using the following example:

```
1:  struct S { int s1; char s2; } *p, *q;
2:  struct T {   struct S t1;
                 int t2;
                 char t3; } t;
3:  char *x, *y;
4:  p = &t.t1;
5:  x = &(*p).s2;
6:  q = (struct S *)&t.t2;
7:  y = &(*q).s2;
```

When Statement 5 is processed, the call *lookup*(`struct S, s2, t.t1.s1`) is made (`t.t1.s1` is the normalized form of `t.t1`). Field `t.t1.s1` is the first field of `t.t1`, which in turn is the first field of `t`. Therefore, the candidates for δ are: `[]`, `t1`, and `t1.s1` (where `[]` means the empty sequence of field names). The type of `t.t1` matches the given type (`struct S`), so *normalize*(`t.t1.s2`) is returned.

When Statement 7 is processed, the call *lookup*(`struct S, s2, t.t2`) is made. Field `t.t2` is not the first field of any enclosing structure; therefore, the only candidate for δ is `t2` itself. The type of `t.t2` does not match the given type (`struct S`), so *lookup* returns all of the (normalized) fields of `t` starting with `t.t2`; i.e., {`t.t2`, `t.t3`}.

Function *resolve* is defined as follows:

$$resolve(s.\alpha, t.\beta, \tau) = \left\{ \langle\gamma,\ \gamma'\rangle \;\middle|\; \begin{array}{l} \delta\ is\ a\ field\ of\ \tau,\ and \\ \gamma \in lookup(\tau,\ \delta,\ s.\alpha),\ and \\ \gamma' \in lookup(\tau,\ \delta,\ t.\beta) \end{array} \right\}$$

Function *resolve* uses *lookup* to match each field δ of τ (the declared type of the left-hand side of the assignment) with the corresponding field(s) of $\mathtt{s}.\alpha$ (the destination object), and with the corresponding field(s) of $\mathtt{t}.\beta$ (the source object). It then pairs the resulting fields from $\mathtt{s}.\alpha$ with the resulting fields from $\mathtt{t}.\beta$:

- If the three types (τ, $\tau_{s.\alpha}$, and $\tau_{t.\beta}$) are all the same, then every field α in the destination is also in the source (and only those fields will be copied). In this case, *lookup* will match each field δ of τ with the δ field of $\mathtt{s}.\alpha$ and the δ field of $\mathtt{t}.\beta$, and those pairs of fields will be returned by *resolve*.
- If τ is the same as $\tau_{s.\alpha}$ but different from $\tau_{t.\beta}$, then for each field δ of τ, *lookup* will return the δ field of $\mathtt{s}.\alpha$, and *all* fields of `t`, starting with β. In turn, *resolve* will return the cross-product of all sub-fields of $\mathtt{s}.\alpha$, with all fields of `t` starting with β.
- Similarly, if τ is the same as $\tau_{t.\beta}$ but different from $\tau_{s.\alpha}$, then *resolve* will return the cross-product of all fields of `s` starting with α, with all of the sub-fields of $\mathtt{t}.\beta$.

- Finally, if τ is different from both $\tau_s.\alpha$ and $\tau_t.\beta$, then *resolve* will return the cross-product of all fields of `s` starting with α, with all fields of `t` starting with β.

4.3.3 "Common Initial Sequence" Approach

Two structures share a common initial sequence if for a sequence of one or more initial fields, all corresponding fields have compatible types. The ANSI C standard requires that the offsets of corresponding fields in the common initial sequence of two structures be the same [ISO90, Sections 6.3.2.3 and 6.5.2.1].

We can define versions of *normalize*, *lookup*, and *resolve* that take advantage of this requirement to produce a pointer-analysis algorithm that is both portable and more precise than the one that uses the "Collapse on Cast" approach. The idea is to "collapse" fields of a structure only when it is accessed as a type different from its declared type, and the accessed field(s) are not part of a common initial sequence. This approach can be implemented using the definitions of *normalize*, *lookup*, and *resolve* given below.

The definitions of *normalize* and *resolve* are the same as the ones used in the "Collapse on Cast" approach:

$$
\begin{array}{l}
normalize(s.\alpha) = \\
\quad \textbf{if } \textit{s.}\alpha \textit{ is a structure object with first field s1} \\
\quad \textbf{then } normalize(s.\alpha \| s1) \\
\quad \textbf{else } s.\alpha
\end{array}
$$

$$
resolve(s.\alpha, t.\beta, \tau) = \left\{ \langle \gamma,\ \gamma' \rangle \;\middle|\; \begin{array}{l} \delta \textit{ is a field of } \tau, \textit{ and} \\ \gamma \in lookup(\tau,\ \delta,\ s.\alpha), \textit{ and} \\ \gamma' \in lookup(\tau,\ \delta,\ t.\beta) \end{array} \right\}
$$

Function *lookup*, defined below, uses an auxiliary function named *commonInitialSeq*. Function *commonInitialSeq* takes two arguments: a type τ and an object $\mathtt{t}.\beta$. If there is some field δ such that (i) $normalize(\mathtt{t}.\delta) = \mathtt{t}.\beta$, and (ii) τ and $\mathtt{t}.\delta$ have a non-empty common initial sequence, then *commonInitialSeq* returns that sequence (i.e., the set of pairs of fields $\langle \gamma,\ \delta \| \gamma' \rangle$, where γ is a field in τ, and γ' is the corresponding field in $\tau_{t.\delta}$). Otherwise, *commonInitialSeq* returns the empty set.

$$
\begin{array}{l}
lookup(\tau, \alpha, t.\beta) = \\
\quad \textbf{if } \textit{there is a pair } \langle \alpha,\ \alpha' \rangle \textit{ in } commonInitialSeq(\tau,\ t.\beta) \\
\quad \textbf{then } \{normalize(t.\alpha')\} \\
\quad \textbf{else}\ \ \textbf{let}\ \ \gamma \textit{ be the first field of t that follows the common} \\
\qquad\qquad\quad \textit{initial sequence of } \tau \textit{ and } t.\beta, \\
\qquad\qquad\quad \textit{or } \beta \textit{ itself if that sequence is empty} \\
\qquad\quad \textbf{in} \quad \left\{ normalize(t.\delta) \;\middle|\; \begin{array}{l} \delta = \gamma, \textit{ or} \\ \delta \in followingFields(\tau_t,\ \gamma) \end{array} \right\}
\end{array}
$$

Function *lookup* first determines whether field α is part of a common initial sequence of τ (the type that the dereferenced pointer is declared to point to) and $\mathtt{t}.\beta$ (the object that the pointer actually points to). If so, it simply returns the corresponding field (normalized). If not, it returns all fields in `t`, starting with the first field that follows the common initial sequence.

This definition of *lookup* can be illustrated using the following example:

```
1:  struct S { int s1; int s2; int s3; } *p;
2:  struct T { int t1; int t2; char t3; int t4; } t;
3:  int *x, *y;
4:  p = (struct S *)&t;
5:  x = &(*p).s2;
6:  y = &(*p).s3;
```

When Statement 5 is processed, the call *lookup*(`struct S, s2,` *normalize*(`t`)) is made. This leads to the call *commonInitialSeq*(`struct S, t.t1`). The field `t.t1` itself does not have a common initial sequence with `struct S`; however, `t.t1` is the first field of `t`, and `t` does have a common initial sequence with `struct S`. The corresponding pairs of fields in that sequence are: {⟨`s1,t1`⟩, ⟨`s2,t2`⟩}. Since there is a pair with `s2` in this set, *lookup* returns { *normalize*(`t.t2`) } = { `t.t2` }.

When Statement 6 is processed, the call *lookup*(`struct S, s3,` *normalize*(`t`)) is made. The same call to *commonInitialSeq* is made and the same set of pairs is returned. There is no pair with `s3` in that set; therefore, *lookup* returns all of the fields of `t` starting with `t.t3`, the first field of `t` that follows the common initial sequence. In this case, that means that *lookup* returns the two (normalized) fields: {`t.t3, t.t4`}.

5 Experiments

We have implemented our pointer-analysis framework in C++ using the SUIF compiler infrastructure [WFW+94] (i.e., our code works on the intermediate form produced by the SUIF front-end). Calls to library functions are handled by providing summaries of the potential pointer assignments in each library function (we used copies of the summaries that were used in the experiments reported in [WL95]). The basic algorithm builds a graph with one node for each variable or temporary, and one edge for each normalized assignment statement in the program. It then uses the rules of inference to add additional edges, each of which represents one points-to fact. We have also implemented the four versions of *normalize*, *lookup*, and *resolve* defined in Sections 4.2.2 and 4.3 to produce four instances of the framework.

Tests were carried out on an Ultra 10 with 256MB of RAM and 1.1G of swap space. We used 20 C programs (including Gnu Unix utilities, Spec benchmarks, and programs used for benchmarking by Landi [LRZ93] and by Austin [ABS94]). We found that in 8 of the test programs all structure accesses and copies were done with variables of the correct types (i.e., casting of structures and structure pointers was not an issue).

Figure 3 provides information about each of the test programs: the number of lines of source code and the number of normalized assignment statements. The 8 test programs that did not involve structure casting are shown first (sorted by number of lines), followed by the 12 test programs that did involve structure casting. Figure 3 also provides information about calls to *lookup* and *resolve* in the "Collapse on Cast" and "Common Initial Sequence" algorithms.[7] Columns 5 and 6 give the percentage of the total number of such calls that involved structures, and columns 7 and 8 give the percentage of those calls in which the types did not match (i.e., casting was involved). (Columns are not included for the "Collapse Always" or "Offsets" algorithms.

[7] The definitions of *resolve* in Sections 4.3.2 and 4.3.3 include calls to *lookup*; however, the implementations are instrumented so that calls to *lookup* from *resolve* are not counted.

	Program	# Lines	# Norm Assigns	% of Total Calls to *lookup* or *resolve* That Involve Structs		% of Struct Calls to *lookup* or *resolve* That Involve Casts	
				Collapse On Cast	Common Init. Seq.	Collapse On Cast	Common Init. Seq.
Programs with no structure casting	anagram	647	1,381	3.02	3.02	0.00	0.00
	ks	782	1,782	16.28	16.28	0.00	0.00
	ansitape	1,741	2,485	0.36	0.36	0.00	0.00
	129.compress	1,934	1,665	0.20	0.20	0.00	0.00
	ft	1,986	1,775	12.09	12.09	0.00	0.00
	triangle	2,006	7,807	5.77	5.77	0.00	0.00
	yacr2	3,979	4,941	4.15	4.15	0.00	0.00
	099.go	29,244	42,522	0.66	0.66	0.00	0.00
Programs with structure casting	football	2,353	9,784	13.47	13.58	0.98	0.98
	twig	3,188	4,559	29.16	29.97	86.74	86.74
	agrep	3,990	11,198	12.04	12.04	91.27	91.27
	simulator	4,653	6,114	2.97	2.98	5.74	5.74
	bc	7,295	7,941	12.10	12.10	8.21	8.21
	ispell-4.0	7,373	9,018	2.31	2.31	32.45	32.45
	130.li	7,597	8,210	10.46	10.46	99.78	99.78
	gzip-1.2.4	8,163	9,447	6.53	6.63	51.70	51.70
	bison-1.22	10,727	12,854	15.01	15.01	72.39	72.39
	less-177	10,930	6,802	18.22	10.18	75.62	31.12
	flex-2.4.7	13,541	11,112	2.55	2.55	60.39	60.39
	124.m88ksim	19,227	25,515	20.22	20.22	96.71	96.71

Figure 3: Information about the 20 test programs.

The former treats structures as single objects regardless of whether or not casting is involved, and the latter uses offsets rather than types and field names, so neither includes a test for type mismatch.)

Note that a call to *lookup* corresponds to a use of Inference Rule 2, and a call to *resolve* corresponds to a use of Inference Rule 3, 4, or 5 (each use of a rule may or may not generate new points-to facts). Therefore, the information in columns 5–8 provides some insight into how important structures and casting are in pointer analysis. In particular, note that many of the calls to *lookup* and *resolve* that involved structures also involved a type mismatch, which indicates that casting played a role (either directly, or as a transitive effect as the algorithm propagated points-to information).

The two most significant results of our experiments are described below.

Distinguishing individual fields of structures is important. Figure 4 reports the average points-to set sizes — where the average is taken across the points-to set sizes of all static instances of dereferenced pointers — for the 12 test programs with structure casting. For each test program, the average points-to set sizes are shown for each of the four pointer-analysis algorithms. (To be comparable to the other data, when a pointer `p` is dereferenced, if the "Collapse Always" algorithm includes the points-to fact $pointsTo(p,\ s)$, and `s` is a structure, then that fact is expanded to the set of facts $pointsTo(p,\ s.\alpha)$, for all fields α in `s`.) Notice that in six cases, the sets produced by the "Collapse Always" algorithm are at least twice as large as the sets produced by the other algorithms. (In the worst case, `bc`, the set size is more than 10 times larger.) Since other static analyses require information about what may be accessed by each pointer dereference, it is likely that the sizes of these sets will have a significant impact on the precision of the results of subsequent analyses (e.g., see [SH97a]).

There is relatively little penalty for portability. Figure 4 also supports the claim that portability is not too costly in terms of precision. The average points-to set sizes for the two portable pointer-analysis algorithms that handle casting ("Collapse on Cast" and "Common Initial Sequence") are usually very close to the sizes for the non-portable algorithm ("Offsets"). In more than half of the cases, the difference is less than 2%. In the worst case for "Collapse on Cast" (`less-177`), the average points-to set size for the "Collapse on Cast" algorithm is 63.5% larger than that of the "Offsets" algorithm. The worst case for the "Common Initial Sequence" algorithm (the most precise of the portable algorithms) is `twig`, for which its average points-to set size is 45.7% larger than for the "Offsets" algorithm.

Figure 5 shows the analysis times for each of the algorithms, normalized to that of the "Offsets" algorithm (i.e., the height of the bar for each algorithm is the time for that algorithm divided by the time for the "Offsets" algorithm). The numbers below the bars are the CPU times (user+system time) for the "Offsets" algorithm. Note that in most cases the times for the three algorithms that handle casting ("Collapse on Cast", "Common Initial Sequence", and "Offsets") are about the same (within less than 50% of each other in all but two cases). In the worst case (`less-177`), the time for the "Collapse on Cast" algorithm is 4 times that for the "Offsets" algorithm. On the other hand, for `flex-2.4.7`, the portable algorithms are faster than the "Offsets" algorithm.

Figure 6 shows the total number of points-to edges generated by each algorithm, normalized to that of the "Offsets" algorithm. Since the four algorithms are instances of the same framework, this number is a good reflection of

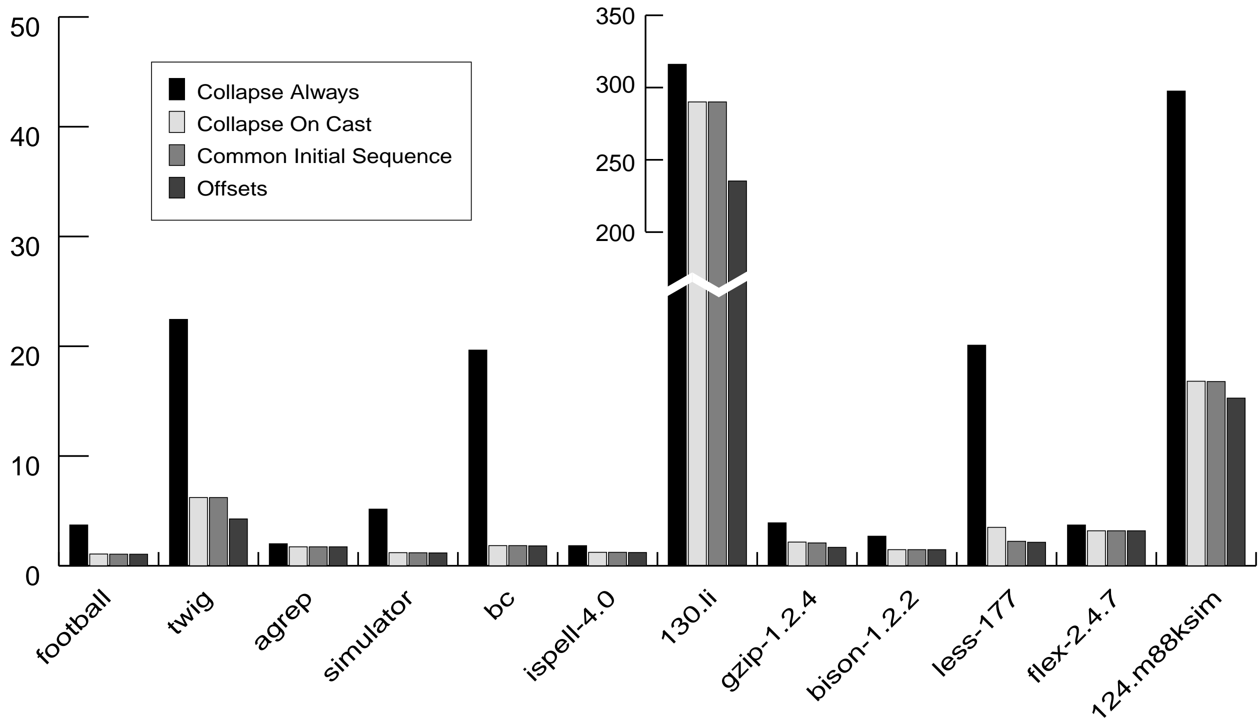


Figure 4: Average points-to set size of a dereferenced pointer.

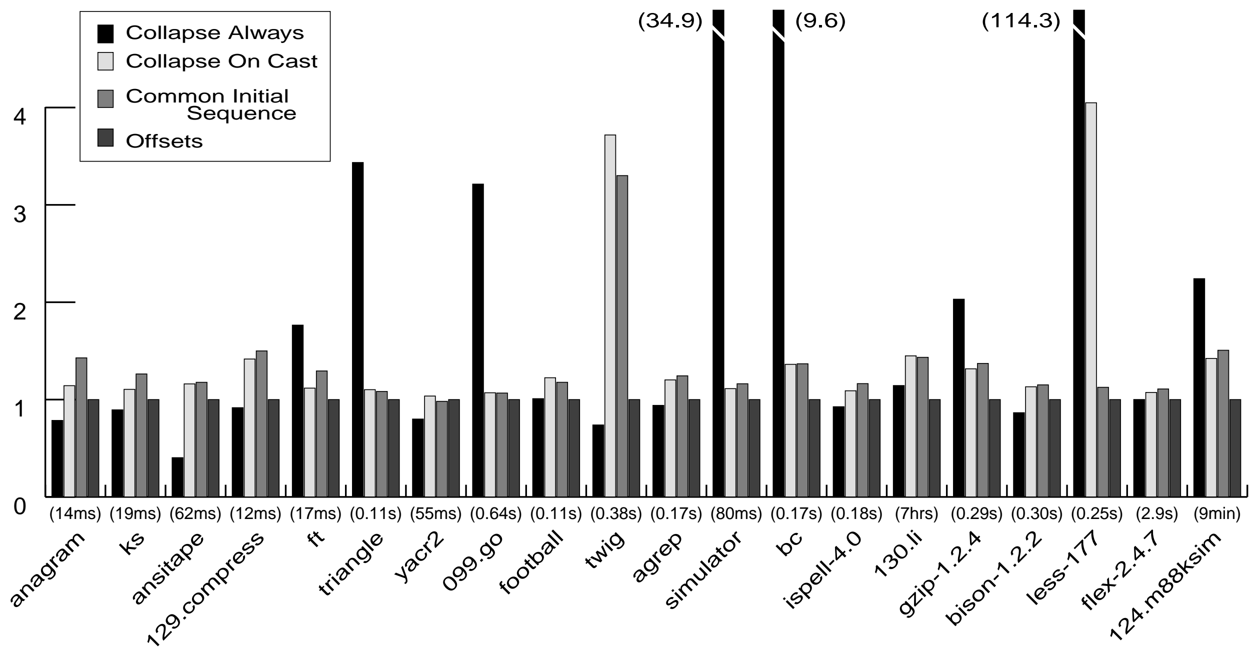


Figure 5: Analysis-time ratios (normalized to the times for the "Offsets" algorithm, which is given below the bars).

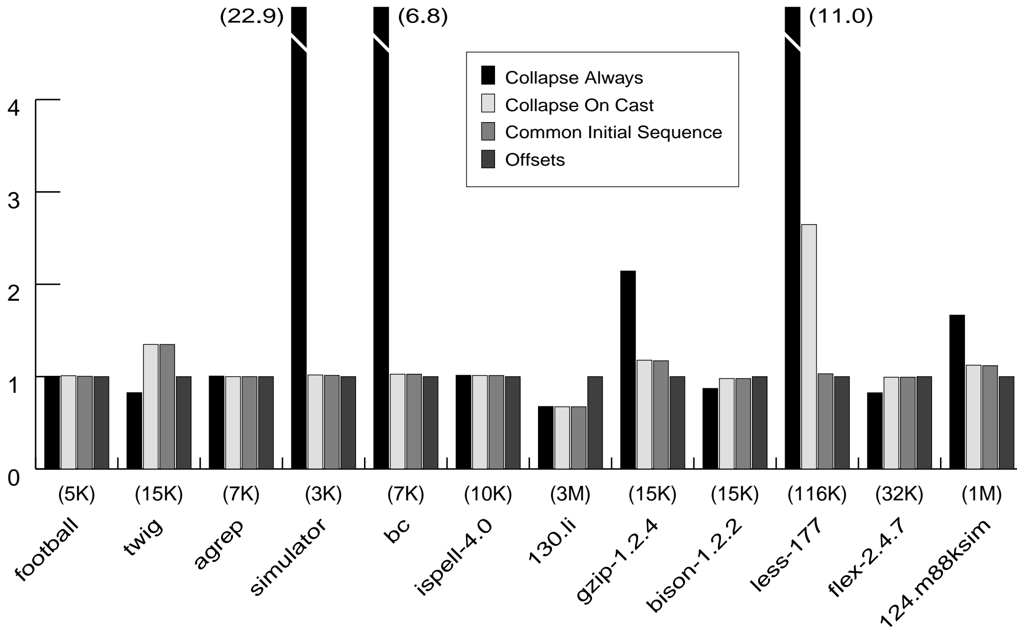


Figure 6: Ratio of the total number of points-to edges (normalized to the number of points-to edges for the "Offsets" algorithm, which is given below the bars).

their relative space requirements.[8] In all but three cases, the points-to set sizes for the two portable algorithms "Collapse on Cast" and "Common Initial Sequence" are within 18% of the sizes for the non-portable "Offsets" algorithm. In their respective worst cases, "Collapse on Cast" has 2.6 times as many points-to edges (`less-177`) and "Common Initial Sequence" has 35% more points-to edges than "Offsets". On the other hand, for `130.li`, the portable algorithms have 33% fewer points-to edges than the "Offsets" algorithm. This is due to the "Offsets" algorithm introducing nodes to represent offsets within structures that do not correspond to real fields. Overall, then, we conclude that the penalty for portability in terms of space is not too large.

6 Related Work

The most closely related work involves algorithms that distinguish fields of structures and handle casting. This includes the work of Steensgaard [Ste96a], Ryder et al. [Ryd98], and Wilson and Lam [WL95]. Of these, only Steensgaard provides a portable algorithm; the others rely on platform-specific information. The "Common Initial Sequence" approach uses the same basic idea as Steensgaard's algorithm. However, Steensgaard's version differs from the "Common Initial Sequence" algorithm in that his algorithm not only distinguishes fields of structures and handles casting, but also keeps the running time of the algorithm as close to linear as possible by using other approximations similar to those used in his original work on flow-insensitive pointer analysis [Ste96b]. (However, the approximations that ensure fast running time can also lead to much less precise results than those provided by the "Common Initial Sequence" algorithm.)

The "Offsets" approach is similar to the one used by Ryder et al. in the alias-analysis algorithms that they implemented as part of their work on the modification side-effects problem [SRLZ98]. Essentially, everything is encoded in a base-offset manner; casting that overlaps fields causes more approximate information to be kept about aliases, while casting that "matches up" fields preserves alias accuracy.

The "Offsets" approach is also similar to the approach used by Wilson and Lam in [WL95]; however, they maintain a "stride" for each object in addition to its offset. This is important when pointer arithmetic is performed on a pointer to an array that is inside a structure. In that case, although the pointer may be moved off of the end of the array, it cannot point to an arbitrary field of the enclosing structure; since pointer arithmetic adds (or subtracts) a value equal to the size of an array element, the pointer can only point to fields at offsets that are some multiple of that size away from the ends of the array.

In addition, there have been algorithms that distinguish fields of structures, but do not handle casting. This includes the work of Choi et al. [CBC93], Burke et al. [BCCH94], Andersen [And94], Emami et al. [EGH94], and Tonella [Ton97].

We cannot categorize the work of Ruf [Ruf95]. While it appears that fields of structures are distinguished, this aspect of the analysis is only mentioned briefly, and it is not possible to tell whether casting is handled.

[8] The fact that the number of edges added by the "Collapse Always" algorithm is sometimes smaller than those of the other three algorithms does *not* mean that it is sometimes more precise: as noted earlier, because the "Collapse Always" algorithm does not distinguish fields of structures, it may represent many points-to facts for the fields of a structure s with a single edge.

References

[ABS94] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[BCCH94] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Language and Compilers for Parallel Computing, 7th International Workshop*, LNCS 892, pages 234–250. Springer-Verlag, August 1994.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interporcedural computation of pointer-induced aliases and side effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[ISO90] ISO/IEC. *International Standard ISO/IEC 9899, Programming Languages - C. 1st Ed.* 1990.

[LRZ93] W. Landi, B. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[Ryd98] B. Ryder. personal communication, September 1998.

[SH97a] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *4th International Symposium on Static Analysis*, LNCS 1302. Springer-Verlag, September 1997.

[SH97b] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, January 1997.

[SRLZ98] P. Stocks, B. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, March 1998.

[Ste96a] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*, April 1996.

[Ste96b] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Ton97] P. Tonella. Points-to analysis for program understanding. In *Proceedings of the International Workshop on Program Comprehension*, May 1997.

[WFW+94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29(12), pages 31–37, December 1994.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.