Effective Whole-Program Analysis in the Presence of Pointers*

Darren C. Atkinson and William G. Griswold
Department of Computer Science & Engineering, 0114
University of California, San Diego
San Diego, CA 92093-0114
{atkinson,wgg}@cs.ucsd.edu

Abstract

Understanding large software systems is difficult. Traditionally, automated tools are used to assist program understanding. However, the representations constructed by these tools often require prohibitive time and space. Demand-driven techniques can be used to reduce these requirements. However, the use of pointers in modern languages introduces additional problems that do not integrate well with these techniques. We present new techniques for effectively coping with pointers in large software systems written in the C programming language and use our techniques to implement a program slicing tool.

First, we use a fast, flow-insensitive, points-to analysis before traditional data-flow analysis. Second, we allow the user to parameterize the points-to analysis so that the resulting program slices more closely match the actual program behavior. Such information cannot easily be obtained by the tool or might otherwise be deemed unsafe. Finally, we present data-flow equations for dealing with pointers to local variables in recursive programs. These equations allow the user to select an arbitrary amount of calling context in order to better trade performance for precision.

To validate our techniques, we present empirical results using our program slicer on large programs. The results indicate that cost-effective analysis of large programs with pointers is feasible using our techniques.

1 Introduction

1.1 Motivation

Large software systems are difficult to understand. These systems have typically evolved over several years, and as systems evolve, their structure degrades [11]. This degenerated structure increases maintenance costs, since a single change may no longer be localized to a module, but rather is dispersed throughout the code [13]. Consequently, a maintainer needs global, rather than local, knowledge about the system in order to correctly reason about the effect of a proposed change. For large software systems, gathering this global knowledge can be a time-consuming activity. Furthermore, large systems typically use sophisticated language constructs in their implementation, such as function pointers. Although the use of these constructs is often necessary to achieve good performance or to ease implementation, they can hinder program understanding. The combination of degraded structure and use of sophisticated language constructs makes large systems especially difficult to understand, maintain, and enhance.

Automated semantic tools have been proposed as a solution to this maintenance problem since they can eliminate some tedious, error-prone tasks. For example, a program slicer [20] helps determine the effects of a proposed change by computing the set of statements that might affect the value of a given variable. As another example, an invariant checker infers facts about the state of the program and checks those facts against assertions provided by the programmer [12]. Unfortunately, the use of pointers in modern programming languages hinders the construction of whole-program analysis tools that are both efficient and sufficiently precise.

First, the use of pointers negates the performance benefits of demand-driven techniques [2, 3, 7] since determining the memory locations possibly referenced through a pointer typically requires a global analysis over the program. For example in the C programming language [8], all files must be analyzed to account for the use of pointers in initializers for static variables, regardless of whether a file contains a function that might be reachable during subsequent data-flow analysis.

Second, in a flexible and performance-oriented language such as C, the way in which pointers are used complicates performing a points-to analysis that is sufficiently precise for the subsequent data-flow analysis. For instance, the use of specialized memory allocators can reduce precision by hindering the analysis's ability to accurately model heap storage. Furthermore, pointer arithmetic on arrays and structures limits the points-to analysis's ability to accurately discern distinct memory locations. Because these aggregates often store pointers to functions, the imprecise analysis can result in an overly conservative call-graph, degrading both the performance and precision of interprocedural data-flow analysis.

Finally, pointer usage can complicate performing the subsequent data-flow analysis. Pointers to local variables are commonly used in C programs to emulate passing parameters by reference, which the language itself does not support. Pointers to local variables in the presence of recursion require changes to the traditional bit-vector equations for data-flow analysis [1], since different activations of a local variable may be referenced in functions other than the function in which it is declared. If the equations are not changed, the data-flow analysis will be in error.

1.2 Approach

Our previous work concentrated on effectively performing whole-program analysis on systems written in a language without point-

*This research is supported in part by NSF grant CCR-9508745.

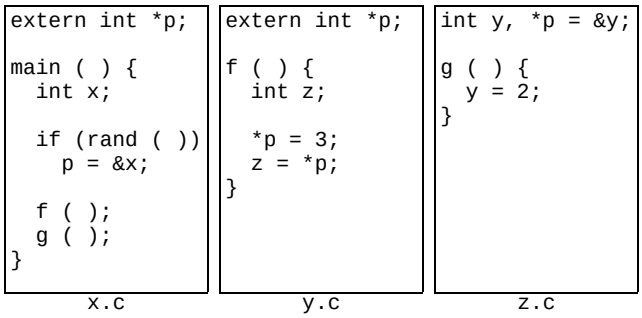

Figure 1: An example C program consisting of three files. The pointer variable p is referenced in all files.

ers [2, 6]. The approach transparently demand-derives all representations. However, representations such as the abstract syntax tree (AST) that are infrequently used are discarded and recomputed when needed to reduce memory requirements. Representations such as the call-graph that are costly to compute, but inexpensive to store, are persistently retained on disk. Also, by allowing the user to parameterize the data-flow analysis, we found that significant time and space could be saved by avoiding the computation of unnecessary data-flow information.

Our goal is to adapt this work to effectively cope with pointers in large systems written in languages that are low-level, flexible, and performance-oriented. Specifically, we wish to construct an efficient slicing tool for large C programs. In attempting to integrate pointers into our approach, three issues arose. First, how should we integrate points-to analysis with our demand-driven data-flow analysis framework? Second, how does the use of pointers in large systems impact the precision of an analysis? Third, how can data-flow analysis be performed in the presence of pointers to local variables and recursion?

First, because points-to information cannot be demand-derived, we use Steensgaard's near-linear time, context-insensitive, flow-insensitive, points-to analysis algorithm [18]. To avoid the cost of an extra pass over the program, the points-to analysis is "piggybacked" with the demand construction of the control-flow graph (CFG). Since our previous approach saves the call-graph to disk to speed-up subsequent executions of the tool, the saved call-graph now includes calls to functions through function pointers (as computed by the points-to analysis).

Second, to increase the precision of pointer analysis without unnecessarily increasing algorithmic complexity, we allow the user to parameterize the analysis. Since our tool is designed for interactive program understanding rather than for batch compilation, we can take advantage of information provided by the tool user. For example, the user might specify that the program being analyzed has only strict ANSI-compliant function prototypes, helping to more accurately determine which functions may be called through a function pointer. Such information cannot be obtained automatically by the tool without substantial additional cost, if at all.

Finally, we derive space-efficient data-flow equations for dealing with pointers in the presence of recursion and pointers to local variables. In our previous work, we presented a tunable mechanism, called context-depth, for selecting an arbitrary amount of calling context during data-flow analysis. Our new data-flow equations extend this work for C programs with pointers.

In the following sections, we discuss the effect of pointers on whole-program analysis, and how we effectively cope with these problems. We discuss our solution to integrating points-to analysis with demand-driven techniques. We present empirical results with user-parameterization of the points-to analysis and its effect on program slicing, and specifically the construction of the call-graph for a program. We also present an efficient implementation of our data-flow equations and compare the implementation to a "naive" implementation.

Our results indicate that our techniques permit slicing 200,000 line programs in seconds or minutes, rather than hours. In particular, parameterization of the points-to analysis can dramatically increase the number of points-to classes, often by an order of magnitude or more. For programs that use function pointers heavily, the precision of the constructed call-graph can be substantially improved. As a result, program slices can be computed an order of magnitude faster and contain far fewer statements. Otherwise, we have found that the subsequent data-flow analysis is mostly insensitive to the improvement in precision, typically yielding a 3% decrease in the number of statements.

2 Integrating Points-To Analysis with Demand-Driven Analyses

Demand-driven techniques attempt to save space and time by computing only those data-flow facts and portions of supporting representations that are necessary to perform the analysis [2, 3, 7]. In this way, large programs can be handled more economically since the amount of information computed and stored is greatly reduced. Effective demand-driven analysis depends upon quickly identifying which portions of a representation are required next and efficiently computing those portions. In backward slicing, for example, it is necessary to quickly identify all the callers of a procedure and efficiently construct the CFG for those calling procedures [2]. Because determining the callers of a procedure requires a global analysis of the program, our demand-driven approach saves the call-graph to disk for future invocations of the slicing tool.

Depending on the algorithm chosen, points-to analysis for large programs can require a large, possibly prohibitive, amount of time and space. Unfortunately, there are problems with either demanding or persistently storing points-to information. We discuss these problems and then present a hybrid compute-and-store solution.

Points-to information is not efficiently computable on demand because computing the effects of any particular pointer reference can require a global analysis of the program. For example, Figure 1 shows a small C program consisting of three source files. If a backward program slice is started at the assignment to `z` in function `f()` of file `y.c`, the points-to set of variable `p` is needed. There is an assignment to `p` in function `main()` in file `x.c`, so `x.c` must be analyzed. Ignoring pointers, a demand-driven slicer would not need to examine this file unless the user requested that slicing should continue into the calling function. File `z.c` must be also examined. Although function `g()` is not reachable during a backward data-flow analysis from `f()`, the file contains the initialization for `p` in a static initializer.

An alternative to demand-driven analysis is to persistently retain the points-to information in a database, as we do with the call-graph. This approach is attractive since the call-graph requires pointer information for computing the effects of calls through function pointers anyway. However, storing the pointer information presents several difficulties. First, a representation would be needed for referencing an arbitrarily nested variable declared within a function. Second, the CFG's three-address statements and associated temporaries would need to be constructed in a reproducible order from one tool invocation to the next. Finally, the database must be recomputed if any variable in the program changes, not just if the call structure changes. Although none of these difficulties is overwhelming, their net complexity led us to consider a third alternative.

Our approach is to demand all the points-to information on invocation of the first slice, employing three techniques to minimize the impact of the required global analysis.

- We use Steensgaard's near-linear time, context-insensitive, flow-insensitive, points-to analysis, which models storage as equivalence classes of locations [18].[1] Although not as precise as some techniques, its time–space characteristics are superior and the difference in precision is often not reflected in the subsequent data-flow analysis [15].
- To avoid an extra pass over the program to perform the global analysis, we piggyback the computation of points-to information with the construction of the portions of the CFG required for the subsequent data-flow analysis. The call-graph, which was formerly used to demand only those portions of the CFG reachable from the initial slicing criterion, is now used to determine which portions of the CFG are needed only for points-to analysis and can therefore be discarded immediately after use.
- To maintain the call-graph's effectiveness in the demand-driven analysis, the graph saved to disk includes the effects of calls through function pointers, as determined by the points-to analysis. Since the points-to analysis is flow-insensitive—in particular it does not require a call-graph—performing points-to analysis in a prior pass to gather function pointer information adds little complexity to the implementation of the program slicer.

Using this approach on GCC, our largest program, computing the points-to information and other supporting data for the call-graph requires 52 seconds and 63 MB of space. Although the CFG itself would require only 52 MB if fully constructed, the total savings due to CFG discarding can be substantial. For example, if only half of the CFG needs to be retained for slicing, the savings of 26 MB might be sufficient for the entire analysis to reside in main memory, eliminating paging and thus improving overall execution time.

3 Use of Pointers in Large Systems

C provides powerful, albeit low-level, language features like type casting, pointer arithmetic, and function pointers. Programmers often use these sophisticated language constructs in order to improve performance and ease implementation. For example, all of our example systems use an array of function pointers to implement a *dispatch table*—a table in which the key is an integer value designating an operation and the corresponding value is the address of a function that performs that operation. Sometimes this dispatch table is an array of structures that contain pointers to functions.

The way that such aggregates are allocated and manipulated often causes their points-to classes to be merged, yielding imprecise resolution of pointer references during analysis. The use of type casting, pointer arithmetic, and custom memory allocators are especially problematic. The resulting merges often cascade, yielding unacceptably conservative results. For example, if two separate dispatch tables become merged by the analysis, then the structures they contain become merged, and finally the fields within the structures are merged. Such collapsing of points-to classes not only results in overly conservative resolution of pointer references during data-flow analysis, but also during the computation of the program call-graph. As a result, the subsequent data-flow analysis can be both very inefficient and imprecise, since the analysis will traverse a large number of function calls that cannot actually occur during program execution.

Although some of these problems with points-to class merging can be overcome by using a context-sensitive points-to analysis, the analysis may then become too expensive [2]. Furthermore, the increase in precision may be small [14] or may not yield substantially better data-flow information [16].

[1] Our implementation treats relational operators differently from arithmetic operators since the former do not yield a pointer value. This fact is mentioned in the reference but not included in its equations.

```
void (*p) ( );
int (*q) ( ), y;

int main ( ) {
  (*p) (1);
  (*q) (2, "a");
  (*q) (3, &y);
}

void f (int x) {
  y = x;
}

int g (int x) {
  return x;
}

int h (int x, void *p) {
  return x + *(int *) p;
}

int i (int x, char *p) {
  return *p + x;
}
```

Figure 2: A program fragment using function pointers.

One way to improve precision without unacceptable cost is to permit the tool user to provide additional, easily specified information to improve the precision of the points-to analysis. Our approach is to allow the user to parameterize several aspects of the analysis in terms of the language syntax. For example, the user might specify that the function named "xmalloc" should be treated as a memory allocator. Alternatively, if the cost of distinguishing structure members is judged to be too high for the expected benefit, the user can choose to have them not be distinguished.

Since our tool is designed for program understanding, we allow the user to provide both optimistic (i.e., "unsafe") and conservative information. In tasks such as compilation or automatic parallelization, the meaning of the program must be preserved. However in program understanding, the tool user is attempting to gain knowledge about the system or provide reassurance of an assumption made about the system. As long as the tool user is readily aware that certain parameters may yield unsafe information, we feel the ability to provide optimistic information is justified.

We have developed several options for parameterizing the analysis that the user of our program slicer may enable. Each has its own effect on the points-to analysis.

Function prototype filtering: In many cases we found it too costly in time and space to compute sufficiently precise points-to sets for function pointers. Consequently, we turned to using type information to achieve better results. In particular, the user may specify whether the program uses weakly (old-style "K&R" C) or strongly ANSI-compliant function prototypes. Function prototypes provide additional typing information for static semantic checking by ensuring that the type and number of formal and actual arguments agree. After retrieving the points-to set for a function pointer reference, the prototypes of the resultant set of function definitions are compared against the prototype implied by the function call. The prototypes are computed from the actual function definition and the function call since the program may be ANSI-compliant, but not be written using ANSI-style prototypes. Enabling this option does not affect the construction of the points-to classes, but rather filters the classes based on the calling statement, reducing the number of functions that may be called for a given function call expression.

For example, Figure 2 presents a small program using function pointers. Let us assume that the four functions, `f()`, `g()`, `h()`, and `i()`, have all been merged into the same points-to class and that both `p` and `q` point to this class. By filtering on the prototypes of the function call and definition, the first function call in `main()` can only refer to function `f()` since the other three functions return an `int` and `p` is declared to return `void`. The second call can refer to either function `h()` or function `i()` since they both require two

`return from f()`

$D_{exit} = D_{exit} \cup (D_i \cap S)$ (3.1)

`x := y`

if $x \in D_i$ then
$D_i = D_i - \{x\} \cup \{y\}$ (3.3)

`*p := x`

if $*p \cap D_i \neq \phi$ then
$D_i = D_i \cup \{p, x\}$ (3.5)

`call to f()`

$D_i = (D_i - S) \cup (D_{entry} \cap S)$ (3.2)

`x := *p`

if $x \in D_i$ then
$D_i = D_i - \{x\} \cup *p \cup \{p\}$ (3.4)

D = *variables of interest*
S = *all global (static) variables*
$*p$ = *points-to set of variable p*

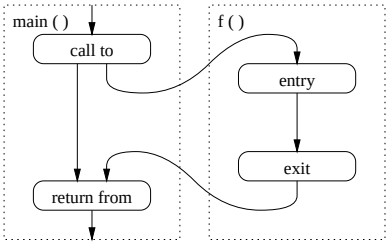

Figure 3: Traditional data-flow equations for slicing in the presence of recursion without pointers to local variables. Sets are subscripted with the program point to which they refer. Sets that are not subscripted are the same for all program points. The current statement has program point i. Unless otherwise noted, a set passes through a program point unchanged. The example CFG shows the nodes relevant to function calls.

arguments and a string is assignable to the `void` pointer argument in function `h()`. The third call can only refer to function `h()` since a pointer to `int` is assignable to a `void` pointer, but not to a `char` pointer.

Private memory allocators: Because large programs typically process lots of information, they can dynamically allocate several thousand objects. Since calling the standard C `malloc()` function for each object incurs an overhead, many large systems employ their own memory allocator. Implementing a private memory allocator is not difficult since C's own memory allocator, `malloc()`, is itself implemented in C. Typically, a private memory allocator is just a "wrapper" around calls to the underlying memory allocator such as `malloc()` that allocates larger blocks of memory and then doles them out in appropriately sized pieces. Another type of simple allocator, such as the `xmalloc()` function in our example programs, is one that merely calls `malloc()` and then checks the return value to see if virtual memory has been exhausted.

The use of private memory allocators can reduce the precision of points-to analysis. One method of modeling dynamically created storage is to treat each static call to `malloc()` as though it has its own heap, which is modeled as if it were a single large array of bytes from which objects are allocated. As a consequence, all pointers that are associated with a particular `malloc()` call site are treated as referencing the same memory address (assuming array indices are ignored). This approach, which we use, is simple to implement and often yields adequate precision [18]. For a program using the `xmalloc()` function described above, the program will contain several distinct calls to `xmalloc()`, but only one static call to `malloc()` (by `xmalloc()` itself). Thus, using `xmalloc()` rather than `malloc()` results in modeling memory as a single large, shared array, rather than several separate ones. All pointers to dynamically allocated memory are treated as referencing the same memory location. In effect, the points-to analysis is penalizing the programmer for writing efficient and modular code.

With the private memory allocator option, the user specifies the names of those functions that should be treated as if they were calls to `malloc()`. Each call site of the memory allocator is treated as if it returned the address of a temporary static variable, rather than all calls returning the address of the same variable.[2] This information may be optimistic if the user is unsure which functions serve as memory allocators. The effect of using this option is to introduce more addresses to the points-to analysis, resulting in more points-to locations.

Commutativity of array operator: In C, the array operator is commutative because array references are semantically equivalent to pointer addition, which itself is commutative. The expressions `a[i]` and `i[a]` are identical. However, the second form in which the pointer value appears within the brackets is generally not used. Normally, the points-to analysis must assume that this second form can be used. Thus, if the index `i` is used to index two distinct locations `a` and `b`, they become indistinguishable to the analysis since it assumes that `a` and `b` may be the indices and that `i` is the pointer value. (Use of the cast operator in C to override the type system makes this possible.) By enabling the array option, the user precludes this possibility, resulting in the two locations not being merged. Using this option may yield unsafe information unless the tool user is sure that the second form of array indexing is never used. However, a special case exists if `a` is declared as an array rather than a pointer. Since an array variable is constant and cannot be assigned, we can be sure that `i` is the index and that `a` is always the pointer value.

Structure members: A typical C program uses structures quite heavily to model objects. A structure may contain pointers to other objects of different types. Since these objects are of different types, they are likely distinct. Although distinguishing structure members in points-to analysis can increase precision, sometimes the benefit is small and is not justified by the higher cost. In the worst case the analysis may require exponential time when structure members are distinguished [17].

To permit managing the time and space complexity of the analysis, our analysis distinguishes structure members only when chosen as an option by the user. Thus, references to `a.x` and `a.y` are normally treated as a reference to `a`. As a result, any objects pointed to by the `x` and `y` members are merged into a single points-to class.

When the user enables the structure members option, two such locations are not merged.[3] A structure assignment is treated as assigning the individual members. If pointer arithmetic is performed on a structure pointer, then the members are "collapsed" (i.e., the points-to sets for all members are merged and the structure is thereafter treated as a single location) since a dereference through the generated pointer value may assign to any member or possibly multiple members. A variant structure or "union" type in C is considered to be a structure whose fields are already collapsed.

Although each parameter can individually improve the precision of the points-to analysis, when combined the results are magnified. For example in GCC, the array commutativity parameter prevents merging of separate arrays of structures, and the structure and prototype parameters distinguish the individual structure components. However, in some cases, the effects of one parameter will

[2] Memory deallocators such as `free()` need to be treated similarly to avoid merging classes due to parameter passing.

[3] The points-to analysis is similar to that described in [17], but assumes that adjacent structure members are infinitely far apart and thus does not take into account the size of an access during the analysis.

subsume the effects of another. In particular, for function pointers we have found that filtering the points-to classes using strong prototypes is the most beneficial (Section 5).

4 Performing Data-Flow Analysis in the Presence of Pointers to Locals

In the absence of either recursion or pointers to local variables, interprocedural slicing is simple and well-understood. However, for a language such as C that provides both of these features, traditional data-flow analyses may yield unsafe results. We first present the traditional data-flow equations and then describe how the analysis may be in error.

Figure 3 presents traditional data-flow equations for backward slicing. At each program point, D represents a data-flow set. At each assignment statement (Equations 3.3–3.5), some set of variables, $defs$, are defined and another set of variables, $uses$, are used. If some variable in $defs$ is also in D, then the killing definitions of $defs$ are removed from D and $uses$ are added to D. Otherwise, D remains unchanged. In Equation 3.5, we assume that all assignments through a pointer dereference are preserving, since the points-to set may contain more than one variable, but only one variable is in fact updated. The remaining two equations (Equations 3.1 and 3.2) show how the D set is manipulated across function calls using the CFG shown. The S set contains all static (global) variables in the program and is used to partition a set into its local and global variables.[4]

Figure 4 shows a small C program with a recursive function `f()`. Consider performing a backward program slice at the assignment to `y` in `f()`. Consequently, we are looking for an assignment to `x`, which is a local variable to `f()`. Proceeding backward through the function, the next statement examined is the recursive call to `f()`. When slicing into the recursive call, we need to remove local variables from the data-flow sets (Equations 3.1 and 3.2) [9]. This step is necessary to avoid finding a definition of the same local variable but with a *different activation* in the recursive call. Continuing our example, the data-flow sets become empty after removing `x`, resulting in no further information being added to the slice by the recursive call. However, we have now erroneously excluded the last assignment in `f()` from the slice. This statement is clearly an assignment to `x`, although which activation of `x` we do not know. This is a *missing definition* of `x`. On the other hand, if we do not remove `x` from the data-flow set, then we can find false definitions of `x` because these definitions may in fact refer to other activations of `x`. If any of these definitions is a killing definition, then we have a *false kill* of `x` and our analysis is also in error.

The main difficulty with incorporating both recursion and pointers to local variables is that the two features require that local variables be treated in contradictory ways when slicing into a function call. To ensure correctness, local variables must be removed for the data-flow sets in the presence of recursion, but must remain in the data-flow sets in the presence of pointers to local variables. We introduce two new data-flow sets, N and P, in our equations, shown in Figure 5. The N set is used to solve the problem of a missing definition, and the P set it used to solve the problem of a false kill.

Missing definition: In the presence of recursion, a local variable must be removed from D in order to avoid finding a killing definition of the same variable but with a different activation. However, the local variable may be referenced through a pointer in a called function. In our approach, local variables are removed from D and

[4] We use the term *local variable* to mean an *automatic variable* in C. Similarly, the term *global variable* should be read as *static variable*. Since C overloads the use of the "static" keyword, we use the terms local and global variable instead.

```
f ( ) {                    int y, *p;
  int x = 1;
                           main ( ) {
  if (g ( ))                 f ( );
    p = &x;                  printf ("%d\n", y);
                           }
  if (g ( )) {
    x = 2;
    f ( );                 g ( ) {
  }                          int z;

  y = x;                     scanf ("%d", &z);
  *p = 3;                    return z;
}                          }
```

Figure 4: Example program showing pointers to local variables in the presence of recursion.

placed into N of the called function (Equation 5.1.c). This process is similar to the mapping and unmapping of nonvisible variables [4, 10]. Consider an assignment made by dereferencing a pointer variable p (Equation 5.5). If the points-to set of p overlaps with N, then a local variable declared in another function has been defined. However, which activation of the variable that has been defined is unknown, and therefore the assignment must be treated as a preserving definition. The assignment statement should be added to the slice and the variables used at that statement added to D, but the variables defined are not removed.

The N set models the transitive closure of the program stack, but only for local variables of interest. When a function is called, the local variables of interest to the caller are added to N of the called function along with the caller's N set (Equation 5.1.c). Since N only contains *nonlocal* local variables of interest, it need only be examined in statements containing an assignment by means of a pointer dereference (Equation 5.5).

False kill: If a local variable is referenced out of scope by means of a pointer dereference, we cannot be certain which activation of the variable is actually used. To be safe, we must therefore assume that all possible activations are referenced. The activations of a local variable can be thought of as an array, with the stack pointer referring to the last element of the array. If a local variable is referenced out of scope, we do not know the array "index" and so must assume all activations are referenced. Thus, a local variable referenced out of scope is treated just like an array—any definition is always a preserving definition. In our approach, the P set keeps track of these variables. Using the S set, the variables referenced by means of a pointer dereference are partitioned into its local and global variables (Equations 5.4.a–5.4.d). The local variables are added to P, and the global variables are added to D. At an assignment statement, if the variable being defined is present in P, then the statement is included in the slice and the corresponding variables used are added to D, but the variable is not removed. Consequently, Equation 3.3 now requires two cases (Equations 5.3.a and 5.3.b), as does Equation 3.4 (Equations 5.4.a-b and 5.4.c-d).

The P set contains those local variables that have been "demoted" to have only preserving definitions. The demotion occurs only if the variable is referenced out of scope by means of a pointer dereference. Once a local variable is added to P, it is never removed. Finally, the demotion propagates through all (backward) reachable program points—P is propagated into a called function (Equation 5.1.a) and also into any calling function (Equation 5.2.a).

Discussion: To provide insight into the correctness of our equations, we can examine how the equations are transformed if pointers to local variables are not allowed. In this case, the points-to

`return from f()`

$P_{exit} = P_{exit} \cup P_i$ (5.1.a)
$D_{exit} = D_{exit} \cup (D_i \cap S)$ (5.1.b)
$N_{exit} = N_{exit} \cup N_i \cup (D_i - S)$ (5.1.c)

`call to f()`

$P_i = P_i \cup P_{entry}$ (5.2.a)
$D_i = (D_i - S) \cup (D_{entry} \cap S)$ (5.2.b)

`x := y`

if $x \in P_i$ then
$D_i = D_i \cup \{y\}$ (5.3.a)
else if $x \in D_i$ then
$D_i = D_i - \{x\} \cup \{y\}$ (5.3.b)

`x := *p`

if $x \in P_i$ then
$P_i = P_i \cup (*p - S)$ (5.4.a)
$D_i = D_i \cup (*p \cap S) \cup \{p\}$ (5.4.b)
else if $x \in D_i$ then
$P_i = P_i \cup (*p - S)$ (5.4.c)
$D_i = D_i - \{x\} \cup (*p \cap S) \cup \{p\}$ (5.4.d)

`*p := x`

if $*p \cap (D_i \cup P_i \cup N_i) \neq \phi$ then
$D_i = D_i \cup \{p, x\}$ (5.5)

N = *nonlocal local (automatic) variables of interest*
D = *variables of interest with killing definitions*
P = *local variables of interest with preserving definitions*
S = *all global (static) variables*
$*p$ = *points-to set of variable p*

Figure 5: Data-flow equations for slicing in the presence of recursion and pointers to local variables. Sets are subscripted with the program point to which they refer. Sets that are not subscripted are the same for all program points. The current statement has program point i. Unless otherwise noted, a set passes through a program point unchanged.

set of a variable p contains only global variables. Consequently, no variables are added to the P set in Equations 5.4.a and 5.4.c. Since these are the only equations in which individual variables are added to P, the P set is therefore always empty and Equations 5.1.a, 5.2.a, 5.3.a, 5.4.a, 5.4.b, and 5.4.c can be eliminated. Also, since the N set contains only local variables and the P set is always empty, both the N and P sets can be eliminated in the conditional test for Equation 5.5. Therefore, without pointers to local variables, the equations reduce to the more familiar data-flow equations for backward program slicing given in Figure 3.

Implementation: If implemented naively, our data-flow equations would require three data-flow sets per block. Since GCC has 238,000 symbols and 120,000 blocks, this would require over 10 GB, assuming bit-sets are used to represent the data-flow sets. However, since N does not change while slicing a function, a single N set can be used for all blocks of the function. Examining the P set, we see that it is nondecreasing in size, since variables are only added to the set and never removed, unlike the D set. This fact suggests that P can be made flow-insensitive with little loss in precision. Consequently, we choose to also use a single P set for all blocks of a function. This decision sacrifices precision slightly in favor of performance. Using this implementation, the space requirements are reduced to approximately 3.4 GB, still an unacceptable amount of space.

However, an analysis of the bit-sets shows that they are very sparse. Examining our equations, we see that P and N contain only local variables, while D contains local variables, global variables, and generated temporaries. Also, temporaries cannot be pointed to and therefore cannot be referenced out of scope. We therefore decide to partition the bit-sets into three distinct classes: global variables, local variables, and function-specific temporaries. The D set now consists of three bit-sets, but requires space to store only all the global variables, local variables, and the *maximum* number of temporaries per function. If we assume for simplicity that the 220,000 temporaries are evenly distributed among the 2,300 functions, the space requirements are reduced to approximately 60 MB, which is acceptable.

5 Results

To evaluate the time, space, and precision characteristics of our approach, we implemented a program slicer for C based on our ideas and measured its performance on three programs, whose basic statistics are shown in Table 1. GCC refers to the `cc1` program of the GNU C compiler, version 2.7.2 for SunOS 4.1.3; EMACS refers to the `temacs` program of the GNU Emacs editor, version 19.34b for SunOS 4.1.3 without window system support; BURLAP refers to the `burlap` program of the FElt finite element analysis system, version 3.02.[5]

	lines of code		AST		CFG	
	before CPP	after CPP[6]	time	space	time	space
GCC	217,675	224,776	24.0	55.3	42.4	51.6
EMACS	99,439	113,596	16.9	39.3	22.1	29.3
BURLAP	49,601	88,057	10.0	23.3	14.8	16.3

Table 1: Statistics for constructing representations of three programs. Time is given in seconds and space in megabytes.

Our slicer correctly handles functions with a variable number of arguments and the effects due to library functions. Library functions are handled by providing a skeleton for each function that correctly summarizes its effects. Our current implementation does not inline library functions, which is a common technique for increasing precision by adding one level of context-sensitivity. Signal handlers and the `longjmp` and `setjmp` functions are not handled.

It was our expectation that the time and space requirements would be acceptable, given our use of a near-linear time points-to analysis, our aggressive approach at implementing our data-flow equations, and piggybacking construction of the CFG with the computation of the points-to sets. We also expected that the parameterization of the points-to analysis would significantly increase the number of points-to classes, and thereby also increase the precision of the subsequent data-flow analysis. By filtering the points-to classes for function pointers based on their computed prototypes, we also expected the average number of functions called by means of a function pointer to decrease, improving the precision of the call-graph and hence the subsequent data-flow analysis. With the exception of distinguishing structure members, which is known to take worst-case exponential time and space, we anticipated that the parameterizations of the points-to analysis would have little effect on its time and space requirements. Finally, we projected that combining parameters would significantly magnify the effects of the individual parameters.

[5] All statistics in this paper were gathered on an idle 200 MHz Sun UltraSparc 2 running Solaris 2.5 with 192 MB of physical memory and 1 GB of swap space.

[6] After expansion of `include` files with CPP, all blank lines were removed.

	basic			mallocs			arrays			arrays, mallocs			structs		
	time	space	classes	time	space	classes	time	space	classes	time	space	classes	time	space	classes
GCC	51.5	62.8	267	53.6	62.8	277	51.0	63.2	472	51.3	63.6	493	117.3	89.5	543
EMACS	26.9	38.8	159	26.4	38.8	162	26.5	38.8	169	26.6	38.8	174	40.2	51.5	5638
BURLAP	15.4	22.7	207	15.3	22.7	207	15.2	22.7	279	15.2	22.7	279	21.3	29.0	1360

	arrays, structs			arrays, structs, mallocs			no collapses			arrays, no collapses			arrays, mallocs, no collapses		
	time	space	classes	time	space	classes	time	space	classes	time	space	classes	time	space	classes
GCC	116.4	90.0	746	125.0	90.1	1013	457.1	88.4	1835	192.7	89.0	1826	175.1	88.2	2028
EMACS	39.8	51.5	5648	40.4	51.5	5654	135.4	51.2	6710	97.9	51.1	6671	102.7	51.1	6624
BURLAP	21.4	29.0	1430	21.3	29.0	1430	24.8	28.4	1787	25.1	28.4	1805	25.0	28.4	1805

Table 2: Performance of the points-to analysis for various parameterizations of the analysis. The time is given in seconds and space in megabytes. Also shown is the number of points-to classes.

	basic	structs	arrays	weak prototypes	arrays w/ weak prototypes	strong prototypes	arrays w/ strong prototypes
GCC (113)	237.3	237.3	78.1	160.1	54.1	29.2	29.2
EMACS (70)	277.7	277.7	277.7	240.0	240.0	78.8	78.8
BURLAP (16)	183.8	118.2	97.8	111.1	85.5	24.9	24.9

Table 3: Average number of functions called per call site using a function pointer (an *indirect call*) for various parameterizations of the points-to analysis. The number in parentheses indicates the number of indirect call sites in each program.

5.1 Points-To Analysis Parameterization

Table 2 presents statistics for performing a points-to analysis of our three example programs. When describing the parameterizations of the points-to analysis, the following titles are used: *basic*—no parameterization, *mallocs*—private memory allocators are specified, *arrays*—array operator is not commutative, *structs*—structure members are distinguished, and *no collapses*—structure members are distinguished, but structures are never collapsed. The last parameterization is extremely optimistic, but provides a best-case upper bound on the number of points-to classes. The space measurements include all necessary data structures including symbol tables. Use of the *prototype* option is not included in this table, since the option does not change how the points-to classes are computed, but rather filters the classes once they are computed.

When comparing the various options, it is important to realize that they affect the points-to analysis in different ways. For example, the *structs* and *mallocs* options actually introduce additional locations to the points-to analysis, and with the exception of collapsing structures, do not affect how the points-to classes are merged. In contrast, the *arrays* option does not add any locations, but merely prevents unnecessary merging of the classes. However, the options are not orthogonal. Should two or more options be combined, the number of resulting additional classes may in fact be less than the sum of the number of additional classes introduced by using the options separately.

As expected, enabling the *mallocs* and *arrays* options do not have an appreciable effect on the performance of the analysis. Yet, these options do increase the number of points-to classes, making them generally beneficial options. However, the degree of effectiveness varies significantly. The *arrays* option improves the results significantly for GCC and BURLAP, but only slightly for EMACS, probably because it uses few static arrays. In contrast, the *mallocs* option improves the results slightly for both GCC and EMACS, but not at all for BURLAP, because it contains few references to private memory allocators.

Using the *structs* option increases both the number of points-to classes and the running time making it a good choice for some programs, but not for others such as GCC. For both EMACS and BURLAP, the number of points-to classes increased by an order of magnitude and the analysis only required approximately 30% more time. For GCC, use of the *structs* option did not yield a significant increase in the number of points-to classes and doubled the running time of the analysis. As we can see from Table 2, this is due to the fact that a larger number of structures were collapsed in GCC than in EMACS and BURLAP. The greater number of collapses also accounts for the increased running time.

Table 2 also shows the results for combining various parameterizations of the points-to analysis. For BURLAP, use of the *mallocs* option does not increase the number of points-to classes, and therefore combining the option with the other options is not worthwhile. For GCC, however, the *mallocs* option does increase the number of points-to classes, and combining the *mallocs* and *arrays* options shows a magnification in the number of points-to classes. By themselves, the *mallocs* option introduces an additional 10 classes and the *arrays* option introduces an additional 205 classes. When the options are combined, the result is an increase of 226 classes. The results are magnified even more when combined with additional parameters. A similar result holds true for EMACS. However, the results are not always magnified when options are combined. In GCC, for example, the combination of the *structs* and *arrays* options actually yields a decrease in the number of classes. The decrease can be explained by recalling that the options are not orthogonal to one another.

The results of the *structs* and *no collapses* options cannot be compared directly with those of the other options. When structure fields are distinguished by the points-to analysis, an extended version of the analysis is actually being performed over a *different* representation of the program. In particular, unless structure members are being distinguished by the points-to analysis and slicing algorithm, references to structure members are excluded from the CFG for performance reasons. Also, the *no collapses* option is extremely optimistic, which explains how the number of classes can actually decrease when it is combined with other options. Finally, the ordering of statements in the program can greatly impact the performance of the points-to analysis if members are distinguished. For example, if all members of a structure are first referenced, and later pointer arithmetic is performed on the structure, then the structure must be collapsed and the points-to classes for all members must be recursively merged. In contrast, if the pointer arithmetic is done first, then the structure is first collapsed, requiring little time, and all

	criteria	time	space	size of slice
GCC	(*sched.c:4964,*{*reg_n_calls_crossed*})	34.2	74.8	229,378
EMACS	(*alloc.c:1610,*{*gc_cons_threshold*})	5.3	31.2	128,434
BURLAP	(*apply.c:243,*{*result*})	1.1	8.3	40,232

Table 4: Statistics for the largest slices performed on the three example programs. Time is given in minutes and space in megabytes. The size of the slice is given as the number of three-address statements in the slice. The slicing criterion is a pair consisting of the statement, specified as a filename and line number, and a set of variables.

later references to the structure members refer to the same points-to class, since the structure is now treated as a whole. However, the *structs* and *no collapses* option do provide insight into whether distinguishing structure members has an appreciable effect on the analysis, given the more complex implementation and additional running time required.

Table 3 shows the number of functions called at each call site using a function pointer—an *indirect call*. Use of the *arrays* option reduces the average number of functions called for both GCC and BURLAP, but not for EMACS, since EMACS has few arrays of function pointers. However, the *prototypes* option works well for all three programs. Strong prototype filtering works extremely well, so much so that the results do not improve further even if the *arrays* option is enabled. Since one of the co-authors is the author of BURLAP [5], we verified its results by hand and found them to be near-perfect.

5.2 Program Slicing

To evaluate the effectiveness of our approach in data-flow analysis, we have performed several program slices of our example programs. For all slices, we tried to choose variables that might be selected by a programmer during debugging.

The results for the largest slices that we performed are presented in Table 4. The time given in the table is the time necessary to perform only the slice, not to compute the points-to information, which is given in Table 2. We believe that the time and space requirements are acceptable for a large program such as GCC. For smaller programs, such as BURLAP, the slicer performs extremely well. These results indicate that slicing large programs is feasible using our approach.

Table 5 shows the effects of user-parameterization of the points-to analysis on program slicing. The improvements due to the *arrays*, *structs*, and *mallocs* parameters are small (at most 3%), which suggests that the precision of the slicing algorithm is relatively insensitive to changes in the points-to classes [15]. This is in contrast to our original expectations, since we saw significant improvements in the number of points-to classes using these parameters.

However, the slices performed using the *prototypes* option show a dramatic improvement. The number of statements in the slice decreases several fold, as does the number of statements examined during slicing. This reduction is due to the better analysis of function pointers in both programs. Without use of the strong prototype filter, the computed call-graph is highly imprecise (i.e., containing a large number of false calls) resulting in a far greater number of statements being examined and subsequently included in the slice than is necessary. This implies that the filtering of function pointers based on their prototypes would be beneficial to many interprocedural data-flow analyses. We had expected to see similar results from use of the *arrays* option on GCC due to its dramatic reduction in the average number of calls made through a function pointer. However, this option fails to remove key functions from certain call sites, leading to false recursion between major subsystems.

	points-to options	time	size of slice	statements examined
GCC	basic	49.44	236,366	282,192
(*c-decl.c:2298,*{*b*})	arrays	39.11	230,306	282,192
	prototypes	43.66	235,037	235,037
GCC	basic	42.32	236,354	282,192
(*unroll.c:3085,*{*const0_rtx*})	mallocs	48.20	236,351	282,192
	arrays	32.41	230,305	282,192
	prototypes	0.46	9,702	13,281
	combined	0.42	9,702	13,281
BURLAP	basic	1.29	40,135	51,863
(*arith.c:145,*{*type_error*})	arrays	1.04	40,204	51,863
	structs	1.72	40,883	53,535
	prototypes	0.20	8,849	12,195
	combined	0.23	8,661	12,319
BURLAP	basic	1.29	40,135	51,863
(*matrixfunc.c:767,*{*status*})	arrays	1.07	40,204	51,863
	structs	1.62	40,858	53,535
	prototypes	0.22	9,332	12,764
	combined	0.25	9,144	12,890

Table 5: Statistics for various slices of the example programs with different parameterizations (*basic*, *arrays*, *structs*, and strong prototypes) of the points-to analysis.[8] For GCC, the *arrays* and *prototypes* options were combined; for BURLAP, the *structs* and *prototypes* were combined. Time is given in minutes.

We were unable to find a parameterization of the points-to analysis that produced any substantial improvement for EMACS. We were disappointed that the use of *prototypes* option did not yield an improvement as it had for GCC and BURLAP. After examining the source code for EMACS, we believe that the points-to sets for function pointers are fairly precise. Rather, it is the nature of EMACS that is the source of the problem. In particular, it appears that the subsystems of the EMACS interpreter are in fact recursive due to the implementation of dynamically scoped error handling.

Table 5 also shows the effects of combining pairs of parameters. Although benefits are typically seen from the combination, the improvements tend to be small (at most 2%). The more dramatic results of combining more than two parameters on the points-to analysis suggests that we should see better but not dramatic results in slicing.

6 Conclusion and Future Work

Performing effective whole-program analysis is difficult in the presence of pointers. Points-to analysis requires a global analysis over the program, making it difficult to integrate with demand-driven techniques, which are a necessity when analyzing large systems. The use of pointers in large programs, specifically function pointers and pointers to local variables, complicates performing an effective data-flow analysis.

To overcome these problems, we presented a solution for integrating points-to analysis with demand-driven analyses, along with techniques for parameterizing the analysis to achieve better points-to results. We also presented data-flow equations for computing an interprocedural backward program slice in the presence of both recursion and pointers to local variables, and described an efficient implementation of our equations. Each aspect of our approach contributes to its effectiveness.

- Piggybacking the construction of the CFG with the computation of the points-to sets eliminates an extra pass over the program, saving time.

[8] Some of the additional three-address statements generated for structure member accesses have been removed for comparison with the other parameterizations. However, not all additionally generated statements could be easily removed.

- Persistently retaining the call-graph on disk allows only the reachable portions of the CFG to be retained, saving space and also time by avoiding the use of virtual memory.

- Parameterization of the points-to analysis increases the effectiveness of the subsequent data-flow analysis. In the absence of function pointers, the increase in the number of points-to sets does not result in a substantial increase in precision, due to the transitive effects of the data-flow analysis. However, in the presence of function pointers, the computed call-graph is substantially more precise, which greatly increases the precision of the data-flow analysis with respect to function calls and realizable paths.

- Through an aggressive implementation of the data-flow sets, significant space can be saved, making whole-program analysis practical in the presence of pointers to local variables in recursive programs.

To validate our techniques, we constructed a program slicer for C programs and performed several slices of three large programs. By parameterizing the points-to analysis, the number of points-to classes could be increased with little performance cost. Also, the number of functions called through a function pointer could be substantially decreased. In particular, filtering the points-to classes based on their prototypes greatly reduces the number of calls. For program slicing, the improved points-to information for functions resulted in more accurate and faster program slices, due to the increased precision of the computed call-graph, which suggests that our techniques would be applicable to many interprocedural data-flow analyses. For the largest slices that we performed, the time and space requirements were acceptable for today's desktop computers, indicating that practical whole-program analysis of large C programs is feasible using our techniques.

Since prototype filtering works so well on improving the precision of the computed call-graph, one might wish to extend the technique to ordinary program variables. By using the type system of the language, similar filters can be constructed for variables. For example, if the points-to set for a variable p points to both an integer variable n and a real variable x, but p is declared to be a pointer to integer, then x can be removed from the points-to set. Since distinguishing structure members in the points-to analysis has poor time and space characteristics for some larger programs, filtering based on structure member types could be an inexpensive alternative. For languages like C, however, such filtering is almost certainly unsafe in the general case because the programmer can violate the type system. For languages with strong typing, on the other hand, type violations are not supported by the language, so filtering can be safely applied. We have found that filtering based on function prototypes is worthwhile given its benefits and the infrequent violation of the type system for function pointers in C programs (i.e., casting function pointers is rare in C).

One question that remains is whether our techniques can be applied to other programming languages. Since points-to analysis requires a global analysis over the system for any programming language, our approach to integrating the analysis with demand-driven analyses applies generally. Although parameterizations of the points-to analysis such as array commutativity are specific to the C language, the general ideas can be applied to any programming language that is flexible and performance-oriented—any sufficiently powerful language will have pointer constructs whose use complicates effective points-to and data-flow analysis. For example, although private memory allocators may not be used for performance reasons, they are often used for reasons of encapsulation and code reuse. The constructor and destructor functions in C++ [19] are obvious examples. Distinguishing structure members and filtering points-to classes based on their computed prototypes are also necessary in C++, since its classes are little more than structures composed of many function pointers and some data. Finally, an aggressive implementation of the data-flow sets such as ours is necessary to achieve good performance when analyzing large programs, even if the generalization for pointers to locals is not required.

Acknowledgements: Thanks to Jeanne Ferrante for her help on the data-flow equations. Thanks also to Robert Bowdidge, Van Nguyen, and Nicholas Mitchell for their help on improving the organization of this paper. Finally, we would like to thank Daniel Weise and the other members of the program committee for their advice on improving the focus of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, Berlin, Germany, March 1996.

[3] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, CA, January 1995.

[4] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM '94 SIGPLAN Conference on Programming Language Design and Implementation*, pages 20–24, Orlando, FL, June 1994.

[5] J. I. Gobat and D. C. Atkinson. The FElt system: User's guide and reference manual. Computer Science Technical Report CS94-376, University of California, San Diego, Department of Computer Science & Engineering, 1994.

[6] W. G. Griswold and D. C. Atkinson. Managing the design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1–2):99–116, July–August 1995.

[7] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 104–115, Washington, DC, October 1995.

[8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[9] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction*, pages 125–140, Paderborn, Germany, October 1992.

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

[11] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press, Orlando, FL, 1985.

[12] G. N. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering*, pages 93–105, San Francisco, CA, November 1996.

[13] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

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

[15] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, pages 16–34, Paris, France, January 1997.

[16] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.

[17] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 136–150, Linköping, Sweden, April 1996.

[18] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 1996.

[19] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1991.

[20] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.