Application of the Pointer State Subgraph to Static Program Slicing *

Running title: Pointer State Subgraph

David W. Binkley †
binkley@sst.ncsl.nist.gov

James R. Lyle
jimmy@sst.ncsl.nist.gov

Technology Administration
National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899

Abstract

A new technique for performing static analysis of programs that contain unconstrained pointers is presented. The technique is based on the *pointer state subgraph*: a reduced control flow graph that takes advantage of the fact that in any program there exists a smaller program that computes only the values of pointer variables. The pointer state subgraph is useful in building static analysis tools. As an example the application of the pointer state subgraph to program slicing is considered. Finally, some experimental results, obtained using the ANSI-C slicer Unravel, are reported. These results show a clear reduction in the time taken to compute data-flow information from programs that contain pointers. They also shown a substantial reduction in the space needed to store this information.

Key Words. control-flow graph; data-flow analysis; pointer-state subgraph; program slicing; static analysis tools

---

*supported in part by the National Institute of Standards and Technology and the Defense Information Systems Agency under contract DNR046115

†also at Loyola College in Maryland

1 Introduction

Static analysis tools help programmers understand software that must be enhanced or debugged. Unfortunately, unconstrained pointers, as found in languages such as ANSI C[1], make static analysis difficult. Two reasons for this are the imprecision of this information and the use of dynamic storage allocation. Imprecision comes from indirect references (assignments or uses) through pointer variables: if a pointer potentially points to one of many objects, it is unknown which object is actually referenced. Dynamic storage allocation poses two difficulties. First, an unbounded number of objects may be created. Second, since these objects may contain pointers, an unbounded number of pointers may be created.

The *pointer state subgraph* introduced in this paper assists in performing program analysis of program with pointers. It reduces the space required to represent the pointer state by using a subgraph of the program's control-flow graph. This graph is annotated with a function that describes the behavior of pointer variables.

We show the usefulness of the pointer state subgraph by applying it to the problem of program slicing in the presence of pointers[29, 30, 17]. The *slice* of a program, with respect to program point p and variable x, consists of all statements and predicates of the program that may affect the value of x at point p. Program slicing has applications to program debugging[18, 19, 13, 14], program testing[2, 4], program integration[10, 5], parallel program execution[30], software metrics[23], reverse engineering[3], and software maintenance [8]. Frank Tip provides a survey of program slicing including these uses [27, 28].

Section 2 formalizes the pointer state subgraph. Section 3 demonstrates its usefulness in building static analysis tools by applying it to the problem of program slicing in the presence of pointers. Section 4 presents experimental results collected using Unravel, an ANSI-C slicer that uses the pointer state subgraph [20]. Section 5 discusses some related work, and finally, Section 6 summarizes our work.

2 The Pointer State Subgraph

The pointer state subgraph is motivated by the observation that any program dealing with pointers contains a program that computes the pointer state. That is, it is possible to project out of the original program an *embedded pointer state program* that contains only those statements necessary to compute the values (*i.e.*, addresses) held by pointer variables. This embedded pointer state program could, for example, be obtained by taking the slice of the program with respect to all definitions of pointer variables in the program.

The *pointer state subgraph* (PSS) is much smaller than the full control-flow graph since only assignments to pointer variables change the pointer state; thus, every program statement does not have to be analyzed to determine this state. This section describes the PSS in three steps: first, it defines the PSS and the language it presently supports, then it describes how the PSS is constructed, and, finally, it describes the construction of the *pointer state function*, which annotates the PSS.

2.1 Definition

The *pointer state subgraph* is a reduced *control-flow graph* annotated with a *pointer state function*:

DEFINITION (Control-Flow Graph). A *control-flow graph* is a directed graph whose nodes represent source program components (*e.g.*, `assignment` statements, or the predicate of an `if` or `while` statement). In addition, there are two special nodes labeled "entry" and "exit" that provide a "frame" for the other nodes. Control-flow graph edges represent the flow of control through the program. For example, the node representing an `if` statement has two outgoing edges that represent the condition of the `if` statement being `true` or `false`.

DEFINITION (Pointer State Subgraph). The *pointer-state subgraph* (PSS) is a subgraph of the control-flow. It contains those control-flow-graph nodes that modify the pointer state (*e.g.*, `p = &m`, but not `*p = 5`, which modifies an `int` not a pointer). The edges of the PSS are paths of edges from the control-flow graph: an edge connects node n to node m if a path in the control-flow graph connects n to m and no node on this path is in the PSS.

Each node of the PSS is annotated by a *pointer state function*, which describes the *pointer state* for each pointer variable visible at the node in the current scope. This function maps variables to *objects*: storage locations allocated either statically by variable declarations or dynamically by calls to `malloc` (`malloc` is used to represent any function that returns a newly allocated block of storage). `NULL` is one of the possible objects.

DEFINITION (Pointer State Function). For pointer variable v at node n, the pointer state function $P(n, v)$ contains the set of objects that v may point to (*i.e.*, objects that v may dereference to). $P(n, v)$ is broken down "level-by-level;" thus $P_0(n, v)$ is the value obtained starting with v and dereferencing zero times (*i.e.* v); P_1 (n, v) is the value obtained starting with v and dereferencing one time (*i.e.*, $*v$); and so on. More formally

$$\begin{aligned} P_0(n, v) &= \{v\} \\ P_1(n, v) &= \{\text{object } o \mid v \text{ holds the address of } o\} \\ P_k(n, v) &= \{x \mid x \in P_1(n, y) \text{ and } y \in P_{k-1}(n, v)\}. \end{aligned}$$

EXAMPLE. In the program shown in Figure 1 the pointer state changes three times (the pointer state immediately after each statement is shown to the right of the statement). All three changes are for variable `min`, which is initially undefined, and eventually contain all of the formal parameters. The PSS for this example includes the nodes for the statements on Lines 7–11.

[Insert Figure 1 Here]

In the absence of dynamically allocated objects, a pointer variable can take on a finite (and usually small) number of values. Thus, it is possible to precisely identify non-heap objects. However, approximation techniques are necessary for tracking dynamically allocated objects. The reason for this is illustrated by the following example.

```
1   typedef struct List_struct
2   {
3       int          value;
4       List_struct *next;
5   } List;
        ⋮
6   List  *node;
        ⋮
7   node = (List *) malloc (sizeof(List));
```

Each time Line 7 is executed, `node` receives the address of a new block of memory from the heap. It is not possible to statically determine how many blocks are allocated or to track individual blocks. This implies that an approximation that summarizes dynamically allocated storage blocks must be introduced. For example, the heap could be viewed as an array where assignments to individual members are treated as both an assignment and a reference to the entire array.

The approximation used in annotating the PSS divides objects by allocation site (and thus by type): all objects allocated by the same statement are summarized as one *pseudo-variable*. For pseudo-variables that represent structures, additional pseudo-variables are created for each field of the structure. Pseudo-variables are given special treatment when assigned. A definition of a pseudo-variable is also treated as a reference to the pseudo-variable, because these variables represent more than one block of storage. (In the following discussion, the address of a variable may refer to the address of a pseudo-variable and, thus, to a block of memory returned by `malloc`.) Using this approximation, there are a finite number of possible values for a pointer variable, because there are a finite number of allocation sites. The approximation is safe. A proof of its safety follows from a simplification of the arguments given by Pfeiffer [25].

While we assume adherence to good stylistic programming practices in pointer usage, we realize that in practice we must consider the implications of *creative usage* with constructs such as casts and pointer arithmetic. The PSS supports casts used in such a way that an object's type is referenced consistently. However, if casts are used to take advantage of the memory layout, then the PSS cannot be correctly constructed. For example, consider a structure with an integer as its first member. On some systems it is possible to cast a pointer to the structure to a pointer to an integer and then reference the first structure member through the pointer. The PSS for such a program would see two distinct objects rather than one. Pointer arithmetic can be used on an object that is an array so long as the pointer remains within the bounds of the array. Pointer arithmetic cannot be used to take advantage of the compiler's memory layout for moving from one object (or structure member) to another. In addition, Unravel presently does not handle programs that contain *setjmp*, *longjump*, or pointers to functions.

2.2 Constructing the PSS

Conceptually, for program P, the PSS is constructed from the *embedded pointer state program* of P. This program contains all statements of P that modify the value (*i.e.*, the address) contained in a pointer. It does not contain the statements that compute the values contained in these addresses. The actual construction begins with P's control-flow graph and casts out nodes that do not change the pointer state. This is done by applying the following three rules until none can be applied.

1. If node n has a single successor k, a single predecessor j, and does not change the pointer state then eliminate n and connect j to k.
2. If node n has multiple successors, node j is where all the paths from n rejoin (j dominates n), and all the nodes between n and j have been removed then remove n and j and connect n's predecessor to j's successor.
3. The framing nodes "entry" and "exit" are never removed.

EXAMPLE. The control-flow graph and PSS for the program in Figure 2 are shown in Figure 3. Nodes 2, 4, 11 and 12 in the control-flow graph do not change the pointer state and are eliminated by Rule 1. Since node 4 is eliminated, Rule 2 eliminates the branch and join nodes (Nodes 3 and 5), and connects Node 1 to Node 6. Note that the same PSS is obtained by slicing at Line 13 with respect to *min*.

[Insert Figure 2 Here]

[Insert Figure 3 Here]

2.3 Annotating the Pointer State Subgraph

After the PSS is constructed, each node is annotated with the pointer state function $P_k(n, v)$. (Note that in the implementation P_k for $k > 1$ is not recorded but rather is derived when needed from P_1.) P_1 is constructed by first identifying statements where object addresses are introduced and then tracking these addresses as they are propagated along the edges of the PSS. Addresses are introduced into the pointer state from two sources: taking the *address of* a variable (*i.e.*, by applying the C & operator) and obtaining storage from the heap. Object addresses are propagated along the edges of the PSS using the rules in the following table. In each rule, the pointer state output of a node is the pointer state input to the node with the entry for $P_1(n, a)$ (or $P_1(n, y)$ in the case of the last two rules) replaced as shown. To handle loops, these rules are iterated until a fixed point is reached. Since the number of addresses is finite and the computation of the pointer statement function is monotonically increasing, this computation is guaranteed to terminate.

Statement	Propagation Rule
$a = \&x$	$P_1(n, a) = \{x\}$
$a = b$	$P_1(n, a) = P_1(n, b)$
$a = *b$	$P_1(n, a) = \bigcup P_1(n, x),\ \forall x \in P_1(n, b)$
$a = **b$	$P_1(n, a) = \bigcup P_1(n, x),\ \forall x \in P_2(n, b)$
$a = *^k b$	$P_1(n, a) = \bigcup P_1(n, x),\ \forall x \in P_k(n, b)$
$*a = \&x$	$P_1(n, y) = P_1(n, y) \cup \{x\},\ \forall y \in P_1(n, a)$
$*^k a = \&x$	$P_1(n, y) = P_1(n, y) \cup \{x\},\ \forall y \in P_k(n, a)$

(The notation $*^k b$ denoted b dereferenced k times.)

EXAMPLE. In Figure 3, $P_1(6, min)$ is assigned the value $\{w\}$ by the first rule. Further on, the pointer state for min at Node 9, $P_1(9, min)$, is the union of the pointer states on the incoming flow edges from Nodes 7 and 8. Thus, $P_1(9, min) = P_1(7, min) \cup P_1(8, min) = \{w\} \cup \{c\} = \{w, c\}$.

3 Applying the PSS to Program Slicing

This section discusses application of the PSS to static program slicing in the presence of pointer variables. Before doing so, some definitions are introduced and a brief discussion of program slicing in the absence of pointer variables is given.

Defs(n). The set of variables defined (assigned to) at statement n.

Idefs(n). The set of *indirect definitions* at statement n. Each element is a pair containing a variable and a level of indirection. Level 0 indicates a direct assignment to the variable, Level 1 indicates that the variable contains the address of the object assigned, and so on. (Note that $defs(n) = \{v \mid (v, 0) \in idefs(n)\}$).

Irefs(n). The set of *indirect references* at statement n. Each element is a pair containing a variable and the level of indirection. Level 0 indicates a direct reference to the variable, Level 1 indicates that the variable contains the address of the object referenced, and so on. (Note that $refs(n)$, defined below, is $\{v \mid (v, 0) \in irefs(n)\}$.

Program Slice. P' is a slice of program P taken with respect to variable v at statement n (written $S_{<n,v>}$), iff

1. P' is derived from P by deleting zero or more tokens from P, and
2. when P and P' are executed on the same input for which P terminates, the same values are produced for v immediately before n by P and P'.

The slice taken with respect to a set of variables is defined as the union of the slices taken with respect to each variable in the set.

Refs(n). The set of variables referenced at statement n.

Slicing Criterion. A tuple $<n, v>$ where n is a program statement and v is a variable or a set of variables.

Succ(n). The set of *successor* statements for statement n. This include all statements whose execution can immediately follow n.

3.1 Program Slicing in the Absence of Pointer Variables

Program slicing is a program decomposition technique based on extracting statements relevant to some sub-computation in a program. The relevant sub-computation is identified by a *slicing criterion*, which specifies a set of program variables and a location in the program. A slice can be computed beginning from the slicing criterion by including each predecessor that assigns a value to any variable within the slicing criterion, and generating new slicing criterion for the predecessor by deleting any assigned variables from the original slicing criterion and adding any referenced variables.

More formally, statement n is included in a slice if the slicing criteria for a successor of n includes a variable in $defs(n)$ ($i.e.$, $S_{<succ(n),v>}$ include n if $v \in defs(n)$). If n is included then the slice also includes the slices taken with respect to the each variable in $refs(n)$ ($i.e.$, $S_{<n,refs(n)>}$) and the control statements that directly control n's execution ($i.e.$, $S_{<c,refs(c)>}$ for each control statement c whose execution determines whether or not n is executed.

3.2 Program Slicing in the Presence of Pointer Variables

This section considers simple pointers to non-structures and then pointers to structures. In each case, we first describe the slices necessary to capture the execution behavior of a pointer reference that is included in the slice and then the conditions under which a statement with an indirect assignment is included in the slice.

Indirect Reference by Pointers

If a statement with an indirect reference is included in a slice, each variable that may be referenced, including the pointer variable, is relevant to the computation and further slices for each must be generated. For the simple case of a single level of indirection:

$$n: \quad \cdots * b \cdots$$

the necessary slices are

$$S_{<n, P_0(n,b)>} \cup (S_{<n, P_1(n,x)>} \text{ where } (x, 1) \in \mathit{irefs(n)}).$$

For example, if the pointer state for b is $P_1(n, b) = \{E, F\}$, then the new slices for everything that $*b$'s value may depend on are $S_{<n,E>}$, $S_{<n,F>}$, and $S_{<n,b>}$ (recall that $P_0(n, b) = \{b\}$).

Now consider a general reference having k levels of indirection:

$$n: \quad \cdots *^k\, b \cdots$$

If statement n is included in the slice then the following slices should also be included:

$$S_{<n, P_i(n,x)>} \text{ where } (x, k) \in \mathit{irefs}(n) \text{ and } 0 \leq i \leq k.$$

Indirect Assignment by Pointers

To determine if an expression statement with an indirect assignment should be included in a slice, every possible location that the assignment may modify must be known. This is complicated by the use of multiple levels of indirection.

First, consider the special case of a statement n with an assignment through one level of indirection:

$$n: \quad *a = \cdots$$

For the slicing on criteria $<\mathit{succ}(n), c>$, statement n is included in the slice if $P_1(n, a)$ includes c. In addition to slices necessary to capture computation on the right hand side of n (*i.e.*, those described above), slices are generated to capture the statements that give a (not $*a$) its value. For example, if the statement $a = \&c$ had given a its value then the slice $S_{<n,a>}$ captures the assignment of $\&c$ to a. If a may point to a variable other then c then the slice $S_{<n,c>}$ is also included to capture the other assignments to c.

Now consider the general case for k levels of indirection. Here each intermediate level must be captured. Thus, the statement

$$n: \quad *^k a = \cdots$$

is included in the slice $S_{<succ(n),v>}$, if v is any of the variables to which $*^k a$ may point (*i.e.*, n is included if $v \in P_k(n,a)$ and $(a,k) \in idefs(n)$).

The additional slices generated must include only relevant intermediate indirect references. This is captured by the *relevant intermediate indirect references* function $R_{i,k}(n,v,x)$, which returns the set of intermediate pointers that may be used at level i of indirection for an assignment at statement n using k levels of indirection through variable x to variable v. Function R effectively prunes away indirect pointers in P that are not relevant to a particular slicing criterion. In other words, $P_i(n,v)$ is the set of variables that v may point to. However, only those members of $P_i(n,v)$ that may dereference to x (after $k-i$ levels of indirection) are relevant.

$$R_{i,k}(n,v,x) = \{r \mid r \in P_i(n,a) \text{ and } v \in P_{k-i}(n,r)\}$$

EXAMPLE. Figure 5 shows the pointer state represented as a directed graph for the variable a to three levels of indirection for the program fragment in Figure 4. In Figure 5, an edge from node n to node m means that n could point to m. Figure 5 also shows the pruned sets of indirect pointers for the criteria $<succ(n),w>$ and $<succ(n),z>$.

[Insert Figure 4 Here]

If $S_{<succ(n),v>}$ includes statement n then in addition to the slices used to capture the computation of the right-hand side, the following slices are taken:

$$S_{<n,R_{i,k(n,a,v)}>} \text{ where } 0 < i < k \text{ and } (a,k) \in \mathit{idefs}(n)$$

This is sufficient if there is only one member of $P_k(n,a)$ because the object referenced by the pointer is unique and is definitely killed by the assignment. However, if there is more than one member of $P_k(n,a)$, then any analysis must account for the possibility that the object is only potentially modified. Thus, if $\exists(a,k) \in \mathit{idefs}(n)$ such that $|P_k(n,a)| > 1$ then the slice $S_{<n,v>}$ must be included.

[Insert Figure 5 Here]

References to Structure Members via Pointer

This section, considers *pointer chain expressions*, references to fields of structures accessed through pointer variables. Pointer chain expressions are composed of a pointer variable p followed by one or more structure field-names f_i separated by "$\rightarrow$" (the dereference then field select operator). For pointer chains, it is necessary to identify the variables associated with a field of the pointer chain. This is done by the field function F:

$$\begin{aligned} F(v,f) = {} & \text{the variable that represents field } f \text{ of variable} \\ & v \ (\textit{i.e.}, \text{"}v.f\text{"}). \end{aligned}$$

First, consider the case of a length one pointer chain expression

$$n: \quad \cdots p \rightarrow f \cdots$$

If n is included in a slice, then slices on p and all the variables $x.f$ where x is in $P_1(n,p)$ must be included (the entries of $P_1(n,p)$ are assumed to be structures). These slices are $S_{<n,v>}$ and $S_{<n,F(x,f)>}$, where $x \in P_1(n,p)$.

When a slice includes a statement with a general pointer chain expression, such as

$$n: \quad \cdots (p \rightarrow f_1 \rightarrow \cdots \rightarrow f_k) \cdots$$

then the slice must also included slices with respect to all the variables to which the end of the chain (f_k) may point, and all variables to which the intermediate links of the chain ($p, f_1, \cdots, f_{k-1}$) may point. An example will help explain the need for these slices.

```
    typedef struct
    {
        int c;
    } alpha;

    typedef struct
    {
        int    value;
        alpha *b;
    } beta;

    beta   *a;
    beta   r,s,t,u;
    alpha w,x,y,z;

    foo( ··· )
    {
n:      ···a → b → c···
    }
```

Assume that at statement n the pointer state function is as follows:

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
P_1(n,a) & = & \{s,t\} & F(s,b) & = & s.b & F(y,c) & = & y.c \\
P_1(n,r.b) & = & \{w,x,z\} & F(r,b) & = & r.b & F(z,c) & = & z.c \\
P_1(n,s.b) & = & \{w,y\} & F(t,b) & = & t.b & & & \\
P_1(n,t.b) & = & \{y,z\} & F(u,b) & = & u.b & & & \\
P_1(n,u.b) & = & \{w,x,y\} & F(w,c) & = & w.c & & &
\end{array}
$$

If statement n is included in a slice, then slices on any variable pointed to by $a \to b \to c$ must be included to account for variables that the entire chain may reference. These slices are $S_{<n,w.c>}$, $S_{<n,y.c>}$, and $S_{<n,z.c>}$. Slice $S_{<n,w.c>}$, for example, is included because a points to s ($s \in P_1(n,a)$) and $F(s,b) = s.b$, which points to w ($w \in P_1(n, s.b)$). This example chain has one intermediate link, $a \rightarrow b$. Slices for this link are $S_{<n,s.b>}$ and $S_{<n,t.b>}$.

In general, if statement n is included in the slice then the following slices are needed to capture the relevant statements for each pointer chain expression

$$S_{<n,w_i>}, \text{ where } w_0 = \{p\}, \text{and}$$
$$w_i = \{F(z, f_i) \mid z \in P_1(n, r) \text{ and } r \in w_{i-1}\}.$$

Assignment to Structure Members via Pointer

Consider first the case of a single field reference

$$n:\ p \rightarrow f = \cdots$$

Statement n is included in the slice $S_{<succ(n),v>}$ if v is one of the variables identified by $p \rightarrow f$. Thus, n is included if

$$x \in P_1(n, p) \text{ and } v = F(x, f)$$

In the general case of a statement with k field references

$$n:\ \ p \rightarrow f_1 \rightarrow \cdots \rightarrow f_k = \cdots$$

Statement n is included in the slice $S_{<succ(n),v>}$ if v is one of the variables identified by $p \rightarrow f_1 \rightarrow \cdots \rightarrow f_k$. Thus, n is included if

$$v \in X_k, \text{ where } X_0 = \{p\}, \text{ and}$$
$$X_i = \{F(x, f_i) \mid x \in P_1(n, X_{i-1})\}$$

Parallel to the absence of structures, in addition to slices necessary to capture computation on the right hand side of n (described above), slices are generated to capture the statements that give p, $p \rightarrow f_1$, $\ldots$, $p \rightarrow f_1 \rightarrow \cdots \rightarrow f_{k-1}$ (but not $p \rightarrow f_1 \rightarrow \cdots \rightarrow f_k$) their values.

3.3 Other Uses of the PSS in Program Slicing

A further example of the use of the PSS in program slicing is illustrated by Figure 6 where the PSS can reduce the number of statements in a slice. Consider the slice on s at Line 20 in Figure 6. Weiser's initial definition of a slice required the slice to be executable. Later definitions relax this requirement. They define a slice to contain all the statement that may affect the computation at a particular point in the program [11].

The "relaxed" slice with respect to $<20, s>$ consists of the following statements: $\{1, 2, 6, 9, 14, 15, 17, 18, 21\}$. However, by omitting Lines 11 and 12, the slice is not executable ($*z$ may be undefined and cause a run-time error when attempting to assign to $**z$ on Line 18). The variable z could contain either the address of w or the address of y. If z contains the address of w then executing Line 18 sets the value of s since w points to s. However, if z contains the address of y then executing Line 18 has no effect on the value of s. Lines 11 and 12 are not relevant to the computation of s; anything could be executed so long as y is given the address of a variable not used in the slice. Since only w contains the address of s, Line 18 is only needed in the slice when the condition z points to w.

[Insert Figure 6 Here]

A program slicer producing executable slices has two possible solutions:

1. Add statements to the slice that are irrelevant to the slicing criterion but are required to keep the slice executable.

2. Add conditions to guard against executing statements that would cause a run-time error when the slice is executed.

Having the PSS available a slicer can determine that Line 18 has no effect on the variable s. A tool using the PSS can alert the user of any indirect assignments that may be to more than one possible object. The user could instruct the tool to assume assignment to a particular object and display the result.

4 Experimental Results

Table 1 illustrates the effectiveness of the PSS in computing pointer state information. The data was collected using Unravel (an ANSI-C program slicer developed at NIST [20]) on a Sun Sparc Station 2 with 64MB memory. The first group of programs were used by Pande *et. al.,* to study def-use associations for single level pointers (*i.e.*, no pointers to pointers) [24]. They include several UNIX library functions and programs from the TACTIC [22] test suite used to examine data-flow based test coverage for C programs. The second group of programs contains multi-level pointers (*i.e.*, pointers to pointers) and pointer chains. They include the implementation of several abstract data-types (`pdg` is a program dependence graph implementation [7]) and two more UNIX library functions: `qsort` and `strtod`. The last two programs are part of the Unravel implementation itself: the `analyzer` program provides the X11 interface for Unravel; the `project` program projects a source program given a set of nodes in a slice.

The data collected is summarized in Table 1. For reference this table includes the number of source lines (non-blank lines after removing comments) and compile time (without optimization) for each program. Note that the compile time should not be directly compared with the other times as it includes the time taken to write the compiled code to disk.

The data in the table clearly demonstrate that the PSS reduces the time and space needed to compute the pointer state. For the single-level pointer programs, the average size reduction is 75%; thus the PSS is, on average, a quarter the size of the control-flow graph (CFG). The size reduction for the multi-level pointer-programs, is smaller 56.4%, but still substantial. The reduction in size has significant impact on the time taken to compute pointer state information. For both the single-level pointer programs and the multi-level pointer program the average reduction is 54%.

[Insert Table 1 Here]

5 Related Work

There is a vast literature on data-flow analysis for programs with pointers. Most of this work has concentrated on discovering aliases: when two expressions refer to a common memory location. This is a hard problem; Even within a single function, the problem of precisely computing the structure aliases visible is NP-complete [16]. Thus, all techniques use heuristics and introduce some form of approximation. To simplify the problem, some approaches also omit certain language features. For example higher-order functions, function pointers, multi-level pointers, casting, or C union types.

We discuss below a two approaches: k-limited and additional information. Other approaches use abstract interpretation [21], path expressions, which contain nodes labeled by regular expression [16], and access-expressions [6]. In general, increased algorithm complexity has been able to increase the precision of pointer analysis. However, at some point the costs start to out way the benefits (for example the algorithm described by Hummel et. al. may require an exponential amount of time in the worst case, while the observed running time is $O(n^4)$ [12]). In the same way Ruf revisited the usefulness of flow sensitive interprocedural aliasing information [26], the annotation of the PSS and the results discussed in Section 4 illustrate how a straitforward approach can efficiently produce good results.

The k-limited approach is most commonly applied to lisp-like languages [15, 9]. It approximates the objects reachable from each pointer variable by a graph that is limited to depth k. The depth of the structures represented in the functions annotating PSS can vary from 1 to the number of allocation sites in the program. Thus, the approximation can be better or worse than that obtained using k-limiting.

Additional information comes in the form of programmer supplied axioms about the pointer state or programmer supplied annotations added to the program. For example, the input to the algorithm presented by Hummel et. al. requires including aliasing axioms in addition to the program being analyzed [12]. It is unclear that programmers can or will generate the correct axioms. Those that are unobtrusive, such as modified type declarations provide the greatest chance for success.

Finally, our techniques may work well with others. For example, summarizing with a single pseudo variable all the allocations that take place at a particular program point leads to impression when the allocated memory is used for multiple purposes. Annotations about these uses would improve our technique.

Alternatively, it is possible to improve the accuracy of the functions annotating the PSS (using, for example, the techniques of Deutcsh [6] or Pande et. al. [24]). The trade-off mentioned above can again be seen between our work and the work of Pande et. al who provide a through study of def-use associations for C programs with single-level pointers. Their technique is more accurate than ours (assuming it can be extended to multi-level pointers and recursive data structures) and more costly. Determining if the benefits out weigh the costs is the topic of future research.

6 Summary

This paper presented the pointer state subgraph. It is more efficient to annotate this graph with pointer state information than the standard control-flow graph. Pointer state information is useful in building static analysis tools. We demonstrate this by showing how the pointer state subgraph is used by Unravel to compute program slices from programs that contain pointers. Finally, we reported results obtained by running Unravel on a collection of C programs that contains pointers and pointer chains. The data clearly illustrates the effectiveness of the pointer state subgraph in reducing the space need to compute pointer state information and in reducing the time taken to compute this information.

Other uses of the PSS include providing an interactive static analysis tool where interesting subsets of the pointer state can be selected by the user. This could be useful when debugging a program with pointers where the user needs to understand how a given object is computed and manipulated.

Acknowledgments

We wish to thank Bill Landi for providing the single-level pointer code example.

References

[1] ANSI. American National Standard for Information Systems – Programming Language – C. Technical Report ANSI X3.159-189/FIPS PUB 160, American National Standards Institute, 1430 Broadway New York, New York 10018, December 1989.

[2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, January 10-13 1993.

[3] J. Beck. Program and interface slicing for reverse engineering. *Proceeding of the Fifteenth International Conference on Software Engineering*, 1993.

[4] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. *IEEE International Conference on Software Maintenance*, pages 251–260, October 1995. IEEE Computer Society Washington, DC.

[5] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[6] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond *k*-limiting. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, Florida, June 1994.

[7] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[9] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, 24(7):28–40, July 1989.

[10] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):35–46, January 1990.

[12] J. Hummel, L. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 218–229, Orlando, Florida, June 1994.

[13] J. Hwang, M. Du, and C. Chou. Finding program slices for recursive procedures. *Proceedings of the Twelveth International Computer Software and Applications Conference (* COMPSAC *'88)*, pages 220–227, October 1988.

[14] J. Jiang, X. Zhou, and D.J. Robson. Program slicing for C: The problems in implementation. *Proceedings of the Conference on Software Maintenance*, 1991.

[15] N.D. Jones and S.S. Muchnick. Program flow analysis, theory, and applications. In *Chapter 4, FLow Analysis and Optimization of LISP-like Structures*, pages 102–131. Prentice-Hall, 1982.

[16] J.R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation, Atlanta, GA, June, 1988, ACM SIGPLAN Notices*, 1988.

[17] J. R. Lyle and D. W. Binkley. Program slicing in the presence of pointers. *Proceedings of the 3RD Annual Software Engineering Research Forum*, November 11–12 1993.

[18] J. R. Lyle and M. D. Weiser. *Experiments in Slicing-Based Debugging Aids.* In *Empirical Studies of Programmers,* Elliot Soloway and Sitharama Iyengar, eds. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[19] J. R. Lyle and M. D. Weiser. Automatic program bug location by program slicing. *Proceeding of the Second International Conference on Computers and Applications*, pages 877–882, June 1987.

[20] J.R. Lyle, D.R. Wallace, J.R. Graham, K.B. Gallagher, J.P. Poole, and D.W. Binkley. A case tool to evaluate functional diversity in high integrity software. Technical Report IR 5691, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995.

[21] A. Neirynck, P. Panangaden, and A.J. Demers. Efficient analysis in higher-order languages. *International Journal of Parallel Programming Integration*, 18(1):1–37, 1989.

[22] T.J. Ostrand. Data-flow testing with pointers and function calls. *Proceedings of the Pacific Northwest Software Quallity Conference*, 1990.

[23] L. Ott and J. Thuss. The relationship between slices and module cohesion. *International Conference on Software Engineering*, May 1989.

[24] H. Pande, W. Landi, and B. Ryder. Interprocedurel def-use associations for **C** systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[25] P. Pfeiffer. *Dependence-based representations for programs with reference variables.* PhD thesis, Madison, WI, 1991.

[26] E. Ruf. Context-insesitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, California, June 1995.

[27] F. Tip. *Generation of Program Analysis Tools.* PhD thesis, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, 1995.

[28] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, September 1995.

[29] M. Weiser. Programmers use slicing when debugging. *CACM*, 25(7):446–452, July 1982.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

```
/* call: square_min (&a,&b,&c)
    return square of the minimum of a, b, and c */
int square_min (int *x, int *y, int *z)
{
    int *min, square, w;
           /* Pointer State for:  x  y  z  min    */
    min = x;                  /*  a  b  c  -      */
    if (*y < *min)            /*  a  b  c  a      */
        min = y;              /*  a  b  c  a      */
    if (*z < *min)            /*  a  b  c  a|b    */
        min = z;              /*  a  b  c  a|b    */
    w = *min;                 /*  a  b  c  a|b|c  */
    square = w * w;           /*  a  b  c  a|b|c  */
    return square;            /*  a  b  c  a|b|c  */
}
```

Figure 1: Pointer State Function Example

```
Figure 3
 Node
Number
          /* call: square_min_var (&a,&b,&c)
             return the square of the minimum of a, b, and c */
          int square_min_var (int *x, int *y, int *z)
          {
             int *min, square, w;

  2          w = *x;
  3          if (*y < w)
  4              w = *y;
  6          min = &w;
  7          if (*z < *min)
  8              min = z;
 10          w = *min;
 11          square = w * w;
 12          return square;
          }
```

Figure 2: An example program

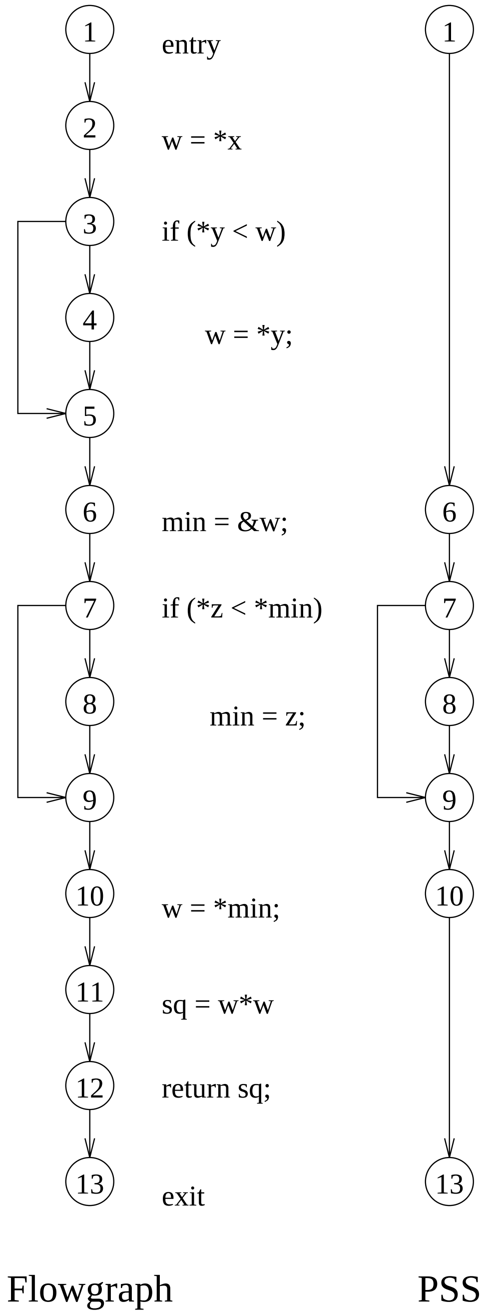

Figure 3: Control-flow graph and PSS for the program shown in Figure 2.

```
{
    int w, x, y, z, *e, *f, *g, *h, *i, *j,  **b, **c, **d, ***a;
    ⋮
    a = cond() ? (cond() ? &b : &c) : &d;
    ⋮
    b = cond() ? &e : &f;
    c = cond() ? &g : &h;
    d = cond() ? &i : &j;
    ⋮
    e = cond() ? &w : &x;
    f = &x; g = &y; h = &z; i = &w;
    j = cond() ? &w : &z;
    ⋮
n:  ***a = ···
    ⋮
}
```

Figure 4: Pointer Code Fragment

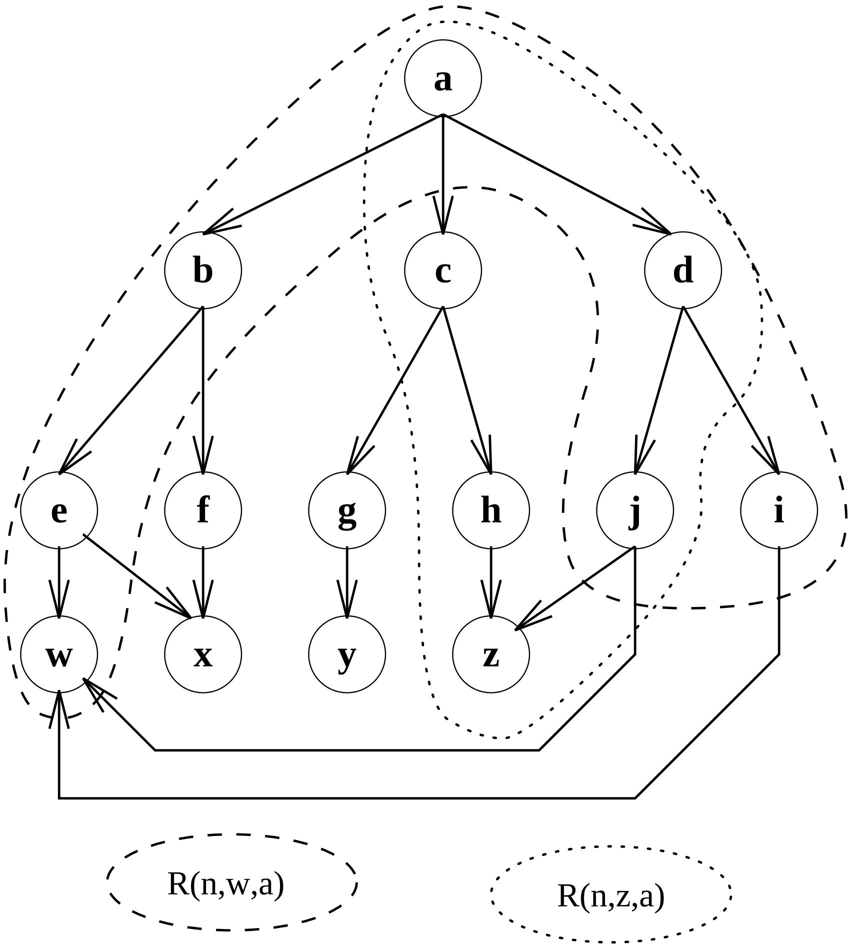

Figure 5: Pruned Pointer State For $***a$

```
main()
{
    int a,b,s,u,x;
    int *w,*y,**z;

    s = 1;
    a = 2;
    b = 3;
    scanf (“%d %d”,&x,&u);

    if (x) y = &a;
    else y = &b;

    if (u) z = &y;
    else z = &w;

    w = &s;
    **z = 4;

    printf (“a %d b %d s %d\n”,a,b,s);
}
```

Figure 6: Missing Execution Example

name	source lines	Size: CFG nodes	PSS nodes	size reduction	Time: without PSS	with PSS	savings	compile time
strcpy	11	10	3	70%	<1ms	<1ms	-	800ms
strncat	18	23	3	87%	<1ms	<1ms	-	800ms
crypt	235	236	17	93%	6ms	<1ms	-	1600ms
random	180	172	79	54%	5ms	3ms	40%	1300ms
parser	874	640	41	94%	8ms	2ms	75%	3700ms
fixoutput	342	340	35	90%	4ms	1ms	75%	1800ms
weather	118	109	85	22%	3ms	2ms	33%	1000ms
pattern	116	93	27	71%	1ms	< 1ms	-	1000ms
jacobi	163	164	9	95%	2ms	< 1ms	-	1300ms
sor	166	168	16	90%	2ms	< 1ms	-	1400ms
ht	189	145	82	43%	5ms	3ms	40%	1400ms
pdg	1143	704	55	92%	38ms	9ms	76%	5000ms
set	357	289	106	63%	9ms	4ms	56%	1700ms
qsort	171	302	197	35%	4ms	4ms	0%	1700ms
strtod	1664	1871	1241	34%	210ms	47ms	78%	8300ms
analyzer	3959	694	303	56%	48ms	15ms	69%	22300ms
project	407	305	85	72%	10ms	4ms	60%	2100ms

Table 1