Efficient Field-Sensitive Pointer Analysis of C

DAVID J. PEARCE
Victoria University of Wellington, New Zealand
and
PAUL H.J. KELLY
Imperial College London, United Kingdom
and
CHRIS HANKIN
Imperial College London, United Kingdom

The subject of this paper is flow- and context-insensitive pointer analysis. We present a novel approach for precisely modelling `struct` variables and indirect function calls. Our method emphasises efficiency and simplicity and is based on a simple language of set constraints. We obtain an $O(v^4)$ bound on the time needed to solve a set of constraints from this language, where v is the number of constraint variables. This gives, for the first time, some insight into the hardness of performing field-sensitive pointer analysis of C. Furthermore, we experimentally evaluate the time versus precision trade-off for our method by comparing against the field-insensitive equivalent. Our benchmark suite consists of 11 common C programs ranging in size from 15,000 to 200,000 lines of code. Our results indicate the field-sensitive analysis is more expensive to compute, but yields significantly better precision. In addition, our technique has been integrated into the latest release (version 4.1) of the GNU Compiler GCC. Finally, we identify several previously unknown issues with an alternative and less precise approach to modelling `struct` variables, known as field-based analysis.

Categories and Subject Descriptors: F.3.2 [**Semantics of Programming Languages**]: Program Analysis

General Terms: Algorithms, Theory, Languages, Verification

Additional Key Words and Phrases: Set-Constraints, Pointer Analysis

This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through a PhD studentship and grant number GR/R15566.

A preliminary version of this paper was presented at the international Workshop on Program Analysis for Software Tools and Engineering (PASTE'04), 2004 [Pearce et al. 2004a]. This work provides a more detailed examination of the subject material, including a larger experimental study, more examples and a complexity analysis of our algorithm.

Authors' Addresses: David J. Pearce, School of Mathematics, Statistics and Computer Science, Victoria University of Wellington, Wellington, New Zealand, e-mail: david.pearce@mcs.vuw.ac.nz; Paul H.J. Kelly, Department of Computing, 180 Queen's Gate, South Kensington Campus, Imperial College London, SW7 2AZ, United Kingdom, e-mail: p.kelly@imperial.ac.uk; Chris Hankin, Department of Computing, 180 Queen's Gate, South Kensington Campus, Imperial College London, SW7 2AZ, United Kingdom, e-mail: c.hankin@imperial.ac.uk.

1. INTRODUCTION

Pointer analysis is the problem of determining statically what the pointer variables in a program may target. Consider the following C program:

```
void foo() {
  int *p,*q,a,b;
  p = &a;
  if(...) q = p;
  else q = &b;
  /* point P1 */
  ...
}
```

Here a pointer analysis concludes that, during any execution of the program, the following will hold at P1: `p` *points-to* `a` and `q` *points-to* `a` or `b`. We write $p \mapsto \{a\} \wedge q \mapsto \{a, b\}$ to state this formally, where $\{a\}$ and $\{a, b\}$ are the *target sets* of p and q respectively. A solution is *sound* if the target set obtained for each variable contains all its actual runtime targets. Thus, $q \mapsto \{a\}$ is an unsound solution for the above because q can also point to b. A solution is *imprecise* if an inferred target set is larger than necessary and the superfluous targets are called *spurious*. So, for the above example, an imprecise (but sound) solution for p is $p \mapsto \{a, b\}$. In general, obtaining a perfectly precise and sound solution is undecidable [Landi 1992b; Ramalingam 1994] and, in practice, even relatively imprecise information is expensive. The applications of pointer analysis are many, but perhaps the most important uses today are in Compilers and Software Engineering.

Compilers. Modern superscalar and VLIW processors require sufficient Instruction Level Parallelism (ILP) to reach peak utilisation. For this reason, exposing ILP through instruction scheduling and register allocation is a crucial role of the compiler. This task is complicated by the presence of instructions which indirectly reference memory, since their data dependencies are not known. For languages such as C/C++, this problem is particularly acute because pointer variables (the main source of indirect memory references) can target practically every memory location without restriction. Therefore, to achieve maximum pipeline throughput, the compiler must rely on pointer analysis to disambiguate indirect memory references.

Automatic parallelisation is another example of how the compiler can achieve a speedup by exposing parallelism within the program. This type of transformation is performed at a higher level than those for ILP and, hence, larger gains are possible. Indeed, much success has been achieved through automatic parallelisation of numerical FORTRAN programs (e.g. [Padua et al. 1980; Wolfe 1982; Padua and Wolfe 1986; McKinley 1994; So et al. 1998]). However, similar results have yet to be seen on programs written in C/C++ or Java. The main reason for this is simply that, without precise information about pointer targets, compilers for these languages cannot perform automatic parallelisation safely.

Finally, pointer analysis finds many other important uses within the compiler. In particular, it often enables traditional optimisations (e.g. common sub-expression elimination) to be applied at places which would otherwise be deemed unsafe.

Software Engineering. Reliability of large software systems is a difficult problem facing software engineering. Subtle programming errors, which go undetected during testing, can have disastrous consequences. An historic example is the 1988 worm which caused havoc by infecting large parts of the internet [Eichin and Rochlis 1989]. The worm replicated by exploiting a *buffer overrun* vulnerability in the `fingerd` daemon, which existed through programming error. This type of mistake is usually associated with the misuse of pointers and accounts for the majority of security holes in modern software [Wagner et al. 2000]. One approach to tackling these problems is to construct tools which either aid program understanding or, in some way, check for programming error. Examples of the former include program slicers (e.g. [Reps and Turnidge 1996; Ball and Horwitz 1993; Harman et al. 2003; Binkley 1998]), static debuggers (e.g. [Bourdoncle 1993a; Flanagan 1997]) and software visualisers (e.g. [Jones et al. 2002; Systä et al. 2000; Myers 1986; Reiss 1997]). Examples of tools which check for programming error can usually be divided into two camps: static analysis tools (e.g. [Dor et al. 2003; Flanagan et al. 2002; Blanchet et al. 2002; 2003; Wagner et al. 2000]) and model checkers (e.g. [The Vis Group 1996; Godefroid 1997; Alur et al. 1998; Holzmann 1997; Henzinger et al. 2003]). The former generally operate on programs directly, whilst the latter operate on abstract models of programs. In languages such as C/C++ and Java, pointer analysis is invariably found in all these tools where it forms a foundation for other analyses.

The focus of this work is on developing efficient algorithms for *field-sensitive* pointer analysis of C. The basic idea is that the precision of a pointer analysis can be improved by distinguishing the fields of `struct` variables. Almost all previous pointer analyses for C fail to do this properly for a variety of reasons: either they treat fields of a `struct` variable as one (e.g. [Fähndrich et al. 1998; Hasti and Horwitz 1998; Hind and Pioli 1998; Shapiro and Horwitz 1997]); or, they distinguish fields, but lose the ability to differentiate separate *instances* of a `struct` (e.g. [Heintze and Tardieu 2001b; Andersen 1994; Ghiya et al. 2001]); or, they are unable to analyse all but the simplest of programs (e.g. [Wilson and Lam 1995; Emami et al. 1994; Landi 1992a]). The main contributions of this work are:

(1) An extension to the language of set-constraints, which elegantly formalises a field-sensitive pointer analysis for the C language. As a byproduct, function pointers are supported for free with this mechanism.

(2) For the first time, an $O(v^4)$ bound on the time needed for field-sensitive pointer analysis of C is obtained, where v is the number of nodes in the constraint graph.

(3) The largest experimental investigation into the trade-offs in time and precision of field-insensitive and -sensitive analyses for C. Our benchmark suite contains 11 common C programs, ranging in size from 15,000 to 200,000 lines of code.

(4) The identification of several previously unknown problems with a similar approach, known as *field-based* pointer analysis, when applied to the analysis of C programs.

Our technique is not the first field-sensitive, constraint-based pointer analysis for C — previous work has covered this (see [Yong et al. 1999; Chandra and Reps 1999a]). Our claim then, is that we go beyond their initial treatment by considering efficient implementation and some important algorithmic issues not adequately addressed before. In particular, our technique is designed specifically to work with the *points-to* or solution sets implemented as integer sets. This permits the use of data structures supporting efficient set union, such as bit vectors or sorted arrays, which are necessary for scalable pointer analysis. Furthermore, we are the first to obtain a complexity bound on the time needed to solve this problem. In doing this, we find that the field-sensitive pointer analysis problem is fundamentally harder for C than for Java. Indeed, while several previous field-sensitive pointer analyses for Java are known (see e.g. [Rountev et al. 2001; Lhoták and Hendren 2003; Whaley and Lam 2002]), these turn out to be insufficient when analysing C.

2. CONSTRAINT-BASED POINTER ANALYSIS

In this paper, we consider pointer analyses which are formulated using a general approach to program analysis known as *set constraints* (or sometimes *inclusion constraints*). Set-constraint systems are not new and can be traced back to [Reynolds 1969; Jones and Muchnick 1981]. Through the work of Heintze, Aiken and others, they have recently become a well established approach to program analysis (e.g. [Heintze 1994; Aiken 1999; 1994; Aiken and Wimmers 1993; 1992]). Applications in this field include control-flow analysis (e.g. [Heintze and McAllester 1997b; 1997a]), debugging (e.g. [Wagner et al. 2000; Flanagan 1997]) and more. The first example of a pointer analysis formulated using set constraints was that of Andersen [Andersen 1994] and, since then, many have followed (e.g. [Foster et al. 1997; Fähndrich et al. 1998; Heintze and Tardieu 2001b; Ghiya et al. 2001; Rountev et al. 2001; Lhoták and Hendren 2003; Pearce et al. 2004b]). Set constraints are not the only way of performing pointer analysis and other techniques, particularly *abstract interpretation* (e.g. [Emami et al. 1994; Wilson and Lam 1995; Landi 1992a; Hind et al. 1999]) and *unification* (e.g. [Steensgaard 1996; Das 2000; Das et al. 2001]), are popular. Understanding the differences between these different approaches is not easy, although a common view holds that abstract interpretation is precise but slow, while unification is fast but imprecise. Set constraints lie somewhere in the middle — they are more precise than unification, but still capable of analysing programs with a hundred thousand lines of code or more (e.g. [Fähndrich et al. 1998; Heintze and Tardieu 2001b; Lhoták and Hendren 2003; Pearce et al. 2004b]).

We now present our set-constraint formulation of the pointer analysis problem, which is based upon the following language:

$$p \supseteq q \mid p \supseteq \{q\} \mid p \supseteq *q \mid \ *p \supseteq q$$

Here, p and q are constraint variables and $*$ is the usual dereference operator. We can think of each constraint variable as containing the set of variables it points to. Thus, $p \supseteq \{x\}$ states *p may point to x.* Those involving "$*$" are referred to as *complex constraints.* Those knowledgeable about set constraints will notice a lack of

$$[trans]\ \frac{p \supseteq \{q\} \quad r \supseteq p}{r \supseteq \{q\}} \quad [deref_1]\ \frac{p \supseteq *q \quad q \supseteq \{r\}}{p \supseteq r} \quad [deref_2]\ \frac{*p \supseteq q \quad p \supseteq \{r\}}{r \supseteq q}$$

Fig. 1. An inference system for pointer analysis

general constructors and projection. Essentially, we have simplified the traditional set-constraint system by specialising it to our problem domain.

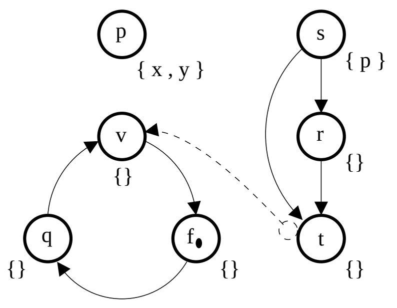
To perform the analysis we first translate the source program into the set-constraint language, by mapping each source variable to a unique constraint variable and converting assignments to constraints. Then, we solve the constraints to find a least solution, which can be formalised as deriving all possible facts under the inference system of Figure 1. An example program, along with its translation and derived solution, is shown in Figure 2. Regarding the hardness of this problem, it is well-known that a set of t constraints can be solved in $O(t^3)$ time [Melski and Reps 1997; Heintze 1994]. For completeness, we provide a similar proof here:

LEMMA 2.1. *A set of t constraints can be solved in $O(t^3)$ time under the inference system of Figure 1.*

PROOF. This result stems from two facts: firstly, the total number of trivial constraints generated (i.e. those of the form $p \supseteq \{q\}$) is bounded by $O(v^2)$, where v is the number of variables; secondly, at most $O(v^2)$ simple constraints (i.e. those of the form $p \supseteq q$) are possible. From these it follows that, in the worse case, the *trans* rule must be applied $O(v^3)$ times, since at most v trivial constraints can be propagated across each simple constraint. Note, the *deref* rules need only be applied $O(tv)$ times, since each dereferenced variable has $O(v)$ targets and there are $O(t)$ complex constraints. Since v is $O(t)$, we obtain an $O(t^3)$ bound on worst-case solving time. □

3. SOLVING THE ANALYSIS

Thus far, we have said the aim is to derive all possible facts using the inference system of Figure 1. To do this efficiently, we formulate the constraints as a directed graph, where each variable is represented by a unique node and each constraint $p \supseteq q$ by an edge $p \leftarrow q$. In addition, we associate with each variable n a set $Sol(n)$, into which the *points-to* solution for n is accumulated. Thus, for the example of Figure 2 we obtain the following graph:

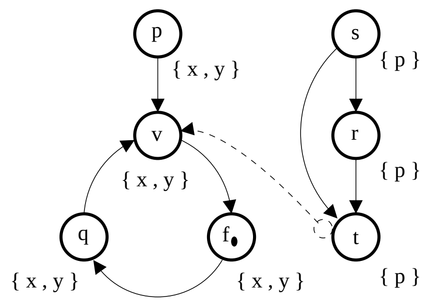


`int *f(int *v) { return v; }`	(1)	$f_\bullet \supseteq v$
`void g() {`		
` int x,y,*p,*q,**r,**s,**t;`		
` s=&p;`	(2)	$s \supseteq \{p\}$
` if(...) {`		
`  p=&x;`	(3)	$p \supseteq \{x\}$
`  r=s;`	(4)	$r \supseteq s$
`  t=r;`	(5)	$t \supseteq r$
` } else {`		
`  p=&y;`	(6)	$p \supseteq \{y\}$
`  t=s;`	(7)	$t \supseteq s$
` }`		
` q=f(*t);`	(8)	$v \supseteq *t$
	(9)	$q \supseteq f_\bullet$
` f(q);`	(10)	$v \supseteq q$
`}`		

(12)	$r \supseteq \{p\}$	$(\mathit{trans}, 2+4)$
(13)	$t \supseteq \{p\}$	$(\mathit{trans}, 5+12)$
(14)	$v \supseteq p$	$(\mathit{deref}_1, 8+13)$
(15)	$v \supseteq \{x\}$	$(\mathit{trans}, 3+14)$
(16)	$v \supseteq \{y\}$	$(\mathit{trans}, 6+14)$
(17)	$f_\bullet \supseteq \{x\}$	$(\mathit{trans}, 1+15)$
(18)	$f_\bullet \supseteq \{y\}$	$(\mathit{trans}, 1+16)$
(19)	$q \supseteq \{x\}$	$(\mathit{trans}, 9+17)$
(20)	$q \supseteq \{y\}$	$(\mathit{trans}, 9+18)$

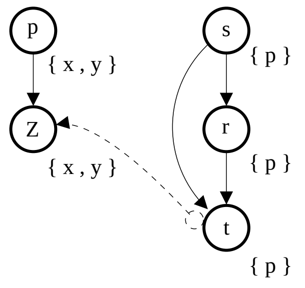
Fig. 2. An example illustrating how a simple program is translated into constraints and then solved. The initial constraint set (shown above the line) is generated directly from the program source. The inference rules of Figure 1 are then applied to obtain the complete derivation shown below the line. We assume for simplicity that variable names are unique, which can easily be achieved in practice by augmenting with the name of the enclosing method. Also, $f_\bullet$ represents the return value of f. The final solution for each variable is the smallest set satisfying the fully derived constraint system. Thus, in the example, constraints 19+20 imply the smallest solution for q is $\{x, y\}$. Therefore, our analysis concludes $q \mapsto \{x, y\}$ holds at all points in the program. The key point here is that *we must derive all facts in order to make a sound conclusion.* This is because we cannot be sure that the solution for a given variable is complete until all facts are known. A number of *demand-driven* systems have also been proposed (e.g. [Fähndrich et al. 2000; Heintze and Tardieu 2001a; Vivien and Rinard 2001; Sridharan et al. 2005; Saha and Ramakrishnan 2005; Kodumal and Aiken 2005]) which can often avoid deriving all facts (although not in the worst-case). For the purposes of this work, however, we take the more traditional approach of assuming that all facts must be derived and note that it would be interesting future work to extend our analysis to be demand driven.

In the above, we have placed $Sol(n)$ for each variable n below its corresponding node. This initially contains x iff $n \supseteq \{x\}$ is in the initial constraint set. The dashed edge, known as a *complex edge*, represents the complex constraint $v \supseteq *t$, with the circle indicating which end is dereferenced. At this point, the constraints can be solved by repeatedly selecting a (non-complex) edge $x \to y$ and merging $Sol(x)$ into $Sol(y)$ until no change is observed. This is often referred to as *converging* or *reaching a fixpoint*. During this process, new edges arising from the complex constraints must be added to the graph. To see why, recall that our example contained the complex constraint $v \supseteq *t$. We know that, initially $Sol(t) = \emptyset$, but at some point during the analysis $Sol(t) \supseteq \{p\}$. Clearly then, there is a dependence from p to v and this could not have been known at graph construction time. Therefore, the edge $p \to v$ must be added as the solution for t becomes available. Thus, solving the above constraint graph gives:



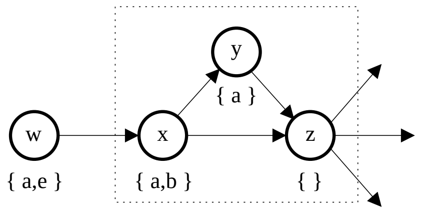
Thus, a new edge has been added because of the constraint $v \supseteq *t$. When reading the constraint graph, we can determine what edges will be added by examining the solution at the circle end of a complex edge: edges will be added between the nodes in that solution and the node at the other end of the complex edge.

A useful observation is that nodes in the same cycle (when ignoring complex edges) always end up with the same solution [Fähndrich et al. 1998; Heintze and Tardieu 2001b]. So, in our example, nodes v, $f_\bullet$ and q have the same final solution. Therefore, we can simplify the graph by collapsing them into a single representative, giving:



The gain from this simplification comes from time saved by not propagating targets between internal nodes. However, identifying these cycles is complicated by the dynamic nature of the graph as edges added during solving may introduce new cycles. Therefore, to exploit all such simplification opportunities we must be able to efficiently determine when a newly added edge introduces a cycle.

In a similar vein, a technique we refer to as *subsumed node compaction* can also help simplify the constraint graph. The idea, originally suggested by Rountev and Chandra [Rountev and Chandra 2000], is illustrated by the following:



Here, x, y, z must have the same solution and, hence, can be collapsed into one. Note, we assume here that y and z have not had their address taken and are not targeted by a constraint such as $y \supseteq *p$. Rountev and Chandra provided a linear time algorithm for detecting such opportunities in the constraint graph. Note, unlike with cycle detection, new opportunities for applying this optimisation cannot arise during the analysis.

The approach to solving set constraints we have presented is sometimes called *Standard Form (SF)* [Aiken and Wimmers 1993]. An alternative to this, known as *Inductive Form (IF)*, is often described in the literature as a sparser and more efficient representation [Su et al. 2000; Rountev et al. 2001]. In general, we find there is little evidence to support this claim that IF is more efficient than SF: the only experimental study is [Fähndrich et al. 1998]. This appears to show that inductive form has an advantage over standard form. Unfortunately, this result remains inconclusive because the cycle detection algorithm used did not identify and collapse all cycles for efficiency reasons. Thus, it happens that under inductive form the algorithm consistently collapses more cycles, giving it an apparent advantage. However, we have since developed more efficient cycle detection algorithms which can collapse all cycles under standard form, thereby eliminating this distinction between them [Pearce et al. 2003; 2004b]. Therefore, we cannot draw concrete conclusions about the relative efficiency of either approach and, in general, equal success has been achieved (e.g. [Heintze and Tardieu 2001b; Lhoták and Hendren 2003] versus [Rountev et al. 2001; Fähndrich et al. 1998]). For the purposes of this paper, we are concerned only with Standard Form and, in the remainder, it is assumed.

3.1 Field-Sensitivity

So far, we have not indicated how `struct` variables should be handled by our analysis and there are three approaches: *field-insensitive*, where field information is discarded by modelling each aggregate with a single constraint variable; *field-based*, where one constraint variable models all *instances* of a field; and finally, *field-sensitive*, where a unique variable models each field of an aggregate. The following example aims to clarify this:

```
typedef struct { int *f1; int *f2; } aggr;
```

`aggr a,b;`	(field-insensitive)	(field-based)	(field-sensitive)
`int *c,d,e,f;`			
`a.f1 = &d;`	$a \supseteq \{d\}$	$f1 \supseteq \{d\}$	$a_{f1} \supseteq \{d\}$
`a.f2 = &f;`	$a \supseteq \{f\}$	$f2 \supseteq \{f\}$	$a_{f2} \supseteq \{f\}$
`b.f1 = &e;`	$b \supseteq \{e\}$	$f1 \supseteq \{e\}$	$b_{f1} \supseteq \{e\}$
`c = a.f1;`	$c \supseteq a$	$c \supseteq f1$	$c \supseteq a_{f1}$
Conclude	$c \mapsto \{d, f\}$	$c \mapsto \{d, e\}$	$c \mapsto \{d\}$

Here, the field-insensitive and field-based solutions are imprecise in different ways. In general, their relative precision depends upon the program in question. For example, analysing a program with many aggregates of the same type would likely be better done with a field-insensitive analysis. This is because the field-based analysis will combine the solution for each instance of a given field into one, thereby losing a lot of information. In contrast, if the program has a small number of aggregates with a large number of fields then the opposite will be true. By comparison, the field-sensitive approach does not suffer such problems and obtains the best precision in either situation.

Most previous set constraint-based pointer analyses are either field-insensitive (e.g. [Foster et al. 2000; Hasti and Horwitz 1998; Hind and Pioli 2000; Fähndrich et al. 1998]) or field-based (e.g. [Andersen 1994; Heintze and Tardieu 2001b; Ghiya et al. 2001]). Algorithms for field-sensitive analysis are harder to develop and implement, which may explain why they have received less attention. Furthermore, the majority of those which have been developed are for the analysis of Java [Rountev et al. 2001; Liang et al. 2001; Whaley and Lam 2002; Lhoták and Hendren 2003]. For C, only three field-sensitive pointer analyses are known [Yong et al. 1999; Chandra and Reps 1999a; Johnson and Wagner 2004] and this might stem from the fact that, as will be shown in Section 4, it is a fundamentally harder problem than for Java.

3.2 Indirect Function Calls

In the literature, function pointers are either dealt with in ad hoc ways (e.g. [Heintze and Tardieu 2001b; Lhoták and Hendren 2003]) or through the *lam* constructor (e.g. [Foster et al. 2000; 1997]). The latter uses a special rule for function application:

$$[func]\ \frac{\begin{array}{c}*p(w_1,\ldots,w_n)\\ p \supseteq \{\ lam_v(v_1,\ldots,v_n)\ \}\end{array}}{\forall 1\leq i\leq n.\ v_i \supseteq w_i}$$

which is used to resolve indirect function calls in the following manner:

```
int *f(int *r) { return r; } (1)  f• ⊇ r

int *(*p)(int*) = &f;        (2)  p ⊇ { lam_f(r) }
int *q = ... ;               (3)  q ⊇ { ... }
p(q);                        (4)  *p(q)
─────────────────────────────────────────────────────
                             (5)  r ⊇ q          (func, 2+4)
                             (6)  ...
```

Here, we see that constraints are introduced on-the-fly between the actual parameters and their caller values. The main issue here is the implementation of *lam*. Certainly, we don't wish to sacrifice the ability to implement solutions as integer sets. One approach is to place the *lam* constructs into a table, so they are identified by index. Thus, if care is taken to avoid clashes with the variable identifiers, the two element types can co-exist in the same solution set. However, this is inelegant as we must litter our algorithm with special type checks. For example, when dealing with $*p \supseteq q$, we must check for *lam* values in $Sol(p)$. In the next section, we present a simple and elegant solution for dealing with indirect function calls which forms part of our mechanism for field-sensitivity.

4. EXTENDING THE BASIC MODEL

We now present our approach to modelling indirect function calls and, in the following section, we will build upon this to obtain a field-sensitive analysis.

An important assumption of our approach is that each constraint variable is identified by a unique integer from $\{0\ldots n-1\}$, where n is the number of constraint variables. Thus, the solution sets can be represented by integer sets and this has several advantages: firstly, the sets themselves can be stored in an array indexed by variable identifier for fast lookup; secondly, common data structures for implementing the solution sets (e.g. bit vectors, sorted arrays and balanced trees) are most naturally suited to holding integers. For example, bit vector elements are accessed by *index*. Thus, when storing integers (from a finite range), element lookup is fast because the (integer) element can directly index the vector. Storing anything else requires mapping elements to indices — adding an extra level of indirection. Arrays and trees benefit on both element lookup *and* set union since they can use the machine's internal (integer) comparison instruction, rather than a *comparator method* which will likely be more expensive.

$$[deref_4]\ \frac{\begin{array}{c} p \supseteq *(q{+}k) \quad q \supseteq \{r\} \\ idx(s) = idx(r){+}k \\ idx(s) \leq end(r) \end{array}}{p \supseteq s} \qquad [deref_5]\ \frac{\begin{array}{c} *(p{+}k) \supseteq q \quad p \supseteq \{r\} \\ idx(s) = idx(r){+}k \\ idx(s) \leq end(r) \end{array}}{s \supseteq q}$$

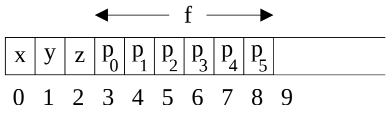
Fig. 3. Extended Inference Rules.

The crucial observation is that using integer identifiers allows us to reference a variable by an offset from another. Thus, we introduce the following forms:

$$p \supseteq *(q{+}k) \mid *(p{+}k) \supseteq q$$

Here k is an arbitrary constant and $*(p+k)$ means "load $Sol(p)$ into a temporary set, add k to each element and dereference as before". To understand this more clearly, consider two variables x and y indexed by 2 and 3 respectively. If $p \mapsto \{x\}$ then $*(p{+}1) \supseteq q$ evaluates to $*(\{x\}{+}1) \supseteq q$, which is really $*(\{2\}{+}1) \supseteq q$, and applying the addition gives $*(\{2{+}1\}) \supseteq q \equiv *(\{3\}) \supseteq q \equiv *(\{y\}) \supseteq q \equiv y \supseteq q$. Of course, when $k{=}0$, these new forms are equivalent to those of the original language. The corresponding inference rules are given in Figure 3, where idx maps variables to their index. For now, ignore the use of $end(\cdot)$ — we return to discuss it later.

Now, for each function $f(p_0, \ldots, p_i)$ in the program whose address has been taken (either via the `&` operator or by direct assignment to a variable), we create $i{+}1$ consecutively indexed constraint variables to represent $p_0 \ldots p_i$:



Here, x, y and z represent some other variables allocated before those of f. The key point is that each parameter of f can be accessed as an offset from p_0 and, thus, we model the address of f by that of p_0. Figure 4 aims to clarify this.

The purpose of using $end(\cdot)$ in the inference rules is to prevent unwanted value flow in the case of a function pointer that has the wrong type for a function it points to. The following illustrates this:

```
void f(int *q) { ... }
int g(int *a,int *b) {

 void (*p)(int *,int*);
 p = (void(*)(int*,int*)) &f;
 *p(a,b);
}
```

$$\begin{array}{l} idx(q) = 0,\ end(q) = 0 \\ idx(a) = 1,\ end(a) = 2 \\ idx(b) = 2,\ end(b) = 2 \\ idx(p) = 3,\ end(p) = 3 \\ p \supseteq \{q\} \\ *(p{+}0) \supseteq a \\ *(p{+}1) \supseteq b \end{array}$$

For each variable, $end(\cdot)$ determines where the enclosing block of consecutively allocated variables ends. Without constraints on $end(\cdot)$, the inference rules of Figure 3 would derive $a \supseteq b$ from $*(p{+}1) \supseteq b$ above as $idx(a) = idx(q){+}1$. While it remains unclear how best to model this, it certainly does not make sense to propagate information into the parameters of `g()`. Therefore, we provide $end(\cdot)$ information to prevent this. Furthermore, while the above example results from

```
void f(int **q, int *r) {
 *q = r;
}

void g(...) {
 void (*p)(int**, int*);
 int *a,*b,c;
 p = &f;
 b = &c;
 p(&a,b);

}
```

(1,2)	$idx(q) = 0,\ idx(r) = 1$
(3)	$*q \supseteq r$
(4,5)	$idx(p) = 2,\ idx(a) = 3$
(6,7)	$idx(b) = 4,\ idx(c) = 5$
(8)	$p \supseteq \{q\}$
(9)	$b \supseteq \{c\}$
(10)	$t \supseteq \{a\}$
(11)	$*(p{+}0) \supseteq t$
(12)	$*(p{+}1) \supseteq b$

(13)	$q \supseteq t$	$(deref_5, 8{+}11{+}1)$
(14)	$q \supseteq \{a\}$	$(trans, 10{+}13)$
(15)	$r \supseteq b$	$(deref_5, 1{+}2{+}8{+}12)$
(16)	$r \supseteq \{c\}$	$(trans, 9{+}15)$
(17)	$a \supseteq r$	$(deref_2, 3{+}14)$
(18)	$a \supseteq \{c\}$	$(trans, 16{+}17)$

Fig. 4. This example illustrates how the analysis deals with parameter passing through function pointers. Constraint 8 is the key as `&f` is translated into q — the first parameter of `f` — allowing us access to r through the offset notation in (11) above. In fact, return values can be modelled using this mechanism if we allocate the corresponding variable (e.g. $f_\bullet$) to the slot following the last parameter. Thus, we can always determine the offset of the return value from the type of the function pointer being dereferenced. Note, allocating the return slot before the parameter slot(s) causes problems when the function is incorrectly typed. This commonly occurs in C, when a function has multiple prototypes of which some incorrectly assign a `void` return type. The problem is that code which uses the function through an incorrect prototype will still compile and run correctly (so long as the return value is not needed). However, our analysis would fail (if return slot was allocated first) for calls to the function in the presence of an invalid prototype. This is because the first argument of that call will be written into the first parameter slot, which is actually the return value! Of course, this can be overcome by simply allocating a return slot for all functions regardless of their return type; however, this will not work for the field-sensitive formulation of Section 4.1, where multiple slots may be required for functions which return aggregates by value.

programming error, similar examples can be constructed which arise solely from the inaccuracies inherent in flow- and context-insensitive analysis. Note, attempting to read a return value where none exists is also prevented by the $end(\cdot)$ information. This is because the return slot(s) are considered part of the block of consecutively allocated variables for a function definition (see Figure 4 for more on return slots).

Finally, the C language supports functions with variable length argument lists (`varags`). These can be supported quite easily (albeit conservatively) by providing a special constraint variable at the end of the parameter block (before the return value) for a function to capture those extra parameters passed to that function.

4.1 Field-Sensitive Pointer Analysis

In this section, we further extend the language of set-constraints to support field-sensitive pointer analysis of C. Our formulation can be regarded as an instance of the general framework for field-sensitive pointer analysis of C by Yong *et al.* [Yong et al. 1999]. In fact, it is equivalent to the most precise analysis of portable (as defined under the ISO/ANSI standard) C programs their system can describe and we consider here some important algorithmic issues which they did not address.

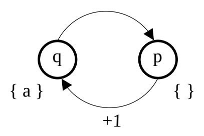
For Java, there are also several existing extensions to the set-constraint language which support field-sensitive analysis [Rountev et al. 2001; Liang et al. 2001; Whaley and Lam 2002; Lhoták and Hendren 2003]. However, Java presents a simpler problem than C in this respect, since it does not permit the address of a field to be taken. Indeed, it turns out the language of the previous section is sufficient for field-sensitive analysis of Java. This works by using blocks of constraint variables, as we did for functions, to represent aggregates. For example, in the following, a block of two constraint variables (one for each field) is used to represent "`aggr a`":

`typedef struct { int *f1; int *f2; } aggr;`	
`aggr a,*b;`	$idx(a.f1)=0, idx(a.f2)=1, idx(b)=2$
`int *p,**q,c;`	$idx(p)=3, idx(q)=4, idx(c)=5$
`b = &a`	$b \supseteq \{a.f1\}$
`b->f2 = &c;`	$*(b+1) \supseteq t,\ t \supseteq \{c\}$
`p = b->f2;`	$p \supseteq *(b+1)$

Here, the address of `a` is modelled by the address of its first field. In this way, the fields of `a` can be accessed via `b` without problem.

To analyse C, however, we must also be able to translate "`q=&(b->f2);`". This is a problem since we want to load the *index* of $a.f2$ into $Sol(q)$, but there is no mechanism for this. Therefore, we extend the language to permit the translation: $q \supseteq b+1$, meaning *load* $Sol(b)$ *into a temporary, add 1 to each element and merge into* $Sol(q)$. Note the inference rule in Figure 5. This form can be represented by turning the constraint graph into a weighted multigraph, where weight determines increment — so $p \supseteq q+k$ gives $q \xrightarrow{k} p$. One difficulty with this new form is the *Positive Weight Cycle (PWC)* problem. For example:

`aggr a,*p; void *q;`	
`q = &a;`	$q \supseteq \{a\}$
`p = q;`	$p \supseteq q$
`q = &(p->f2);`	$q \supseteq p+1$
`/* now use q as int* */`	



This is legal and well-defined C code. Here, the cycle arises from flow-insensitivity, but other forms of imprecision can also cause them. Figure 6 provides a (somewhat contrived) example of a positive-weight cycle arising from imprecision in the heap model. So, it seems that any formulation of a field-sensitive analysis for C will necessarily have to deal with positive weight cycles. This is supported by the work of Chandra and Reps, who encounter the same issue with a similar field-sensitive analysis [Chandra and Reps 1999a; 1999b]. In general, the problem is that cycles

$$[add]\ \frac{p \supseteq q{+}k \quad q \supseteq \{r\} \quad idx(s) = idx(r){+}k \quad idx(s) \le end(r)}{p \supseteq \{s\}}$$

Fig. 5. An inference rule for constraints of the form $p \supseteq q{+}k$

`typedef struct { int *f1; int *f2; } aggr;`	
`void *f(int s) { return malloc(s); }`	(1) $f_\bullet \supseteq \{HEAP0\}$
`aggr **p,*a; int **q,*b;`	
`p = (aggr **) f(sizeof(aggr *));`	(2) $p \supseteq f_\bullet$
`q = (int **) f(sizeof(int *));`	(3) $q \supseteq f_\bullet$
`*p = ... ;`	(4) $*p \supseteq \ldots$
`a = *p;`	(5) $a \supseteq *p$
`b = &a->f2;`	(6) $b \supseteq a + 1$
`*q = b;`	(7) $*q \supseteq b$

(8) $p \supseteq \{HEAP0\}$	$(trans, 1{+}2)$
(9) $q \supseteq \{HEAP0\}$	$(trans, 1{+}3)$
(10) $a \supseteq HEAP0$	$(deref_2, 5{+}8)$
(11) $HEAP0 \supseteq b$	$(deref_1, 7{+}9)$

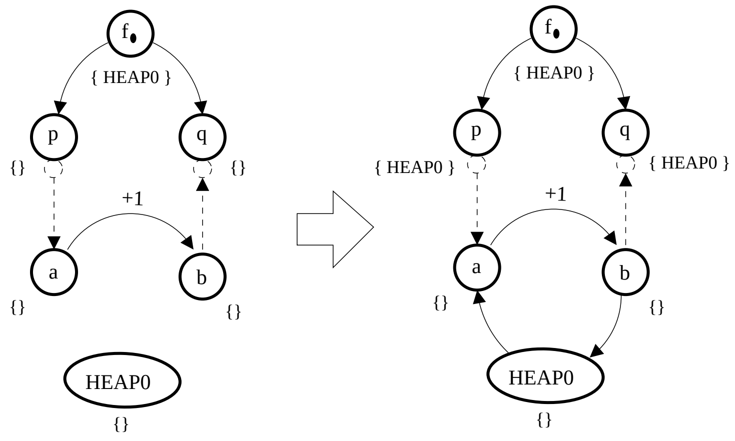
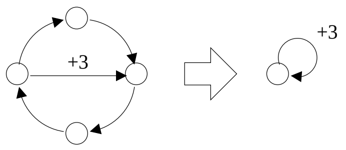


Fig. 6. An example illustrating how a positive-weight cycle can arise from imprecision in the *heap model*. We have included both the constraint derivation and the corresponding graph representation for clarity. The key observation is that a *malloc wrapper* (i.e. `f(int)`) is used to allocate storage. This means that, since a *static heap model* is being used where all objects returned by a particular call to `malloc` are represented by one constraint variable, the local variables `p` and `q` end up containing the same target constraint variable (i.e. *HEAP0*). Therefore, *HEAP0* actually represents two distinct heap objects in the program and we have carefully used this to construct a positive-weight cycle involving *HEAP0*, *a* and *i*. A static heap model is the most common heap model since the alternative, a *dynamic heap model*, can be very costly [Nystrom et al. 2004b].

describe infinite derivations. To overcome this, we use *end()* information, as with function pointers, so that a variable is only incremented within its enclosing block.

Another problem with weighted edges is that *cycle elimination* is now unsafe, since nodes in a cycle need no longer share the same solution. To tackle this, we observe that a cycle can be collapsed when there is a zero weighted path between all nodes and intra-cycle weighted edges are preserved as self loops. The following example demonstrates this, where unlabelled edges are assumed to have zero weight:



An interesting question is whether or not code such as the following will produce positive-weight cycles:

```
typedef struct link { int *data; struct link *next; } link
int *p = ...; link q = ...;
while(p != null) { p = p + 1; q = q->next; }
```

In fact, neither statement in the above loop results in a positive-weight cycle. The statement involving `q` can be rewritten as "`q = *(q+k)`", where `k` is the offset of `next`, and corresponds to a constraint from Figure 3 (hence, it is not a positive-weight cycle). The statement "`p = p + 1`" is harder to understand. One might expect this to be translated into the constraint $p \supseteq p+1$ (which does represent a positive-weight cycle). In fact, it is translated into $p \supseteq p$ because only *field-accesses* (i.e. expressions such as `x.y` and `x->y`) are translated into constraints of the form $p \supseteq q+k$. This may seem problematic, if one considers that pointer arithmetic could be used to simulate a field access. However, using pointer arithmetic to access anything other than an array has undefined behaviour under the ISO/ANSI standard (as does casting a `struct` to an array) [ISO90 1990, 6.3.6]. Furthermore, as outlined in Section 5, we model array objects with a single constraint variable and, hence, $p \supseteq p$ is a safe translation for "`p = p + 1`".

A further source of complication for our system is the difficulty in determining how many fields a heap variable should have. This is especially true if a static heap model is used, as highlighted in the following:

```
typedef struct {double d1; int *f2;} aggr1;
typedef struct {int   *f1; int *f3;} aggr2;
```

`void *f(int s) { return malloc(s); }`	$f_\bullet \supseteq \{\mathit{HEAP0}\}$
`void *g(int s) { return malloc(s); }`	$g_\bullet \supseteq \{\mathit{HEAP1}\}$
`aggr1 *p = f(sizeof(aggr1));`	$p \supseteq f_\bullet$
`aggr2 *q = f(sizeof(aggr2));`	$q \supseteq f_\bullet$
`int *x = f(100);`	$x \supseteq f_\bullet$
`int *y = g(100);`	$y \supseteq g_\bullet$

Here, only one constraint variable is created to model every object allocated by `f(int)` and this legitimately ends up representing multiple objects of different type (i.e. *HEAP0* represents an `aggr1` object *and* an array of `int`s). Indeed, it is often impossible to determine the types that a heap variable will represent to tell whether a heap variable will even be used to represent an aggregate.

This presents a problem as we need to determine which objects represent aggregates in order to allocate the necessary blocks of constraint variables. Thus, we either model heap variables field-insensitively (not ideal) or assume they always represent aggregates. Our choice is the latter, which raises a further problem: *which aggregate should a heap variable represent?* A simple solution is to assume it will be the largest `struct` in the program, since this ensures enough constraint variables are allocated for every eventuality. Effectively then, each heap variable is modelling the C `union` of all `struct`s. So, in the above, *HEAP0* and *HEAP1* both model `aggr1` and `aggr2` and are implemented with two constraint variables: the first representing fields `f1` and `d1`; the second `f2` and `f3`. The observant reader will have noticed something strange here: *the first constraint variable models fields of different sizes.* This seems a problem as, for example, writing to `d1` would invalidate `f1` and `f3` on many platforms/architectures. In practice, however, this cannot be exploited without using undefined C code, since it relies on implementation dependent information regarding type size:

```
typedef struct {double d1; int *f2;} aggr1;
typedef struct {int   *f1; int *f3;} aggr2;

aggr1 *p = malloc(sizeof(aggr1));        idx(HEAP0.F0) = 0
                                          idx(HEAP0.F1) = 1
int a,*r;                                 p ⊇ {HEAP0.F0}
aggr2 *q = (aggr2 *) p;                   q ⊇ p
q->f3 = &a;                               *(q+1) ⊇ t1, t1 ⊇ {a}
p->d1 = 1.0; /* clobbers q->f3 */         *(p+0) ⊇ t2, t2 ⊇ {?}
r = q->f3;                                r ⊇ *(q + 1)
```

Here, our analysis concludes $r \mapsto \{a\}$, which is unsound on platforms where `sizeof(double)` is larger that `sizeof(int)` because the assignment to `p->d1` overwrites part of `q->f3`. Note the special value "?", used to indicate that a pointer may target anything. In general, we are not concerned with this issue as our objective is to model portable C programs only. Having said that, our system can support non-portable programs by using actual (i.e. physical) offsets instead of field-numbers. This requires some extension to our inference system to support situations (like the above) where an assignment affects multiple fields at once. Indeed, Nystrom *et al.* claim to have done just this, although they do not discuss exact details [Nystrom et al. 2004b]. Finally, nested `struct`s are easily dealt with by "inlining" them into their enclosing `struct`, so that each nested field is modelled by a distinct constraint variable.

4.2 Constraint Solving

We now present our algorithm, referred to as PW, for solving constraint sets from our extended language. Pseudo-code for the algorithm is given in Figure 7 and, essentially, it follows the procedure outlined in Section 3. That is, it repeatedly propagates the solution of each node into its successors (referred to as *visiting* a

```
foreach $x \in V$ do $\Delta(x) = Sol(x)$;

while $\exists x.\Delta(x) \neq \emptyset$ do
  // invoke cycle detection algorithm here to collapse all zero-weight cycles
  foreach $n \in V$ in topological order do
    // VISIT $n$
    if $\Delta(n) \neq \emptyset$ then
      $\delta = \Delta(n)$;
      $\Delta(n) = \emptyset$;

      // STAGE 1: process all constraints involving $*(n + k)$
      foreach $c \in C(n)$ do case $c$ of
        $*(n + k) \supseteq w$:
          foreach $i \in \delta$ do
            $f = i + k$; // field variable being accessed
            if $f \leq end(i) \wedge w \xrightarrow{0} f \notin E$ do
              $E = E \cup \{w \xrightarrow{0} f\}$;
              $t = Sol(w) - Sol(f)$;
              if $t \neq \emptyset$ then
                $\Delta(f) = \Delta(f) \cup t$;
                $Sol(f) = Sol(f) \cup t$;
        $w \supseteq *(n + k)$:
          foreach $i \in \delta$ do
            $f = i + k$; // field variable being accessed
            if $f \leq end(i) \wedge f \xrightarrow{0} w \notin E$ do
              $E = E \cup \{f \xrightarrow{0} w\}$;
              $t = Sol(f) - Sol(w)$;
              if $t \neq \emptyset$ then
                $\Delta(w) = \Delta(w) \cup t$;
                $Sol(w) = Sol(w) \cup t$;

      // STAGE 2: propagate $\delta$ to successors
      foreach $n \xrightarrow{k} w \in E$ do
        foreach $i \in \delta$ do
          $v = i + k$;
          if $v \leq end(i) \wedge v \notin Sol(w)$ then
            $\Delta(w) = \Delta(w) \cup \{v\}$;
            $Sol(w) = Sol(w) \cup \{v\}$;
```

Fig. 7. Algorithm PW, a worklist-style algorithm supporting field-sensitive pointer analysis with function pointers. The algorithm assumes that $Sol(p)$ has been initialised with all trivial constraints of the form $p \supseteq \{q\}$. The set $C(n)$ contains all complex constraints involving "$*(n + k)$". Notice that the code for collapsing cycles has been omitted for brevity. This simply collapses all cycles which have a zero-weight path between their nodes (see Section 4.1 for more on this); it has no other effect on the constraint graph. The algorithm uses a topological iteration strategy to improve performance. Computing the topological sort necessary for this can often be done for free by the cycle detector. Finally, the set $\Delta(n)$ holds the *change* in $Sol(n)$ since the last time n was visited. This ensures a node z is a member of δ for at most one visit of each node.

node) until no change is observed. Normally, this is done using a *worklist* algorithm which maintains a list (the *worklist*) of those nodes remaining to be visited. Every node is initially on the worklist and the algorithm begins by removing a node n and visiting it. In doing this, any (immediate) successor of n whose solution has changed is placed back onto the worklist. This continues until the worklist is empty.

A well-known complication is that the order in which nodes are visited (i.e. taken off the worklist) can greatly affect performance [Horwitz et al. 1987; Burke 1990; Bourdoncle 1993b; Chen and Harrison 1994; Schön 1995; Fecht and Seidl 1996; Nielson et al. 1999; Pearce et al. 2004b; Pearce 2005]. A good choice is to visit nodes in *topological order* (also known as *reverse postorder*). For some types of program analysis this approach is considered optimal (see [Horwitz et al. 1987; Bourdoncle 1993b]). Unfortunately, this is not the case for the pointer analysis problem being studied here [Pearce 2005]. Nevertheless, it remains an excellent choice, especially considering that no better alternative is known. Therefore, this strategy is used in PW although, to implement this efficiently, a *worklist* in the true sense of the word is not actually used. Instead, all nodes are sorted topologically at the beginning of each round and then visited in sequence. For each node n, the algorithm first determines whether its solution has changed since it was last visited. This is done by checking whether $\Delta(n)$, which records the *change* in n's solution between visits, is empty. If not, the algorithm processes all complex constraints involving $*(n + k)$ and propagates $\Delta(n)$ to all successors. Note, $\Delta(n)$ is loaded into δ to ensure that $\Delta(n)$ can be cleared before propagation. This is necessary to properly deal with (positive weight) self loops.

The use of $\Delta(\cdot)$ is crucial to obtaining an optimal complexity bound on the runtime of PW, because it ensures a node z is a member of δ for at most one visit of each node. This technique has been previously referred to in the literature as *difference propagation* [Fecht and Seidl 1998; Pearce et al. 2004b] and *incremental sets* [Lhoták and Hendren 2003; Berndl et al. 2003]. Another important point about PW is that we have omitted details of which cycle detection algorithm to use. The standard approach is to employ Tarjan's well-known algorithm for detecting *strongly connected components* (see [Tarjan 1972; Nuutila and Soisalon-Soininen 1994; Gabow 2000]). This algorithm always visits every node and traverses every edge when called. However, as with difference propagation, it is only necessary to examine those parts of the graph which have changed since the last round. Therefore, several *dynamic* cycle detection algorithms have been proposed which aim to perform an amount of work proportional to the size of the graph change (see e.g. [Shmueli 1983; Fähndrich et al. 1998]). In fact, we have elsewhere developed the fastest known solutions for this problem [Pearce et al. 2004b; Pearce 2005]. These came out of our work on *dynamic topological sort* (see [Pearce and Kelly 2004; 2006]) and, it turns out, the two problems are closely related. In spite of the advantages of these more advanced algorithms we choose, in the experimental comparison which follows, to use Tarjan's algorithm for detecting cycles. The reason for this is to simplify our experimental results and to aid comparisons with previous works (which did not have such cycle detection algorithms available to them). In fact, we find in practice that Tarjan's algorithm is surprisingly competitive (compared with these alternatives) and experimental data looking at this can be found in [Pearce 2005].

Lemma 4.1. *Let $D = (V, E, K, C)$ be a directed weighted constraint multigraph, where K is the set of possible (integer) edge weights and $C(n)$ contains all complex constraints involving $*(n+k)$ for any node $n \in V$, where $k \in K$. Algorithm PW needs at most $O(v^4)$ time to solve D, where $v = |V|$.*

Proof. Let E_* be the set of edges in the solved multigraph. As edges are only added by the algorithm, it follows that $E \subseteq E_*$. Furthermore, let $E_*^+(n)$ be the set of outedges from n in E_*. Now, there are several key points to note:

(i) For any $z, n \in V$, $z \in \Delta(n)$ for at most one visit of n. This holds because: firstly, after initialisation, z is only added to $\Delta(n)$ if $z \notin Sol(n)$; secondly, at no point is z added to $\Delta(n)$ without it also being added to $Sol(n)$; finally, $\Delta(n)$ is cleared when n is visited and elements are never removed from $Sol(n)$.

(ii) For any $k \in K$, it holds that $k < |V|$. This is because all elements propagated across an edge $x \xrightarrow{k} y$, where $k \geq |V|$, will always fail the *end*() test. Therefore, constraints with $k \geq |V|$ can be culled from the constraint set before solving, since they have no effect. This means $|E|$ (hence $|E_*|$) is $O(v^3)$ in the worst case, rather than $O(v^2)$ (which is normal for non-multigraphs).

(iii) $C(n)$ has at most $O(v^2)$ elements. For each constraint of the form $w \supseteq *(n+k)$, we know $k \in K$ and $w \in |V|$. Therefore, there are $|K| \times |V| \leq |V|^2$ possible instances of it in $C(n)$. Similar arguments hold for the other two complex constraint forms, meaning $|C(n)| \leq 3 \cdot |V|^2$.

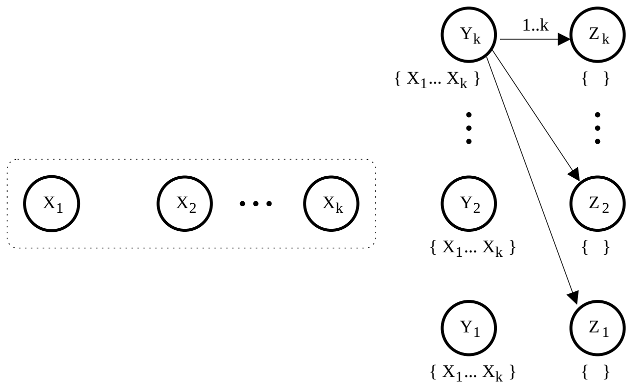
We will now show that at most $O(v^4)$ time is spent executing each stage of the main loop. Starting with stage 2 (as marked on Figure 7), it holds that the inner loop body executes at most $v \cdot |E_*^+(n)|$ times for each node n. This follows from (i) above which implies that $\sum_{i=0} |\delta_i| \leq |V|$, where δ_i represents δ on the i^{th} visit of n. Thus, the total time spent executing stage 2 is $O(v^2 \cdot |E_*^+(n)|) \equiv O(v^4)$.

For stage 1, it holds that the loop body of each case statement is executed at most $v \cdot |C(n)|$ times per node. As before, this follows from (i) above. Now, for each case the innermost if-body needs at most $O(v)$ time to merge t into $Sol(f)$ etc. Furthermore, the if-body only executes when a new edge is added — meaning at most $O(v \cdot |E_*^+(n)|)$ time per node is spent executing the if-body itself. Therefore, the total time spent executing each case in stage 1 is $O(v \cdot (|E_*^+(n)| + |C(n)|))$ per node which, following from (ii) and (iii) above, is $O(v^4)$ in the worst case.

Finally, it remains to show that the amount of time spent executing statements outside the if-body (since we have already bounded those inside) is also $O(v^4)$. This is easy enough, since the outer for-loop will only execute if there is some change in the state of the multigraph. Furthermore, as $\Delta(n)$ can change at most v times (point (i) above), there are at most $O(v^2)$ possible changes. In the worst case, each change requires a complete iteration of the outer for-loop and, hence, the for-loop body executes at most $O(v^3)$ times. □

This proof has several implications for the implementation of PW. Specifically, inserting an element into $Sol(n)$ must take constant time and, thus, we implement $Sol(n)$ with a bit vector. Likewise, iterating $\Delta(n)$ must take $O(|\Delta(n)|)$ time and, thus, we implement $\Delta(n)$ with an array (note, this can be unsorted since the algorithm guarantees no element is added to $\Delta(n)$ more than once). Also, δ follows the implementation of $\Delta(n)$.

An interesting question is what a worst-case input for algorithm PW looks like and the following provides a simple example:



Here, $k = \frac{1}{3}v$, the nodes $X_1 \ldots X_k$ represent an aggregate variable and, although not shown completely, every edge given by $\{Y_i \xrightarrow{l} Z_j \mid 1 < i, j, l < k\}$ is present. Thus, there are $O(v^3)$ edges and the algorithm attempts to propagate $O(v)$ elements across them (although many will fail the *end*() test and/or already be present in target solution set). Therefore, algorithm PW needs $O(v^4)$ time to solve this graph.

A subtle aspect of our complexity analysis is the choice to measure complexity in terms of the number of *constraint variables* (i.e. v), rather than the number of *constraints*. If instead we measure complexity in terms of t, the number of initial constraints, then PW needs only $O(t^3)$ time in the worst case. While this may seem surprising, it holds because constraints of the form $q \supseteq x+k$ cannot be introduced during solving.

LEMMA 4.2. *A set of t constraints can be solved in $O(t^3)$ time under the inference system of Figures 1, 3 and 5.*

PROOF. Let t_t, t_s and t_c be (respectively) the initial number of trivial, simple and complex constraints in the constraint set. Then, $t = t_t + t_s + t_c$. Let v be the number of constraint variables and k the number of possible constraint weights. Now, we know $k \leq v \leq 2 \cdot t$ and that at most kv^2 simple constraints are possible. However, the $deref$ rules can only generate v simple constraints from each complex constraint (since only weightless constraints can be generated). Thus, up to vt_c simple constraints can be generated in total whilst solving the constraint set. Furthermore, $O(v)$ trivial constraints can be propagated across each simple constraint and, hence, the $trans$ rule can be applied $O(v \cdot (t_s + vt_c)) \Rightarrow O(t^3)$ times. □

The $O(t^3)$ result tells us that the underlying hardness of constraint solving is the same as for a standard field-insensitive analysis (recall from the end of Section 2, that a standard insensitive analysis needs $O(t^3)$ as well), since t measures input size for the *solving algorithm*. However, the $O(v^4)$ result reflects the fact that our field-sensitive analysis has a much larger *input domain* than a standard field-insensitive analysis (due to edge-weights). This means that, *for a given program*, many more constraints may be generated for our field-sensitive analysis than for the field-insensitive analysis. This is important, since it tells us that field-sensitive analysis for C may take considerably longer to solve *for a given program* (and this is what we really care about).

5. EXPERIMENTAL STUDY

In this section, we provide empirical data over a range of benchmarks comparing our field-insensitive and -sensitive systems. Table I provides information on our benchmark suite. With two exceptions, all are available under open source licenses and can be obtained online (e.g. `http://www.gnu.org`). Note that `cc1` is the C compiler component of `gcc`, while `named` is distributed in the BIND package. While both `147.vortex` and `126.gcc` are not available under open source licences, they form part of the SPEC95 benchmark suite and have been included to aid comparison with previous work.

The SUIF 2.0 research compiler from Stanford [SUIF2] was deployed as the frontend for generating constraint sets. In all cases, we were able to compile the benchmarks with only superficial modifications, such as adding extra "`#include`" directives for missing standard library headers or updating function prototypes with the correct return type. The constraint generator operates on the full C language and a few points must be made about this:

—*Heap model.* The static model discussed briefly in Section 4.1 was used. Recall that this uses a single constraint variable to represent all heap objects created from a particular call to `malloc` and other related heap allocation functions.

—*Arrays.* These are treated by ignoring the index expression and, hence, representing all elements of an array with one constraint variable.

—*String Constants.* A single constraint variable was used to represent all string constants. In other words, we consider the right hand side of `p="foo"` and `q="bar"` as referring to the same object.

—*External Library Functions.* These, almost entirely, came from the GNU C library and were modelled using hand crafted summary functions, capturing only aspects relevant to pointer analysis.

—*Variable Length Argument Lists.* These were dealt with using the technique outlined in Section 4, where a single constraint variable is used to represent every possible parameter in the `vararg` list of a method.

Our experimental framework contained a direct implementation of algorithm PW and its *field-insensitive* variant, henceforth referred to as PWFI. The latter was almost identical to PW, except those parts relating to field-sensitivity were removed. Furthermore, our constraint generator produced field-sensitive constraints (see Figures 3 + 5) for PW, but field-insensitive constraints (see Figure 1) for PWFI. Cycle detection and subsumed node compaction (recall Section 3) were always applied to the initial constraint set, although projection merging (see [Su et al. 2000]) was not

			Constraints		
	Ver	LOC	Triv	Simp	Comp
uucp	1.06.1	15,501 / 10,255	798	3,047	1,512
make	3.79.1	22,366 / 15,401	1,417	4,592	2,428
gawk	3.1.0	27,526 / 19,640	2,354	8,180	4,589
147.vortex	SPEC95	52,624 / 40,247	9,774	12,113	8,365
bash	2.05	70,913 / 50,947	3,400	12,693	5,295
sendmail	8.11.4	68,106 / 49,053	5,185	10,925	5,027
emacs	20.7	128,859 / 93,151	10,629	12,554	16,866
126.gcc	SPEC95	193,752 / 132,435	7,410	38,716	25,232
cc1 (gcc)	2.95.1	271,053 / 188,535	15,096	59,176	36,684
named	9.2.0	109,001 / 75,599	17,645	30,613	35,428
ghostscript (gs)	6.51	215,605 / 159,853	20,449	52,307	68,029

Table I. Structural information on our benchmark suite. LOC measures lines of code, not including those in header files. The first figure reports the total, while the second only non-comment, non-blank lines. The constraint counts are from the initial (i.e. unsolved) constraint set and show Trivial ($p \supseteq \{q\}$), Simple ($p \supseteq q$) and Complex (involving '*'). Note, these counts are for the field-insensitive constraint sets only. Values for the field-sensitive constraint sets have been omitted for brevity, since we find there is very little difference between the two.

as we found this degraded performance[1]. Note, our reported measurements do not include the time needed for generating the initial constraint sets and performing these simple optimisations. To implement $Sol(n)$ and $\Delta(n)$, both implementations used bit vectors and arrays respectively. They also employed the hash-based duplicate set compaction scheme of Heintze and Tardieu to ensure identical solution sets are shared [Heintze and Tardieu 2001b]. This turns out to be necessary for solving the largest benchmark (ghostscript), which without compaction needs well over 1GB of memory to complete. To validate our solvers we manually inspected the output produced on a test suite of small programs and also by ensuring that each algorithm produced the same output. The full source code for our solving algorithms and constraint generator is available online at `http://www.mcs.vuw.ac.nz/~djp`. Finally, our experimental machine was a 900Mhz Athlon with 1GB of main memory, running Mandrake 10.2. The executables were compiled using gcc 3.4.3, with compiler switches "`-O3`" and "`-fomit-frame-pointer`". Timing was performed using the `gettimeofday` function (which offers microsecond resolution on x86 Linux platforms) and averaged over ten runs — this was sufficient to generate data with a variation coefficient of $\leq$ 0.05, indicating low variance between runs. To reduce interference, experiments were performed with all non-essential system daemons/services (e.g. X windows, `crond`) disabled and no other user-level programs running. The code itself was in C++, making extensive use of the *Standard Template Library* and *Boost Library (version 1.30.2).*

Table II provides a comparison of some interesting differences between the constraint sets generated for the field-sensitive analysis and those used for the insensitive analysis. In particular, we see that the sensitive analysis always uses more constraint variables, which is expected as each field is now modelled with a separate

[1]Projection merging was originally designed for use with inductive form, which we do not use. Thus, we conclude that the technique is simply not suited for use with standard form.

	Constraint Variables			
	Total	Addr	Heap	# PWC
uucp	3,319 / 5,208	199 / 1,929	20 / 940	1
make	4,786 / 6,933	259 / 2,396	69 / 1,794	0
gawk	7,303 / 10,140	333 / 3,137	96 / 2,496	0
147.vortex	11,929 / 16,019	2,201 / 5,943	21 / 2,310	0
bash	10,852 / 13,130	699 / 2,881	36 / 936	0
sendmail	10,276 / 12,428	682 / 2,726	13 / 546	1
emacs	17,972 / 38,181	3,844 / 23,618	172 / 12,900	0
126.gcc	27,895 / 50,654	1,113 / 23,774	231 / 22,407	0
cc1	43,888 / 76,383	1,492 / 33,856	258 / 31,992	1
named	34,778 / 47,230	4,278 / 15,764	24 / 1,704	1
gs	65,338 / 105,114	8,754 / 34,683	18 / 2,412	2

Table II. Data comparing the field-insensitive and -sensitive constraint sets. The breakdown of constraint variables shows the total count, the number of address-taken and the number modelling the heap. In all cases, the two values given apply to the insensitive and sensitive constraint sets (in that order). #PWC reports the number of Positive Weight Cycles in the final (i.e. solved) constraint graph.

variable. Another interesting point is that `named` and `ghostscript` have noticeably fewer constraint variables modelling the heap than others of similar size. In fact, ghostscript uses a malloc wrapper (a custom memory allocator which wraps `malloc`) called `png_malloc`. This explains the small heap variable count, since it implies there will be fewer direct invocations of `malloc`. The `named` benchmark does something similar, in that it uses a memory pool (based on methods such as `isc_buffer_allocate`) to allocate memory in many cases. Finally, the "# PWC" metric shows the number of positive weight cycles in the final graph. It is important to realise that this count may be higher during solving, because some cycles could end up being combined in the final graph. Note that, if at least one positive weight cycle is created then "# PWC" will report a count greater than zero. This is because cycle detection cannot eliminate positive weight cycles — it can only reduce them to self loops.

Figure 8 looks at the effect of field-sensitivity on solving time. It shows clearly that the field-sensitive analysis is more expensive to compute. Furthermore, with the exception of `emacs`, those benchmarks which have positive weight cycles are significantly more expensive, relatively speaking, than the others.

Comments: Figure 9 gives some indication why the field-sensitive analysis is more costly as it shows a reasonably clear correlation between performance and visit count. Recall that, for the field-sensitive analysis, there are generally more constraint variables (recall Table II) and these correspond to nodes in the graph which must be visited. However, Figure 10 indicates that, in many cases, the cost of performing a set union is actually lower. This suggests that the increased precision offered by field-sensitivity could actually lead to improved performance. Indeed, this idea is not new and others have made such observations before (see e.g. [Lhoták and Hendren 2003; Rountev et al. 2001; Whaley and Lam 2002; Heintze and Tardieu 2001b]). An interesting point here is that (with the exception of `gs`) average set size is always *lower* for benchmarks which don't have positive weight

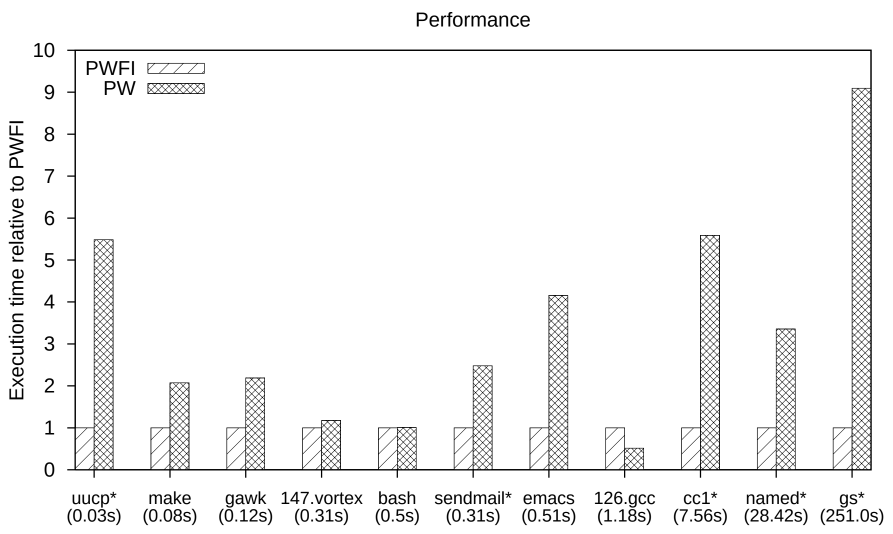


Fig. 8. A chart of our experimental data investigating the effect of field-sensitivity on the performance of algorithm PW. This is given relative to the field-insensitive implementation (PWFI) to allow data for different benchmarks to be shown on the same chart. Below each benchmark, the absolute time taken by PWFI is shown for reference (Table III provides absolute values for both implementations). Benchmarks containing positive weight cycles are marked with an asterisk.

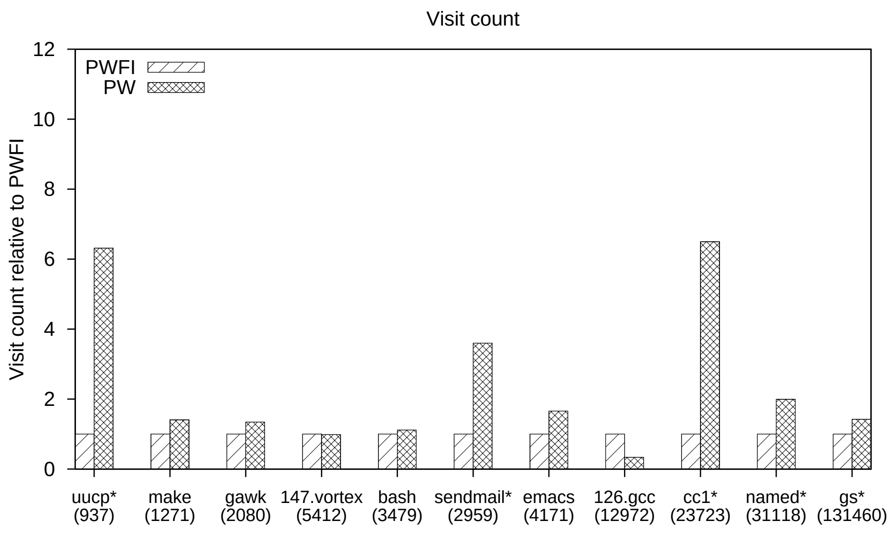


Fig. 9. A chart of our experimental data investigating the effect of field-sensitivity on visit count for algorithm PW. It shows the number of nodes visited by the field-insensitive (PWFI) and -sensitive (PW) algorithms. This is given relative to PWFI implementation and, below each benchmark, the exact number of nodes visited by PWFI is provided (Table III provides absolute values for both implementations). All experimental parameters are the same as for Figure 8 and benchmarks containing positive weight cycles are marked with an asterisk.

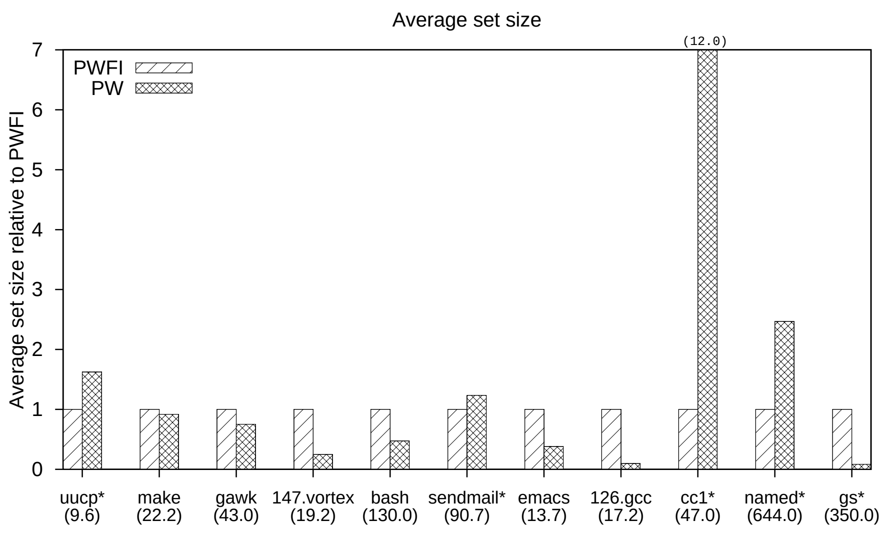


Fig. 10. A chart of our experimental data investigating the effect of field-sensitivity on average set size for algorithm PW. It shows the average set size across all set union operations for the field-insensitive (PWFI) and -sensitive (PW) implementations. This is given relative to the PWFI implementation and, below each benchmark, exact values for PWFI are provided (Table III provides absolute values for both implementations). All experimental parameters are the same as for Figure 8 and benchmarks containing positive weight cycles are marked with an asterisk.

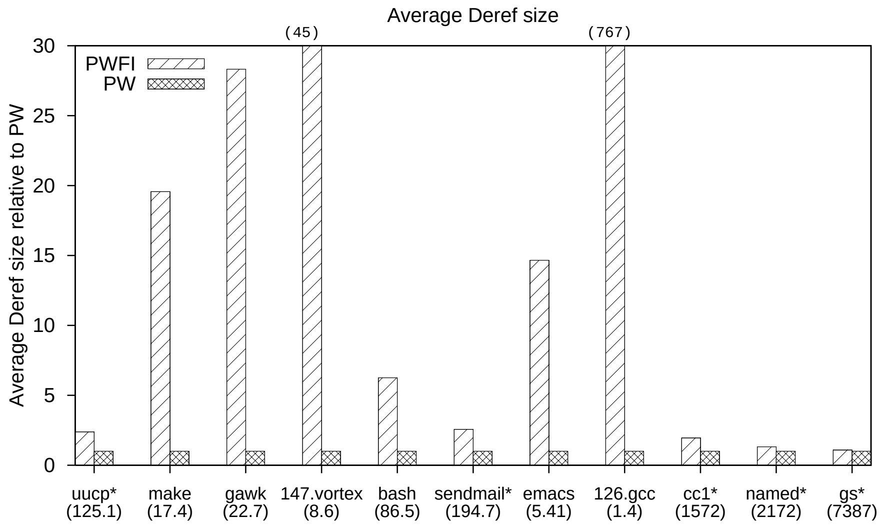


Fig. 11. A chart of our experimental data investigating the effect of field-sensitivity on the Average Deref metric. This is shown for the field-insensitive (PWFI) and -sensitive (PW) implementations and is given relative to PW. Below each benchmark, the exact figures for PW are given for reference (Table III provides absolute values for both implementations). All experimental parameters remain the same as for Figure 8 and benchmarks containing positive weight cycles are marked with an asterisk.

Benchmark		Time (s)	Visit Count	Avg Set Size	Avg Deref Size
uucp	PWFI	0.031	937.0	9.6	298.1
	PW	0.17	5,916.0	15.6	125.1
make	PWFI	0.075	1,271.0	22.2	341.2
	PW	0.156	1,785.0	20.4	17.44
gawk	PWFI	0.12	2,080.0	43.0	643.4
	PW	0.26	2,796.0	32.3	22.72
147.vortex	PWFI	0.31	5,412.0	19.2	384.1
	PW	0.36	5,331.0	4.78	8.6
bash	PWFI	0.5	3,479.0	130.0	544.0
	PW	0.5	3,878.0	61.5	86.5
sendmail	PWFI	0.31	2,959.0	90.7	498.8
	PW	0.77	10,650.0	112.0	194.7
emacs	PWFI	0.51	4,171.0	13.7	79.34
	PW	2.1	6,904.0	5.18	5.41
126.gcc	PWFI	1.18	12,972.0	17.2	1,044.0
	PW	0.61	4,352.0	1.68	1.36
cc1	PWFI	7.56	23,723.0	47.0	3,069.0
	PW	42.2	154,051.0	555.0	1,572.0
named	PWFI	28.42	31,118.0	644.0	2,869.0
	PW	95.4	61,973.0	1,590.0	2,172.0
gs	PWFI	251.0	131,460.0	350.0	8,026.0
	PW	2,130.0	186,757.0	29.8	7,387.0

Table III. Actual data values for the four metrics shown in Figures 8, 9, 10 and 11.

cycles, whilst it is always *higher* on those that do. This suggests that positive weight cycles are a major expense and that eliminating them would be beneficial.

Figure 11 looks at the effect on precision of field-sensitivity but, before any discussion, we must first understand exactly what is being shown. The chart reports the number of possible targets for a dereference site, averaged across all dereference sites. This is called the "Average Deref" metric. It gives a more useful measure of precision, compared with the average set size of all pointer variables, since only dereference sites are of interest to client analyses. To facilitate a meaningful comparison (in terms of precision) between the sensitive and insensitive analyses we must normalise the value. To understand why, consider a pointer p targeting the first field of variable "`struct {int f1; int f2;} a`". For the insensitive analysis, we have the solution $p \supseteq \{a\}$, whilst the sensitive analysis gives $p \supseteq \{a.f1\}$. Thus, the two appear to offer the same level of precision, since their solution sets are of equal size. However, this is misleading because the insensitive analysis actually concludes that p may point to *any field* of a. Therefore, we normalise the insensitive solution by counting each aggregate by the number of fields it contains. In other words, we count $p \supseteq \{a\}$ as though it was $p \supseteq \{a.f1, a.f2\}$.

The main observation from Figure 11 is that field-sensitivity gives more precise results across the board. However, we again find there are significant differences between those benchmarks which have positive weight cycles and those which do not. In particular, those without always show a significantly greater increase in precision from field-sensitivity. Figures 12 and 13 break up the Average Deref metric to show its distribution for each benchmark. They concur with our previous findings that field-sensitivity increases precision, as a general shift is seen from right-to-left, indicating that more dereference sites have fewer targets. We also observe that a large proportion of dereference sites for the three largest benchmarks have a thousand elements or more. And yet, the two similar sized benchmarks `emacs` and `126.gcc` (which don't contain positive weight cycles) have much better distributions. From this, we conclude that positive weight cycles are also a major factor affecting the precision of field-sensitive pointer analysis.

Comments: An interesting observation regarding the Average Deref metric is that it represents an upper bound on the imprecision of the field-insensitive analysis. This is because the normalisation procedure counts each heap object in a points-to set as having the most fields of any `struct` in the program (see Section 4.1 for more on this). This represents a worst-case increase for the field-insensitive analysis, since a particular heap object may only ever be accessed as some smaller `struct` in practice. Of course, the Average Deref metric for the field-insensitive analysis could never be lower than that of the field-sensitive analysis. Nevertheless, the difference between them could potentially be reduced by determining a conservative set of types for each heap object. Figure 10 indicates this reduction is unlikely to be significant in most cases, however, since it shows that even without normalisation the field-insensitive set-sizes are often much greater than their field-sensitive counterparts.

Finally, zero-sized sets for Average Deref arise from an artifact of our linker, which attempts to mimic the GNU linker as closely as possible. The issue is that, when a given object file is linked with the program, all functions contained therein are included — even if not used. Therefore, most of the unreachable code arises from our GNU C library model, where many functions are spread over a small number of files.

6. RELATED WORK

Flow- and context-insensitive pointer analysis has been studied extensively in the literature (see e.g. [Pearce et al. 2004b; Lhoták and Hendren 2003; Fähndrich et al. 1998; Heintze and Tardieu 2001b; Rountev and Chandra 2000; Andersen 1994; Steensgaard 1996; Das 2000]). These works can, for the most part, be placed into two camps: extensions of either Andersen's [Andersen 1994] or Steensgaard's [Steensgaard 1996] algorithm. The former use *inclusion constraints* (i.e. set-constraints) and are more precise but slower, while the latter adopt *unification systems* and sacrifice precision in favour of speed. Thus, new developments tend to be focused either on speeding up Andersen's algorithm (e.g. [Heintze and Tardieu 2001b; Fähndrich et al. 1998; Rountev and Chandra 2000; Pearce et al. 2004b]) or on improving the precision of Steensgaard's (e.g. [Das 2000; Das et al. 2001; Liang and Harrold 1999]). Furthermore, there have been numerous studies on the relative precision of these two approaches (see e.g. [Liang et al. 2001; Foster et al. 2000; Hind and Pioli 2000; Shapiro and Horwitz 1997; Das 2000; Das et al. 2001], with the results confirming that set-constraints offer useful improvements in pre-

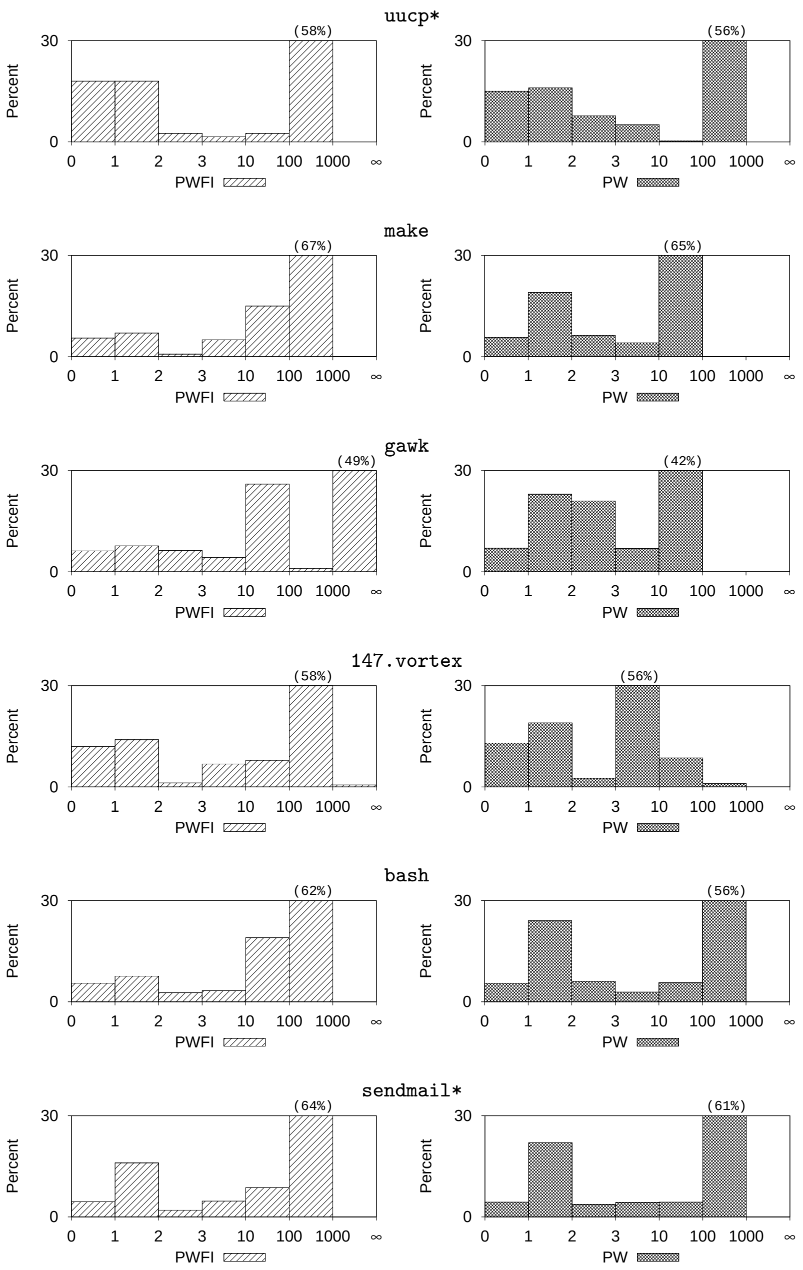


Fig. 12. Charts of our experimental data showing a breakdown of the average points-to set size at dereference sites for each benchmark. Each bar indicates how many dereference sites (as a percentage of the total) have points-to sets of size X, where X lies between the left boundary and up to, but not including, the right boundary. For example, the second bar in each chart gives the number of dereference sites with points-to sets containing exactly one element. Benchmarks containing positive weight cycles are marked with an asterisk. Note, zero sized sets arise from an artifact of our linker (see the discussion for more on this). The exact percentages for both implementations are provided in Table IV.

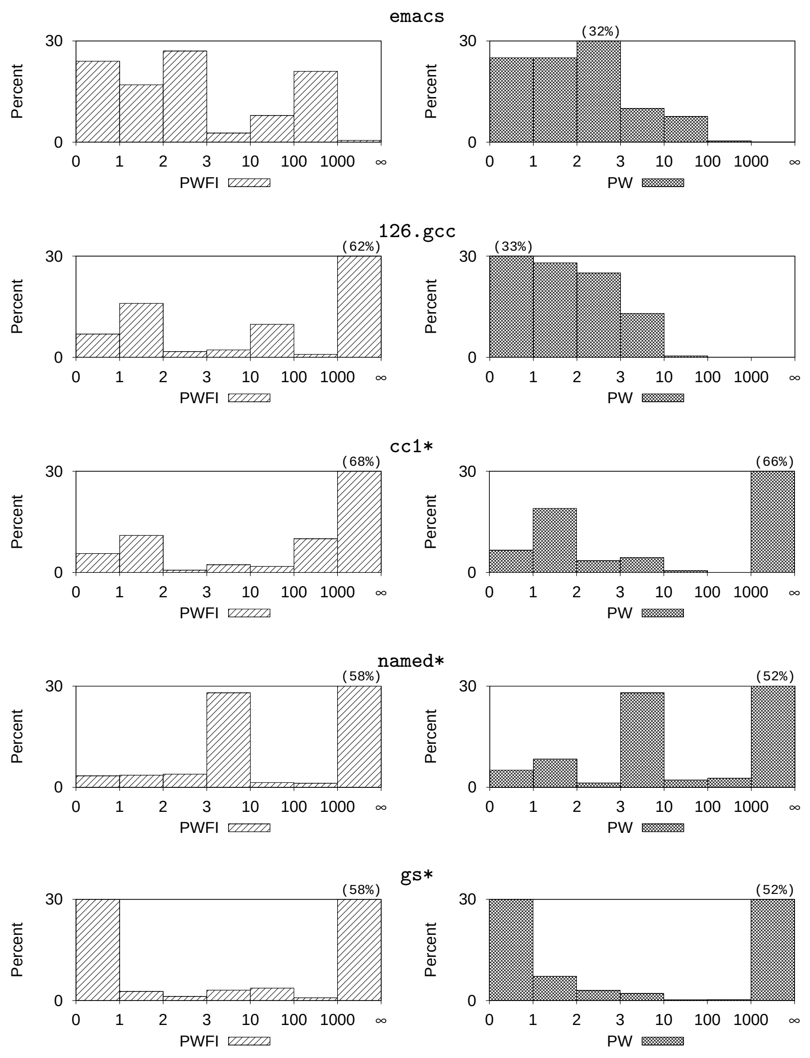


Fig. 13. More charts of our experimental data showing a breakdown of the average points-to set size at dereference sites for each benchmark. Each bar indicates how many dereference sites (as a percentage of the total) have points-to sets of size X, where X lies between the left boundary and up to, but not including, the right boundary. For example, the second bar in each chart gives the number of dereference sites with points-to sets containing exactly one element. Benchmarks containing positive weight cycles are marked with an asterisk. Note, zero sized sets arise from an artifact of our linker (see the discussion for more on this). The exact percentages for both implementations are provided in Table IV.

cision. Much work has also been done on *context-sensitive* analyses which, unlike the analysis studied in this paper, consider a function separately for each calling context (see e.g. [Wilson 1997; Chatterjee et al. 1999; Cheng and Hwu 2000; Foster et al. 2000; Fähndrich et al. 2000; Das et al. 2001; Liang et al. 2001; Whaley and Lam 2004; Nystrom et al. 2004b]). While this can greatly improve precision, it is equivalent to fully inlining each function before performing the analysis and is, generally speaking, impractical for analysing large programs [Whaley and Lam 2004; Nystrom et al. 2004a]. Likewise, previous work has explored *flow-sensitive* analyses which, by taking into account the order of program statements, can also increase precision (see e.g. [Hasti and Horwitz 1998; Hind and Pioli 1998; Hind et al. 1999]). We refer the reader to [Hind 2001] for a more thorough survey of pointer analysis.

Several studies have looked at the relative merits of the three approaches to modelling aggregates, with the conclusion that field-sensitive analyses are considerably more precise than their field-based or field-insensitive counterparts [Yong et al. 1999; Diwan et al. 1998; Liang et al. 2001; Rountev et al. 2001; Lhoták and Hendren 2003]. However, it is important to realise that, since the problem differs between Java and C, it does not necessarily make sense to compare studies of Java with those for C. For example, previous results show that of the three, field-sensitive analyses are generally fastest when analysing Java [Lhoták and Hendren 2003; Rountev et al. 2001; Whaley and Lam 2002], but slowest when analysing C [Yong et al. 1999; Pearce et al. 2004a]. The main reason for this is that in C we can take the address of a field, whereas in Java we cannot. Thus, for C, using a field-sensitive analysis increases the number of potential pointer targets (often dramatically), leading to an increase in average set size (as shown in Section 5). For Java, however, the number of potential targets cannot go up with field-sensitivity — thus, average set size *can only go down.* For the analysis of C programs, there is little data available on the relative precision of the three methods. In [Yong et al. 1999], a field-sensitive analysis is shown to offer twice the precision of an insensitive analysis, although their benchmarks were much smaller than those used in this work. Nevertheless, our results from Section 5 do concur with this to some extent, although they also indicate the pay-off decreases with benchmark size. For field-based analyses, Heintze and Tardieu [Heintze and Tardieu 2001b] present data which appears to show a field-based analysis gives more precise results compared with an insensitive one. However, their data is described as "preliminary" and, in particular, we find their metric for comparison unsatisfactory because it has not been properly normalised (see the discussion of Figure 11 for more on this). Thus, the only real conclusion we draw from this work is that field-based analyses are faster than their insensitive counterparts. Unfortunately, we must also caution that we believe field-based analysis of C may be unsafe and this is discussed further in Section 6.1.

For Java, several studies show field-sensitive analyses are faster and more precise than the alternatives [Rountev et al. 2001; Lhoták and Hendren 2003; Whaley and Lam 2002]. As mentioned already, average set size might be one explanation for this. Another might be the proliferation of indirect function calls (due to virtual functions) in Java. This is relevant because a less precise analysis will identify more

		Dereference Sites (% of normalised total)						
		0	1	2	3	10	100	1000+
uucp	PWFI	18.0	18.0	2.5	1.5	2.5	58.0	0.0
	PW	15.0	16.0	7.7	5.1	0.22	56.0	0.0
make	PWFI	5.5	7.0	0.74	5.0	15.0	67.0	0.0
	PW	5.7	19.0	6.3	4.1	65.0	0.0	0.0
gawk	PWFI	6.2	7.7	6.3	4.2	26.0	0.92	49.0
	PW	7.0	23.0	21.0	6.9	42.0	0.0	0.0
147.vortex	PWFI	12.0	14.0	1.2	6.8	7.9	58.0	0.61
	PW	13.0	19.0	2.6	56.0	8.6	1.0	0.0
bash	PWFI	5.5	7.6	2.7	3.3	19.0	62.0	0.0
	PW	5.5	24.0	6.1	2.9	5.7	56.0	0.0
sendmail	PWFI	4.5	16.0	2.0	4.7	8.7	64.0	0.0
	PW	4.4	22.0	3.7	4.3	4.4	61.0	0.0
emacs	PWFI	24.0	17.0	27.0	2.7	7.9	21.0	0.53
	PW	25.0	25.0	32.0	10.0	7.6	0.37	0.03
126.gcc	PWFI	6.9	16.0	1.7	2.2	9.8	0.89	62.0
	PW	33.0	28.0	25.0	13.0	0.44	0.004	0.0
cc1	PWFI	5.6	11.0	0.68	2.3	1.8	10.0	68.0
	PW	6.6	19.0	3.5	4.4	0.5	0.0082	66.0
named	PWFI	3.4	3.6	3.9	28.0	1.4	1.2	58.0
	PW	5.1	8.4	1.3	28.0	2.2	2.7	52.0
gs	PWFI	33.0	2.7	1.2	3.1	3.7	0.82	55.0
	PW	35.0	7.2	3.0	2.1	0.25	0.3	52.0

Table IV. Actual percentages for the precision data shown in Figures 12 and 13. Each column indicates how many dereference sites (as a percentage of the total) have points-to sets of size X, where X lies between the column's boundary and up to, but not including, the next boundary.

targets for an indirect call, thus introducing more constraints. Furthermore, these constraints are considerably more expensive than those for dereferencing a data pointer, since they cause non-trivial value flow. Unfortunately, the overall picture is complicated by a study showing little difference in precision between a field-based and field-sensitive analysis, with the latter also running slower [Liang et al. 2001]. They argue that this should be expected from the strong encapsulation supported by Java. Indeed, this has some merit, if we consider that most fields in Java programs are read/written through get/set methods. Thus, context-insensitivity combines all distinct accesses to a particular field, yielding the same effect as the field-based approach. An example is given in Figure 14 and it seems that some simple strategies (such as inlining these get/set methods) would be very beneficial here. In fact, at least one analysis attempts something along these lines [Milanova et al. 2002], with promising results. Still, it seems unclear why other studies (e.g. [Lhoták and Hendren 2003]) of field-sensitivity have not encountered this problem and we can only speculate that they use some hidden technique to overcome it.

We now return to consider the relationship between our system and the comparable previous works. The most important of these, due to Yong *et al.* [Yong et al. 1999], is a framework covering a spectrum of analyses from complete field-insensitivity through various levels of field-sensitivity. The main difference from our

```
class myclass {
  private int *f1;
  private int *f2;
  public int *get(myclass *this) {
     return this->f1;                    get• ⊇ *(get_this + idx(f1))
  }
  public void set(int *v, myclass *this) {
    this->f1=v;                          *(set_this + idx(f1)) ⊇ v
  }
};

myclass a,b;
int *c,d,e;
a.set(&d);                               v ⊇ {d}, set_this ⊇ {a}
b.set(&e);                               v ⊇ {e}, set_this ⊇ {b}
c = a.get();                             get_this ⊇ {a}, c ⊇ get•
--------------------------------------------------------------------
Conclude                                 c ↦ {d, e}
```

Fig. 14. Illustrating how get/set methods affect field-sensitivity. Notice that the `this` variable is passed explicitly to each member function, reflecting what actually happens in practice. By combining information at function boundaries we are losing the advantages of field-sensitivity.

work is the approach taken to modelling field-addresses where, instead of integer offsets, string concatenation is used. To understand what this means, consider the following example which illustrates how fields are handled with their approach:

```
typedef struct { int *f1; int *f2; } aggr1;

aggr1 a,*b; int *p,c;
a.f2 = &c;              (1) a.f2 ⊇ {c}
b = &a;                 (2) b ⊇ {a}
p = b->f2;              (3) p ⊇ (*b)||f2
------------------------------------------------------------
                        (4) p ⊇ a.f2        (fdref_1, 2 + 3)
                        (5) p ⊇ {c}         (trans, 1 + 4)
```

Here, the $||$ operator can be thought of essentially as string concatenation, such that $\{a\}||b \Rightarrow a.b$ and $(*a)||b \Rightarrow c.b$, if $a \supseteq \{c\}$. Hence, the corresponding inference rules are:

$$[\mathit{fdref}_1]\ \frac{q \supseteq (*p)||f \quad p \supseteq \{a\}}{q \supseteq a.f} \quad [\mathit{fdref}_2]\ \frac{(*p)||f \supseteq q \quad p \supseteq \{a\}}{a.f \supseteq q}$$

Thus, we see that $p \supseteq (*b)||x$ replaces $p \supseteq *(b+k)$ from our system. While this difference appears trivial, there are hidden complications in dealing with certain uses of casting — *even when such uses are defined as portable within the ISO/ANSI C standard.* The relevant points from the standard can be summarised as follows:

(1) A pointer to a structure also points to the first field of that structure [ISO90 1990, 6.5.2.1]. As a result, the first field of a structure must be at offset 0.

(2) Accessing a union member after the last store was to a different member gives implementation-defined behaviour [ISO90 1990, 6.3.2.3]. Suppose we have "`union{int a;float b;} x`". Now, we can safely write and then read `x.a`, but we cannot safely write to `x.b` and then read `x.a`.

(3) As an exception to the above, if a `union` contains several structures whose initial members have *compatible types*, then it is permitted to access the common initial sequence of any of them [ISO90 1990, 6.3.2.3]. Note, it is sufficient for us to simply take *compatible types* to mean *identical types*, although the actual definition is more subtle. To understand the meaning of this point, suppose we have "`union {T1 a;T2; b} x`", where T1 and T2 are two `struct`'s whose first N members have identical types. Furthermore, suppose we assign to `x.a`. At this point, we may read any of the first N members of `x.b` and, as expected, they will have the same values as the first N members of `x.a`. This contrasts with the previous rule, which stated we may only read from `x.a`.

The first point above is fairly straightforward and the following example demonstrates that the string concatenation approach cannot model it correctly:

```
typedef struct { int *f1; int *f2; } aggr1;
```

	String Concatenation	Integer Offset
`aggr1 a,*q=&a;`	(1) $q \supseteq \{a\}$	(1) $q \supseteq \{a.f1\}$
`int c,*p;`		
`a.f1 = &c;`	(2) $a.f1 \supseteq \{c\}$	(2) $a.f1 \supseteq \{c\}$
`p = *((int*)q);`	(3) $p \supseteq *q$	(3) $p \supseteq *q$
	(4) $p \supseteq a$ $(deref_1, 1{+}3)$	(4) $p \supseteq a.f1$ $(deref_1, 1{+}3)$
		(5) $p \supseteq \{c\}$ $(trans, 2{+}5)$

What we see is that the string concatenation system is unable to correctly conclude $p \mapsto \{c\}$, whereas our system has no trouble. The issue here arises from the translation of "`&a`" into "a", instead of "$a.f1$". Unfortunately, the obvious solution of using the latter translation to resolve this introduces a further problem:

`aggr1 a,*q = &a;`	(1) $q \supseteq \{a.f1\}$
`int *p,c;`	
`a.f1 = &c;`	(2) $a.f1 \supseteq \{c\}$
`p = q->f1`	(3) $p \supseteq (*q)\|\|f1$
	(4) $p \supseteq a.f1.f1$ $(fdref_1, 1{+}3)$

Again, it is impossible to conclude $p \mapsto \{c\}$ from here. The real problem is that individual locations can have multiple names (e.g. `&a` and `&a.f1` above) and a system based solely on unique strings cannot easily deal with this. Another

```
typedef struct { int *f1; int *f2; } aggr1;
typedef struct { int *f3; int *f4; } aggr2;
```

	String Concatenation	Integer Offset
`aggr1 a;`		(1) $idx(a.f1) = 0$ (2) $idx(a.f2) = 1$
`aggr2 b;`		(3) $idx(b.f3) = 2$ (4) $idx(b.f4) = 3$
`void *c;`		(5) $idx(c) = 4$
`int d;`		(6) $idx(d) = 5$
`b.f3 = &d`	(1) $b.f3 \supseteq \{d\}$	(7) $b.f3 \supseteq \{d\}$
`c = &b;`	(2) $c \supseteq \{b\}$	(8) $c \supseteq \{b.f3\}$
`a = (aggr1) *c;`	(3) $a.f1 \supseteq (*c)\|\|f1$	(9) $a.f1 \supseteq *(c+0)$
	(4) $a.f2 \supseteq (*c)\|\|f2$	(10) $a.f2 \supseteq *(c+1)$
	(5) $a.f1 \supseteq b.f1$ $(fderef_1, 2+3)$	(11) $a.f1 \supseteq b.f3$ $(deref_4, 3+8+9)$
	(6) $a.f2 \supseteq b.f2$ $(fderef_1, 2+4)$	(12) $a.f2 \supseteq b.f4$ $(deref_4, 3+4+8+10)$
		(13) $a.f2 \supseteq \{d\}$ $(trans, 7+11)$

Fig. 15. This example illustrates an issue with the string concatenation approach to field-sensitivity. The problem arises because the type of "a" determines which field names are used in the concatenation, leading to constraints involving non-existing variables $b.f1$ and $b.f2$. An interesting point here is that, strictly speaking, this code has implementation-defined behaviour under the ISO/ANSI C standard. This is because the two `struct`'s must be wrapped in a union in order to be well-defined under the standard (see summary point 3 in Section 6). We have not done this purely to simplify the example.

example where this issue arises is given in Figure 15, where a different aspect of the ISO/ANSI standard is exploited (points 2 + 3 from the above summary). In this case, it is the ability to access the common initial sequence of two structures interchangeably which causes the trouble.

To overcome the issues involved with string concatenation, Yong *et al.* introduce three functions, *normalise, lookup* and *resolve*, whose purpose is to bridge the gap between different names representing the same location. This makes their system significantly more complicated and less elegant than our approach, which avoids these issues entirely. However, an important feature of their framework is the ability to describe both portable and non-portable analyses. The latter can be used to support commonly found, but undefined C coding practices which rely on implementation-specific information, such as type size and alignment. In contrast, our system as described cannot safely handle such practices. However, this could be done with only minor modification (i.e. using actual offsets instead of field numbers) and, in fact, Nystrom *et al.* claim to have done just this, although they do not discuss exact details [Nystrom et al. 2004b].

Yong *et al.* do not discuss the positive weight cycle problem, perhaps because it is only relevant to particular instances of their framework. Nevertheless, to obtain an equivalent analysis to ours, this issue must be addressed. Indeed, as we have mentioned, Chandra and Reps do so in their analysis, which they describe as an instance of the Yong *et al.* framework [Chandra and Reps 1999a; 1999b]. Their solution is to adopt a worse-case assumption about pointers in positive weight cycles (i.e. they point to every field of each target). Unfortunately, they do not provide any experimental data which could be used to compare with our system.

Finally, CQual also employs a field-sensitive pointer analysis for C [Foster et al. 1999]. This tool checks user-defined type annotations (e.g. `const`, `nonzero`) in C programs and can infer annotations when they are omitted. CQual was initially field-insensitive, but this proved inadequate and it was later extended to be (partially) field-sensitive [Johnson and Wagner 2004]. The approach taken is fairly conservative (especially with regard to assignment of `struct` variables) compared with that developed here, and this can lead to it being rather more imprecise. It is also unclear whether any attempt was made to make the approach sound with respect to the ISO/ANSI standard. Unfortunately, no experimental data is available on the benefits of using field-sensitivity within CQual which we could compare against our system. Nevertheless, CQual provides an interesting and somewhat unusual example of a client that stands to gain from the analysis we have developed.

6.1 Field-Based Pointer Analysis

At this point, we return to discuss the third technique for modelling aggregate variables, known as the field-based approach. The reader may have found it strange that this was omitted from our experimental study. The reason for this is that we have identified several problem cases (discussed below) relating specifically to field-based analysis of C. Previous works have failed to mention these before (see [Heintze and Tardieu 2001b; Andersen 1994]) and, while solutions may be possible, we feel that a more thorough examination is required.

Recall from Section 3.1, that under the field-based method, only one constraint variable is provided to represent every instance of a particular field of an aggregate type. To implement this type of analysis, it seems at first that our original inference system from Figure 1 can be used. The idea is to simply change the way in which C programs are translated into the constraint language:

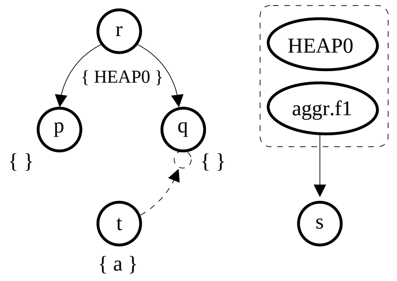
```
typedef struct { int *f1; int *f2; } aggr;
```

	Field-based	Field-insensitive
`aggr a,b;`		
`int c,d,*p;`		
`a.f1 = &c;`	$aggr.f1 \supseteq \{c\}$	$a \supseteq \{c\}$
`b.f1 = &d;`	$aggr.f1 \supseteq \{d\}$	$b \supseteq \{d\}$
`p = a.f1;`	$p \supseteq aggr.f1$	$p \supseteq a$

The $aggr.f1$ variable is provided to model every instance of the corresponding field. Hopefully, it is easy enough to see that the field-based analysis will conclude $p \mapsto \{c, d\}$, whilst the other gives the more precise $p \mapsto \{c\}$. Unfortunately, the following C code, whilst completely portable under the ISO/ANSI standard, is handled incorrectly by this approach:

```
aggr *p; int **q,*s,a; void *r

r=malloc(sizeof(aggr));     r ⊇ {HEAP0}
q = r;                      q ⊇ r
p = r;                      p ⊇ r
*q = &a;                    *q ⊇ t
                            t ⊇ {a}
s = p->f1;                  s ⊇ aggr.f1
```



Looking at the graph representation on the right, we can more easily understand the problem. When r's solution is propagated into q's, a new edge will be added from the temporary node t to *HEAP0* (due to the constraint $*q \supseteq t$). However, s reads from the special variable *aggr.f1* (which represents the corresponding field across all `aggr` instances), rather than from *HEAP0*. Thus, there will be no path from t to s, when in practice there should be, and this leads to the unsound conclusion that $s \not\mapsto \{a\}$. The problem stems from the fact that, under the ISO/ANSI standard, a pointer to a `struct` can be used interchangeably with a pointer to its first field (see summary point 1 on page 32). We have carefully constructed the above example to exploit this, leading to two constraint variables ($aggr.f1$ and $HEAP0$) representing the same physical object. To resolve this we must ensure updates to one are reflected in the other. That is, we must coalesce the two variables together. A good point to do this is when a `struct` pointer is assigned a value of a different type (e.g. at the statement "`p = r`" above).

Another difficulty arises when the same object represents different `struct`'s. For example:

```
typedef struct { int *f1; int *f2; double x;} aggr1;
typedef struct { int *f3; int *f4; int y;} aggr2;
typedef union { aggr1 a; aggr2 b; } aggr;

aggr x;
int c,d,*p;
x.a.f1 = &c;          aggr1.f1 ⊇ {c}
p = x.b.f3;           p ⊇ aggr2.f3
```

The problem here is that the field-based analysis does not conclude $p \mapsto \{c\}$. Again, this arises because two constraint variables ($aggr1.f1$ and $aggr2.f3$) represent the same physical object. As before, this can only be resolved by ensuring updates to one are reflected in the other. A reasonable solution might be to coalesce constraint variables representing `struct`'s which appear in the same union. Recall that, strictly speaking, the `union` is required for the above code to be considered properly ANSI compliant (see summary point 3 on page 33). If this were not the case, field-based analysis of C would (most likely) be completely unworkable because all `struct`'s could be used interchangeably with all others.

At this point, we remain uncertain whether all problem cases have been identified or not. Therefore, we believe a more thorough examination of field-based pointer analysis for C is required before it should be considered safe to use.

7. CONCLUSION

We have presented a novel approach to field-sensitive pointer analysis of C. While this is not the first solution to this problem, we argue it is the simplest and most elegant and have provided numerous examples to support this. We have developed an algorithm which implements our ideas and provided a complexity analysis which demonstrates, for the first time, that the problem of performing field-sensitive pointer analysis for C can be solved in $O(v^4)$ worse-case time. We have performed the largest experimental study to-date evaluating the trade-offs in performance versus precision of using field-sensitivity when analysing C programs. The results of this, which are reported here, demonstrate that field-sensitivity can offer a significant improvement in precision, albeit at some considerable computational cost.

While the overall conclusions of our experiments are positive, they also highlight a significant issue — namely that positive weight cycles are a major hindrance to efficient and precise field-sensitive analysis. Therefore, we feel that future work should consider this issue further and, hopefully, a satisfactory solution will be found. Another area of interest would be to investigate the effect on solving time of using the difference propagation technique with the field-sensitive analysis, which due to time constraints we have been unable to do.

Finally, we are pleased to say that Dan Berlin has independently integrated our technique for field-sensitive pointer analysis into the latest release (version 4.1) of the GNU Compiler GCC [Berlin 2005]. While his implementation differs a little from what we have presented here (e.g. it is intra-procedural and does not employ cycle detection), it is still the same fundamental algorithm underneath.

Acknowledgements. We would like to thank Dan Berlin for useful feedback on our algorithm and the anonymous TOPLAS referees for insightful and valuable comments on earlier drafts of this paper. This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through a PhD studentship and grant number GR/R15566.

REFERENCES

Aiken, A. 1994. Set constraints: Results, applications, and future directions. In *Proceedings of the workshop on Principles and Practice of Constraint Programming (PPCP)*. Lecture Notes in Computer Science, vol. 874. Springer-Verlag, 326–335.

Aiken, A. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming 35,* 2–3, 79–111.

Aiken, A. and Wimmers, E. L. 1992. Solving systems of set constraints. In *Proceedings of the IEEE symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 329–340.

Aiken, A. and Wimmers, E. L. 1993. Type inclusion constraints and type inference. In *Proceedings of the ACM conference on Functional Programming Languages and Computer Architecture (FPCA)*. ACM Press, 31–41.

Alur, R., Henzinger, T. A., Mang, F. Y. C., Qadeer, S., Rajamani, S. K., and Tasiran, S. 1998. MOCHA: Modularity in model checking. In *Proceedings of the conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1427. Springer-Verlag, 521–525.

Andersen, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen.

BALL, T. AND HORWITZ, S. 1993. Slicing programs with arbitrary control-flow. In *Proceedings of the Workshop on Automated and Algorithmic Debugging (AADEBUG)*. Lecture Notes in Computer Science, vol. 749. Springer-Verlag, 206–222.

BERLIN, D. 2005. Structure aliasing in GCC. In *Proceedings of the GCC Developers Summit*. 25–36.

BERNDL, M., LHOTÁK, O., QIAN, F., HENDREN, L. J., AND UMANEE, N. 2003. Points-to analysis using BDDs. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 196–207.

BINKLEY, D. 1998. The application of program slicing to regression testing. *Information and Software Technology 40,* 11-12, 583–594.

BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*. Lecture Notes in Computer Science, vol. 2566. Springer-Verlag, 85–108.

BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 196–207.

BOURDONCLE, F. 1993a. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 46–55.

BOURDONCLE, F. 1993b. Efficient chaotic iteration strategies with widenings. In *Proceedings of the conference on Formal Methods in Programming and their Applications*. Lecture Notes in Computer Science, vol. 735. Springer-Verlag, 128–141.

BURKE, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Language Systems (TOPLAS) 12,* 3, 341–395.

CHANDRA, S. AND REPS, T. 1999a. Physical type checking for C. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM Press, 66–75.

CHANDRA, S. AND REPS, T. 1999b. Physical type checking for C. Technical Report BL0113590-990302-04, Lucent Technologies, Bell Laboiatories.

CHATTERJEE, R., RYDER, B. G., AND LANDI, W. A. 1999. Relevant context inference. In *Proceedings of the ACM symposium on Principles of Programming Languages (POPL)*. ACM Press, 133–146.

CHEN, L.-L. AND HARRISON, W. L. 1994. An efficient approach to computing fixpoints for complex program analysis. In *Proceedings of the ACM Supercomputing Conference (SC)*. ACM Press, 98–106.

CHENG, B.-C. AND HWU, W.-M. W. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 57–69.

DAS, M. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 35–46.

DAS, M., LIBLIT, B., FÄHNDRICH, M., AND REHOF, J. 2001. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, 260–278.

DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. 1998. Type-based alias analysis. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 106–117.

DOR, N., RODEH, M., AND SAGIV, M. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 155–167.

EICHIN, M. W. AND ROCHLIS, J. A. 1989. With microscope and tweezers: An analysis of the internet virus of November 1988. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 326–343.

EMAMI, M., GHIYA, R., AND HENDREN, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 242–256.

FÄHNDRICH, M., FOSTER, J. S., SU, Z., AND AIKEN, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 85–96.

FÄHNDRICH, M., REHOF, J., AND DAS, M. 2000. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 253–263.

FECHT, C. AND SEIDL, H. 1996. An even faster solver for general systems of equations. In *Proceedings of the Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, 189–204.

FECHT, C. AND SEIDL, H. 1998. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *Proceedings of the European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 1381. Springer-Verlag, 90–104.

FLANAGAN, C. 1997. Effective static debugging via componential set-based analysis. Ph.D. thesis, Rice University.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for Java. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 234–245.

FOSTER, J. S., FÄHNDRICH, M., AND AIKEN, A. 1997. Flow-insensitive points-to analysis with term and set constraints. Technical Report CSD-97-964, University of California, Berkeley.

FOSTER, J. S., FÄHNDRICH, M., AND AIKEN, A. 1999. A theory of type qualifiers. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 192–203.

FOSTER, J. S., FÄHNDRICH, M., AND AIKEN, A. 2000. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 1824. Springer-Verlag, 175–198.

GABOW, H. N. 2000. Path-based depth-first search for strong and biconnected components. *Information Processing Letters 74,* 3–4 (May), 107–114.

GHIYA, R., LAVERY, D., AND SEHR, D. 2001. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 47–58.

GODEFROID, P. 1997. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, 476–479.

HARMAN, M., BINKLEY, D., AND DANICIC, S. 2003. Amorphous program slicing. *The Journal of Systems and Software (JSS) 68,* 1, 45–64.

HASTI, R. AND HORWITZ, S. 1998. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 97–105.

HEINTZE, N. 1994. Set-based analysis of ML programs. In *Proceedings of the ACM conference on Lisp and Functional Programming (LFP)*. ACM Press, 306–317.

HEINTZE, N. AND MCALLESTER, D. 1997a. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 261–272.

HEINTZE, N. AND MCALLESTER, D. A. 1997b. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 342–351.

Heintze, N. and Tardieu, O. 2001a. Demand-driven pointer analysis. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 24–34.

Heintze, N. and Tardieu, O. 2001b. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 254–263.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. 2003. Software verification with Blast. In *Proceedings of the Workshop on Model Checking Software*. Lecture Notes in Computer Science, vol. 2648. Springer-Verlag, 235–239.

Hind, M. 2001. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM Press, 54–61.

Hind, M., Burke, M., Carini, P., and Choi, J.-D. 1999. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS) 21,* 4, 848–894.

Hind, M. and Pioli, A. 1998. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 1503. Springer-Verlag, 57–81.

Hind, M. and Pioli, A. 2000. Which pointer analysis should I use? In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 113–123.

Holzmann, G. J. 1997. The Spin model checker. *IEEE Transactions on Software Engineering 23,* 5, 279–95.

Horwitz, S., Demers, A. J., and Teitelbaum, T. 1987. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica 24,* 6, 679–694.

ISO90 1990. ISO/IEC. international standard ISO/IEC 9899, programming languages — C.

Johnson, R. and Wagner, D. 2004. Finding user/kernel pointer bugs with type inference. In *Proceedings of the USENIX Security Symposium*. USENIX, 119–134.

Jones, J. A., Harrold, M. J., and Stasko, J. 2002. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM Press, 467–477.

Jones, N. D. and Muchnick, S. S. 1981. Flow analysis and optimization of lisp-like structures. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, 102–131.

Kodumal, J. and Aiken, A. 2005. Banshee: A scalable constraint-based analysis toolkit. In *Proceedings of the Static Analysis Symposium, SAS*. Lecture Notes in Computer Science, vol. 3672. Springer, 218–234.

Landi, W. 1992a. Interprocedural aliasing in the presence of pointers. Ph.D. thesis, Rutgers University, New Jersey, United States.

Landi, W. 1992b. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems 1,* 4, 323–337.

Lhoták, O. and Hendren, L. J. 2003. Scaling Java points-to analysis using SPARK. In *Proceedings of the conference on Compiler Construction (CC)*. Lecture Notes in Computer Science, vol. 2622. Springer-Verlag, 153–169.

Liang, D. and Harrold, M. J. 1999. Efficient points-to analysis for whole-program analysis. In *Proceedings of the European Software Engineering Confrence (ESEC) and ACM Foundations of Software Engineering (FSE)*. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag / ACM Press, 199–215.

Liang, D., Pennings, M., and Harrold, M. J. 2001. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proceedings of the ACM Workshop on Program Analyses for Software Tools and Engineering (PASTE)*. ACM Press, 73–79.

McKinley, K. S. 1994. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *Proceedings of the IEEE/ACM Supercomputing Conference (SC)*. ACM Press, 54–63.

Melski, D. and Reps, T. 1997. Interconvertibility of set constraints and context-free language reachability. In *Proceedings of the ACM workshop on Partial Evaluation and Program Manipulation (PEPM)*. ACM Press, 74–88.

Milanova, A., Rountev, A., and Ryder, B. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 1–11.

Myers, B. A. 1986. Visual programming, programming by example, and program visualization; A taxonomy. In *Proceedings of the ACM conference on Human Factors in Computing Systems (CHI)*. ACM Press, 59–66.

Nielson, F., Nielson, H. R., and Hankin, C. L. 1999. *Principles of Program Analysis*. Springer-Verlag.

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

Nystrom, E. M., Kim, H.-S., and Hwu, W.-M. W. 2004a. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 3148. Springer-Verlag, 165–180.

Nystrom, E. M., Kim, H.-S., and Hwu, W.-M. W. 2004b. Importance of heap specialization in pointer analysis. In *Proceedings of the ACM workshop on Program analysis for Software Tools and Engineering (PASTE)*. ACM Press, 43–48.

Padua, D. A., Kuck, D. J., and Lawrie, D. H. 1980. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers C-29,* 9 (Sept.), 763–776.

Padua, D. A. and Wolfe, M. J. 1986. Advanced compiler optimizations for supercomputers. *Communications of the ACM 29,* 12, 1184–1201.

Pearce, D. J. 2005. Some directed graph algorithms and their application to pointer analysis (online version available at `http://www.mcs.vuw.ac.nz/~djp`). Ph.D. thesis, Imperial College, London, United Kingdom.

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

Pearce, D. J. and Kelly, P. H. J. 2006. A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithms 11*, 1.7.

Pearce, D. J., Kelly, P. H. J., and Hankin, C. 2003. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the IEEE workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society Press, 3–12.

Pearce, D. J., Kelly, P. H. J., and Hankin, C. 2004a. Efficient field-sensitive pointer analysis for C. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM Press, 37–42.

Pearce, D. J., Kelly, P. H. J., and Hankin, C. 2004b. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal 12,* 4, 309–335.

Ramalingam, G. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages And Systems (TOPLAS) 16,* 5, 1467–1471.

Reiss, S. P. 1997. Cacti: a front end for program visualization. In *Proceedings of the IEEE symposium on Information Visualization (InfoVis)*. IEEE Computer Society Press, 46–50.

Reps, T. and Turnidge, T. 1996. Program specialization via program slicing. In *Selected Papers from the International Seminar on Partial Evaluation.* Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, 409–429.

Reynolds, J. C. 1969. Automatic computation of data set definitions. In *Proceedings of the Information Processing congress (IFIP)*. Vol. 1. North-Holland, 456–461.

Rountev, A. and Chandra, S. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 47–56.

Rountev, A., Milanova, A., and Ryder, B. G. 2001. Points-to analysis for Java using annotated constraints. In *Proceedings of the ACM conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM Press, 43–55.

Saha, D. and Ramakrishnan, C. R. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the ACM conference on Principles and Practice of Declarative Programming (PPDP)*. ACM Press, 117–128.

Schön, E. 1995. On the computation of fixpoints in static program analysis with an application to AKL. Technical Report R95-06, Swedish Institute of Computer Science. Nov.

Shapiro, M. and Horwitz, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1–14.

Shmueli, O. 1983. Dynamic cycle detection. *Information Processing Letters 17,* 4 (Nov.), 185–188.

So, B., Moon, S., and Hall, M. W. 1998. Measuring the effectiveness of automatic parallelization in SUIF. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC)*. ACM Press, 212–219.

Sridharan, M., Gopan, D., Shan, L., and Bodik, R. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the ACM conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 59–76.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 32–41.

Su, Z., Fähndrich, M., and Aiken, A. 2000. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the symposium on Principles of Programming Languages (POPL)*. ACM Press, 81–95.

SUIF2. The SUIF 2 research compiler, Stanford University, http://suif.stanford.edu.

Systä, T., Yu, P., and Müller, H. 2000. Analyzing Java software by combining metrics and program visualization. In *Proceedings of the IEEE conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 199–208.

Tarjan, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1,* 2, 146–160.

The Vis Group. 1996. VIS: a system for verification and synthesis. In *Proceedings of the conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, 428–432.

Vivien, F. and Rinard, M. 2001. Incrementalized pointer and escape analysis. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 35–46.

Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 3–17.

Whaley, J. and Lam, M. S. 2002. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, 180–195.

Whaley, J. and Lam, M. S. 2004. Cloning-based context-sensitive pointer alias analysis using Binary Decision Diagrams. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 131–144.

Wilson, R. P. 1997. Efficient context-sensitive pointer analysis for C programs. Ph.D. thesis, Stanford University, California, United States.

Wilson, R. P. and Lam, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 1–12.

Wolfe, M. J. 1982. Optimizing supercompilers for supercomputers. Ph.D. thesis, Deptartment of Computer Science, University of Illinois at Urbana-Champaign, United States.

Yong, S. H., Horwitz, S., and Reps, T. 1999. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 91–103.