Integrating Influence Mechanisms into Impact Analysis for Increased Precision

Ben Breech
Dept. of Computer Science
University of Delaware
Newark, DE 19716
breech@cis.udel.edu

Mike Tegtmeyer
Army Research Lab
AMSRL-SL-BE, Bldg. 238
Aberdeen Proving Ground,
Maryland 21005-5068
mtegtmeyer@arl.army.mil

Lori Pollock
Dept. of Computer Science
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Abstract

Software change impact analysis is the process of determining the potential effects, or impacts, of a change to a program. Strategies for impact analysis vary in their approach toward the opposing goals of high precision and low analysis time. Fine-grained techniques, such as slicing, can be used to gain very precise knowledge of a change's impact, but may be prohibitively expensive. Coarse-grained techniques such as method-level impact analyses sacrifice precision for faster analysis.

In this paper, we present static and dynamic method-level impact analysis algorithms that utilize value propagation information from the source code to increase precision and keep analysis times low. We experimentally compare the results of our analyses with common static and dynamic impact analysis techniques. Our results show that the precision of the common method-level analyses can be improved with very little added overhead.

1. Introduction

Software change impact analysis seeks to identify the potential effects (i.e., impacts) that a change to a function has on other functions in the program [2]. Change impact information is important for software maintenance as the analysis, which can be applied either before or after changes are made to the software system, can provide a valuable guide to the software engineer. Applying impact analysis before changes are made allows a software engineer to determine the components that may be affected by the changes, gauge the cost of specific changes, and choose among proposed changes using the estimates of their potential costs. After a change is made, impact analysis can be used to guide regression test efforts by reducing the set of test cases to be run to those cases that traverse the change.

Current techniques for impact analysis are either *static*, *dynamic* or *online*. Static analyses (such as [3, 10, 13]) perform either forward slicing or graph traversals to gather data about potential impacts (i.e., compute impact sets). These techniques are conservative, taking into account all possible program inputs and behaviors. However, this may produce static analysis results that are too large to be of practical use. For example, infrequently executed functions, such as error routines, may appear to be impacted by a change, causing a software engineer to spend unnecessary time checking or retesting such functions.

In contrast, dynamic [1, 6, 8] and online techniques [4, 5] seek to compute impact sets that are closer to how the application is actually being used and to avoid including computed impacts resulting from impossible program behavior. These analyses compute impacts using information derived from a set of executions of the program. Since the results are dependent on the inputs used to execute the program, impacts will be missed for functions that are not executed, making the dynamic and online analysis results unsound. However, conservative dynamic and online analyses will not miss impacts among executed functions. The impact sets generated by the dynamic and online analyses tend to be smaller than those produced by the static techniques and to better represent the actual use of the application.

Tradeoffs exist between precision and analysis time among different impact analysis techniques. Interprocedural slicing [12], which we call a fine-grained technique, can be used to obtain very precise results, but can be expensive. Other more coarse-grained techniques which operate at the method level, such as graph closure operations or some of the dynamic analysis algorithms [3, 9], sacrifice precision in exchange for faster analysis by computing impacts based solely on control flow. That is, a change to function f is said to have an impact on function g only if control flows from f to g (directly or indirectly) through a sequence of calls and returns. These approaches ignore important information concerning scoping, function parameter passing and returns that actually enable a change in f to propagate to g.

In this paper, we present coarse-grained impact analysis

algorithms that exploit information about how changes can actually propagate due to scoping and parameter passing mechanisms, with the goal of computing more precise impact sets than current coarse-grained impact analyses, without the expense of fine-grained analyses. In particular, we use static analysis to conservatively estimate the propagation of a change by taking into account scoping, function signatures, and global variable accesses. We refer to the ways in which changes can propagate as *influence mechanisms*. We have developed, implemented, and experimentally evaluated both static and dynamic coarse-grained analysis algorithms that use these influence mechanisms. Our experimental study shows that these algorithms can indeed improve precision compared to the current coarse-grained analyses while maintaining low analysis time.

The rest of this paper is organized as follows; section 2 describes the most well known impact analysis techniques in the current state of the art. Section 3 introduces the influence mechanisms and describes how they can influence change impact propagation. Sections 4 and 5 describe static and dynamic impact analysis algorithms, respectively, that incorporate influences into the analysis. In section 6, we present our experimental evaluation of our approach. Section 7 discusses conclusions and future work.

2. State of the Art

Static Impact Analysis. Static impact analysis techniques usually focus on reachability of graphs [3]. At the method level, a call graph can be used, but it may miss some impacts [6]. A more precise static technique based on statement-level information uses the interprocedural control flow graph (icfg)[1] [7]. For example, a closure on the icfg can produce more accurate results than the call graph since return edges can be followed back to the call site in the function body.

The advantages of static techniques are their conservative nature and their efficiency. Closures on the icfg take into account all possible executions of the program, enabling a conservative estimate of impacts. Furthermore, the closure operation tends to be quick in practice, although some pathological cases could make the run time be as bad as $O(S^2N)$, S being the number of statements and N the number of functions in the application. Unfortunately, the conservatism can yield results that identify a large percentage of the program as potentially being impacted.

For the program given in Figure 1, the impact of a change to function `C` is computed using the icfg by starting at the entry of `C` and finding all callsites in `C`. This step immediately adds `C, F, D` and `E` to the impact set for `C`. The closure algorithm would descend into each of those functions to find others that may be impacted, adding no new functions for this particular case. When finished processing `C`, the closure follows edges back to the callers of `C` and repeats the process. This step adds `A` and `B` into the impact set. Following the return edges of `A` and `B` adds `Main` to the set. The second column of Table 1 shows the impact sets calculated by closure on the icfg for all the functions.

Closure on the icfg can be used as an approximation [8] to slicing with less expense [12] because the closure operation only uses control flow information and in a conservative way. For example, `C` impacts `Main` simply because there is a chain of calls and returns from `C` back to `Main`. We will use closure on the icfg as our base static analysis for comparing our work.

Dynamic Impact Analysis. To our knowledge, `PathImpact` [6] is currently believed to be the most precise dynamic method-level impact analysis. We will use the `PathImpact` approach as a comparison for our dynamic algorithms. Other algorithms exist [5, 1] that compute the same results as `PathImpact` but with less overhead, as well as fine-grained dynamic techniques such as dynamic slicing [12].

`PathImpact` computes impacts by examining the function call trace of a program. `PathImpact` uses the heuristic that the impact set for a procedure p is "any procedure that is called after p, and any procedure which is on the call stack after p returns" [6]. `PathImpact` starts at the first call to p and moves forward through the trace, identifying functions called anytime after p (including after p returns), until the end-of-trace symbol is found. `PathImpact` then moves backward to identify functions that p could impact through its return.

As an example, we use `PathImpact` to compute the impact set of function `H` shown in Figure 1. To calculate the impact set for `H`, `PathImpact` first adds `H` and then scans forward for the end-of-trace symbol, `x`. Functions `A, C, F, D` and `E` are added. `PathImpact` then scans backward to find functions that `H` could possibly return into. This scan adds `G` and `Main` into the set giving a final impact set for `H` as {`Main A C D E F G H`}. Note that if no data dependencies exist between `H` and the later functions, `A, C, F, D` and `E`, then extra impacts are reported where none exist. `PathImpact`, however, will not miss impacts among executed functions. Applying `PathImpact` to `E`, we obtain {`Main A C D E`}. `F, G` and `H` are not added because `E` cannot return into them. The fourth column of Table 1 shows the impact sets calculated by `PathImpact` for all the functions called in the example trace.

3. Influence Mechanisms

Our work focuses on improving the precision of the less expensive coarse-grained techniques by supplementing the

[1] The cfg is a directed graph with nodes that represent source code lines and edges that represent how control passes from one statement to another. An icfg connects the individual cfg's of each function to represent interprocedural control flow through calls and returns.

```
global_type global_var;

int main (int argc,
          char *argv []) {
   global_var = G ();
   if (some_condition){
       B ();
   }
   A ();
}

global_var_type G (void) {
   global_var_type g;
   H ();
   //initialize g
   return g;
}
```

```
void H (void) {
   // no calls
   // access to global_var
}

void F (global_type g) {
  global_type g1
  // setup g1
  if (some_condition_with_g) {
    C (g1);
  }
  else {
     // other stuff
  }
}
```

```
int C (global_type &parm) {
  global_type g, g1;
  // some calc. with
  // g and parm
  if (some_condition) {
    F(parm);
    g1 = D ();
    E (g, g1);
    D ();
  }
  return 0;
}

void E (global_type &g,
        global_type &g1) {
  // ...
}
```

```
void A (void) {
  // ...
  C (global_var);
}

global_type *D(void)
  global_type *g;
  // initialize g
  return *g;
}

void B (void) {
  global_type g;
  // ...
  C (g);
}
```

Trace: `Main G H r r A C F C r r D r E r D r r r x`

Figure 1. Example Program

	Static Algorithms		Dynamic Algorithms	
Function	ICFG Results	Combined Graph results	`PathImpact` Results	Using Influence Mechanisms
Main	{Main A B C D E F G H}	{Main A B C E F H}	{Main A C D E F G H}	{Main A C E F H}
A	{Main A C D E F}	{Main A C E F}	{Main A C D E F}	{Main A C E F}
B	{Main A B C D E F}	{Main A B C E F}	{}	{}
C	{Main A B C D E F}	{Main A B C E F}	{Main A C D E F}	{Main A C E F}
D	{Main A B C D E}	{Main A B C D E}	{Main A C D E}	{Main A C D E}
E	{Main A B C D E}	{Main A B C E}	{Main A C D E}	{Main A C E}
F	{Main A B C D E F}	{Main A B C E F}	{Main A C D E F}	{Main A C E F}
G	{Main A B C D E F G H}	{Main A B C E F G H}	{Main A C D E F G H}	{Main A C E F G}
H	{Main A B C D E F G H}	{Main A B C E F H}	{Main A C D E F G H}	{Main A C E F H}

Table 1. Impact Sets for the Example Program

impact analysis with knowledge of how impacts can propagate among functions. The key observation is that a change to f can propagate to g through control flow, but only if the change can propagate through the parameters, return values and global variables between f and g. Specifically, we say that f "may influence" the change propagation from f to g if certain "value propagation conditions" hold. In this section, we describe the "influence mechanisms" and the kinds of influences they can have on change propagation.

No-Influence Functions. A function has no influence if it does not access any global variables, has no parameters and returns no values. Function f having no influence means that callers of f can not propagate changes or impacts of changes to f. Changes made to f will not propagate back to f's callers. However, f can be changed and pass its changes to functions that f calls. Disregarding the global variable in Figure 1, functions `A` and `H` would have no influence.

Value_in. A function f can be influenced by its formal pass-by-value parameters. Changes to f's caller or impacts propagated to f's caller, could propagate into f through the pass-by-value parameters. `F` is an example of the value_in influence mechanism. A change to `C` could potentially affect `F` by changing the parameter passed to `F`.

Value_out. Function f can influence its callers through a return value. Changes made to f or propagated to f could potentially be propagated back to f's caller through the return value, and then onwards to other functions called later. In our example, because function `G` returns a value, a change to `G` could propagate to `Main`, and then onto other functions called later. Note that the propagated change only has an observed impact if the return value is used by the caller. To avoid expensive interprocedural data flow analysis, we will make the conservative assumption that there may be a later use of the returned value. Thus, we may include impacts in impact sets when a propagated change has no real observed impact.

Ref_in. Passing parameters by reference (or address) acts similar to passing by value in that changes to the caller can propagate into the callee. However, the callee can also modify the parameters, thereby potentially propagating changes back to the caller. Without global variables, this case acts as if the callee has value_in and return_val influences. Reference parameters passed as constant cannot be changed by the callee; changes cannot propagate back to the caller. In this case, and without the effect of globals, reference parameters are similar to value parameters and can be treated as such by the impact analysis.

Ref_out. A function returning an object by reference (or via

a pointer), constant or otherwise, is no different from the return value case, when global variables are not considered.

Handling Aggregate Data Types as Parameters. Aggregate data types, such as `struct` or `union`, complicate the analysis as their member fields can include pointers and reference variables. These aggregate types can allow impacts to propagate out of a given function even if the aggregate variable itself was passed by value or declared "constant." We adopt a conservative approach for estimating the influences through aggregate data types. The member fields of an aggregate data type passed to, or returned from, a function are examined. If any member fields are pointers and reference variables, the function is considered to have ref_in or ref_out influence as appropriate.

Influence through global variables. Global variables also complicate the analysis of change propagation. Functions f and g can implicitly impact each other through global variables regardless of the influence mechanisms of any function called between the execution of f and g. The analysis must keep track of the functions that access a particular global variable. For our purposes, an access to a global variable is either a definition or a use of that variable. This choice is conservative, and introduces some imprecision.

Global variables present an additional complication when combined with function parameters and returns. Without computing expensive statement-level information, we must make the assumption that a function f may call another function g, possibly using a global variable as an actual argument which effectively "masks" the global variable inside g. For example, using the heuristics presented without considering the effect of globals, a change to function `H` would not propagate to `C` since `H` returns no values nor has parameters. However, since `H` accesses `global_var`, the change in `H` could have an impact, namely a change to `H` could affect `global_var`. When function `A` calls `C`, passing `global_var` as an argument, the change to `H` could affect `C`.

Reference parameters and returns further complicate handling of global variables. If f has reference parameters and is passed global variable, gv, f could now modify gv. f can now impact other functions called later that also access gv.

We take the following conservative approach to estimating the influence due global variables. A static preprocessing analysis pass records the globals that are accessed in each function. Without statement-level information, we must assume that the global variables could be passed as actual parameters to a function. During impact analysis, if a function is called that has reference parameters, the global variables that are accessed by the caller are added to those accessed by the callee. For return reference arguments, a similar action occurs, with the global variables of the callee added to those of the caller. Impacts due to global variables then can be accounted for by examining the set of globals accessed by a particular function.

The results of impact analysis using these influence mechanisms on change propagation are shown for both static and dynamic analysis in Table 1. For both the static and dynamic analyses, the analysis using influence mechanisms was able to remove `D` and `G` from the impact set of `Main`. These two functions take no parameters and do not access global variables, so no change to `Main` could impact them. Additionally, `D` was removed from a number of other sets.

4. Static Impact Analysis with Influences

We have defined a novel program model that captures the predicted influences due to the influence mechanisms discussed in Section 3. The influence model is defined to be a graph, called the *influence graph*. Both static and dynamic impact analysis can be performed using the same influence graph, which is constructed through a single preprocessing pass through the source code.

4.1. The Influence Graph

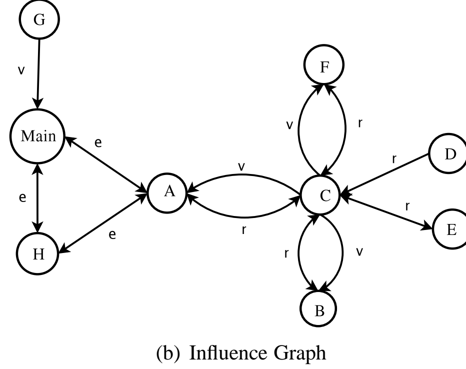
The influence graph for a program *P* consists of a node for each function in *P*. Two nodes *i* and *j* can be connected by a set of directed edges, each edge representing influences between the functions represented by *i* and *j*. There are three different kinds of edges - *value*, *reference* and *external*. The direction of an edge indicates how changes can propagate between the functions represented by the connected nodes. Each of the influence mechanisms corresponds to a particular kind of edge and direction; value edges represent the value_in and value_out mechanisms, reference edges represent ref_in and ref_out mechanisms, and external edges account for influences due to global variable accesses.[2] Value and reference edges are added between callers and callees. For example, if f calls g and f influences g through the value_in (ref_in) mechanism, a value (reference) edge would be added from f to g. If g influences f through value_out (ref_out) then a value (reference) edge would be added from g to f. External edges are added between any two nodes that access the same global variable. External edges are bidirectional in our current approach; however, they could be constructed to be unidirectional and more precise, if more expensive data flow analysis were performed.

Figure 3(a) shows the influence mechanisms of each function in the example program along with the set of functions explicitly accessing `global_var`. Due to space limitations, the set of callsites in the example program is not shown, but can be easily derived from the code shown in

[2] Here, we focus on global variables. However, 'external' is more general as it could account for variables in different namespaces.

Main	value_in, ref_in, value_out, ref_out
A	none
B	none
C	ref_in, value_out, ref_out
D	ref_out
E	ref_in
F	value_in
G	value_out
H	none
global_var	Main, A, H

(a) Influence Mechanisms



(b) Influence Graph

Figure 3. Influences and Influence Graph Example

Algorithm: BuildInfluenceGraph
Input:Program P
Output: Influence graph

```
FOR each callsite in P, DO
    IF callee has value_in influence, THEN
        Add value edge from caller to callee
    IF callee has ref_in influence, THEN
        Add reference edge from caller to callee
    IF callee has return_val influence, THEN
        Add value edge from callee to caller
    IF callee has ref_out influence, THEN
        Add reference edge from callee to caller
FOR each global variable, g, DO
    FOR each pair of functions (n,m) that access g
        Add birectional external edge between n and m
```

Figure 2. Influence Graph Construction

Figure 1. The influence graph for the example program is shown in Figure 3(b).

We note that the influence graph contains no explicit information about function calling relationships. A reference or value edge from p to q infers that there is some calling relationship between p and q, but there is no way of knowing which is the caller and which is the callee. External edges provide no calling information. The influence graph does not explicitly represent control flow, but rather how changes propagate between functions.

Figure 2 presents the algorithm to construct an influence graph for a program. The influence mechanisms can be easily determined by examining the signatures of each function (i.e., the formal argument list and return types). The set of functions that access each global variable (either defining or using) can be computed by scanning the source code.

Following the BuildInfluenceGraph for our example, as we process the callsite set, we find that B calls C. Since C has a ref_in influence, the algorithm adds a reference edge from B to C. The value_out and ref_out influences cause a value edge and a reference edge to be added from C back to B, respectively. In processing for global variable influences, the algorithm determines that Main, A and H all access the same global variable. An external edge is therefore added between each pair of those functions.

As written, the time needed to construct the influence graph is on the order of the number of callsites in the program. However, collecting the information about the callsites and influence mechanisms requires $O(S)$ time, where S is the number of program statements. The influence graph can be constructed at the same time that this information is collected. The space needed for the graph is, at worst, $O(N^2)$, where N is the number of functions in the program. This corresponds to the case where every function is connected to all other functions.

The construction of the influence graph can be enhanced to provide more precise approximation of the predicted influences on change propagation. In the example, the access of the global variable performed in H is overwritten by the definition in Main. A change to H would not propagate to other functions through the global variable. More static data flow analysis could detect such occurrences, which will further improve the precision of the impact results. For our current work, we examine the precision that can be gained without performing data flow analysis, in order to demonstrate the kinds of improvements possible with little investment in static analysis of influences.

4.2. Integrated Static Analysis Algorithm

Once the influence graph has been constructed, an estimate of the impact of function p can be derived by performing a simple closure on the graph starting from p. For static impact analysis, there are no differences between the edges during the closure operation.

For the example program, we can compute how a change to C propagates by starting with C in the influence graph. Following all edges from C adds A, B, E and F to the impact set of C. We then go to A, B, E and F and find all functions reachable from them. This step adds Main and

H to the impact set. The impact set of C is reported to be {Main A B C E F H}.

When we compare this result to the impact sets computed by performing closure on the icfg, we note that function D was not added to the impact set of C because there is no way for changes to C to propagate into D. This is a gain in precision by using the influence graph instead of the icfg. However, H was added, indicating a potential loss of precision by using the influence graph. The potential loss occurs because the influence graph has no access to statement-level information nor explicit information about calling relationships, both of which are available in the icfg. More sophisticated use of data flow information could eliminate this potential precision loss, leaving the influence graph-based closure to provide strictly more precise results.

An alternative to performing and using results of data flow analysis in conjunction with the influence graph is to use the icfg and the influence graph in combination. The CombinedStatic algorithm (**not shown for space reasons**) first estimates the impact set by performing a closure on the influence graph. Then, each impact set element that is not reachable from the changed function in the icfg is removed. For our example, this combination of the influence graph and the icfg produces the sets given in Table 1.

The run time of the CombinedStatic algorithm can be upwards of $O(N^3 + S^2N)$ to perform the closure and reachability operations, where N is the number of functions and S is the number of statements in the program. In practice, the algorithm is much faster as the worst case assumes all functions are connected to each other in the graphs.

5. Dynamic Impact Analysis with Influences

In this section, we discuss two, almost equivalent, approaches to performing dynamic impact analysis using influences. The first approach performs a more accurate analysis, but can be very resource intensive in time and space. The second algorithm approximates the first by using the influence graph.

The high level view of both algorithms is that the impact of a change in function C is computed by examining the calls and returns made by the program during execution. For all function calls and returns in the function execution trace, both algorithms must perform book keeping operations related to global variables. Both algorithms look for the first call to C in the execution trace. All subsequent calls and returns are checked to determine if the change can propagate. In both algorithms, C is always added to the impact set for C to satisfy the trivial case.

5.1. Globals Tracking-based Algorithm

The InfluencedDynamic algorithm performs dynamic impact analysis using influence information to compute the impact of a change to a single function. We provide a high level overview of the algorithm without explicitly showing it due to space constraints and the similarity of InfluencedDynamic to the algorithm using the influence graph (see section 5.2 below).

InfluencedDynamic maintains sets for each function, f, listing the influences that f has. Each function also has a set of global variables associated with it. This set lists all the global variables that f directly accesses. During the analysis, InfluencedDynamic propagates impacts of changes using the influences of the caller and callee. Global variables may also be propagated when the callee has reference influence. Each global variable is tracked separately.

5.2. Influence Graph-based Algorithm

A drawback of InfluencedDynamic is that maintaining the sets (particularly the sets for global variables) for each function can be expensive. We can use the influence graph to provide a faster implementation at the expense of some precision. The influence graph handles global variables by placing an external edge between two nodes if those nodes access the same global variable. Only one edge is used regardless of the number of global variables actually accessed.

Figure 4 presents ApproxInfluencedDynamic which performs dynamic impact analysis using the influence graph. Global variables present a problem when combining dynamic information with the influence graph. Statically, impacts due to global variables will be eventually found by following external edges. Dynamically, however, we only have access to the current call stack, particularly the caller and callee. Function f could modify a global variable during its execution and then return, thus being removed from the call stack. Functions executed later will have no direct connection to f and no knowledge of the impacts due to the global variables. The algorithm in Figure 4 handles this issue by updating the ExtSet of each function connected to f by an external edge when f executes. During impact analysis, the ExtSet for each executed function is examined. If any member of the set has been impacted, then the called function must also be impacted. The ExtSet for a function f is initialized to f before processing begins, and never reinitialized, only augmented. The impacted flag indicates whether the caller has been impacted, and is used to determine whether impacts can be propagated to the callee.

The time for ApproxInfluencedDynamic is $O(FN \ln N)$, where F is the number of calls returns analyzed and N is the number of functions in the program. The space requirement is $O(N^2 + CN)$, C being the number of elements on the call stack, N^2 is the maximum size of the influence graph. The reasoning is the same as for InfluencedDynamic, except that the ExtSet can

have, at most, N elements. As before, the CN term can be ignored.

Example. Again we compute the impact set for `H`, `I(H)`, as an example. The program begins by executing `Main`. Since `Main` is connected to `A` and `H` by external edges, the algorithm adds `Main` to the `ExtSet` for `A` and `H` (lines 16, 17). `Main` then calls `G`, which in turn calls `H`. Until this point, the `impacted` flag has been set to false. As we are interested in the impact of `H`, the flag is now set to true and `H` is added to `I(H)` (lines 13 - 15). `H` is also added to the `ExtSet` for `A` and `Main` (lines 16, 17). `H` finishes, but because there is no edge from `H` to `G` in the influence graph, `G` is correctly not added to the impact set. The `impacted` flag is set back to false (line 29) indicating that `G` is not impacted so the change could not propagate through `G`. `G` returns and now `Main` calls `A`. Since the `ExtSet(A)` contains `H` and `H` is in `I(H)`, `A` could potentially be affected. `A` is added to `I(H)` and `impacted` is set to true (lines 12 - 15). `C` is now called. There is a reference edge from `A` to `C`, so `C` must now be added to `I(H)` (lines 6, 7). Due to the reference edge, any global variable that `A` has access to could also be modified by `C`. Therefore, the `ExtSet(C)` is unioned with `A` (lines 10, 11). The algorithm processes the rest of the trace, which will add `E` and `F` to `I(H)`. Finally, when `Main` returns, the algorithm detects that the external set of `Main` contains impacted functions, so `Main` can also be impacted. `Main` is added to `I(H)`. The algorithm computes `I(H)` as {`Main A C E F H`}.

For comparison, `InfluencedDynamic` would also compute the same impact set for `H`. In general, we do not expect the two algorithms to give the same results due to the different handling of global variables. One advantage of using the influence graph, however, is that other static analysis can be used to eliminate edges in the graph, which provides more precision in the analysis. For example, if the caller of a function, f, does not use the return value, then there is no reason to have an edge from f back to the caller.

Algorithm: `ApproxInfluencedDynamic`
Input: Trace of Function calls and returns of program P
Influence graph of program P
Output: Impact set I of changed function C

1. Begin processing trace of executing P
2. `impacted` = false
3. While not program exit, DO
4. When function `callee` is called, DO
5. Let `caller` be the function on the top of the call stack
6. IF `impacted` AND a value or reference edge exists from `caller` to `callee`, THEN
7. $I = I \cup$ `callee`
8. ELSE
9. `impacted` = false
10. IF reference edge from `caller` to `callee`, THEN
11. `ExtSet(callee)` = `ExtSet(callee)` ∪ `ExtSet(caller)`
12. IF NOT `impacted`, THEN
13. IF(`callee` = C) OR (`ExtSet(callee)` ∩ $I \neq \emptyset$) THEN
14. `impacted` = true
15. $I = I \cup$ `callee`
16. FOR EACH function e such that there is an external edge from `callee` to e, DO
17. `ExtSet(`e`)` = `ExtSet(`e`)` ∪ `ExtSet(callee)`
18. Push `callee`, `impacted` onto the call stack

19. When a function `callee` returns, DO
20. Pop `callee`, `impacted` from call stack
21. Let `caller` be the function on top of the call stack
22. IF (NOT `impacted`) AND (`ExtSet(callee)` ∩ $I \neq \emptyset$) THEN
23. `impacted` = true
24. $I = I \cup$ `callee`
25. IF `impacted` AND a value or reference edge exists from `callee` to `caller`, THEN
26. $I = I \cup$ `caller`
27. IF reference edge from `callee` to `caller`, THEN
28. `ExtSet(caller)` = `ExtSet(caller)` ∪ `ExtSet(callee)`
29. `impacted` = `impacted` flag from top of stack
30. End processing trace

Figure 4. Dynamic IA using Influence Graph

6. Experimental Study

We performed an experimental study to evaluate the effectiveness of using influence mechanisms to calculate more precise impact sets. We chose several C programs from the SPEC [11] application suite as subjects of analysis. These applications are small to medium-sized real world applications. We have also included the `space` program from the European Space Agency. The applications, their descriptions, the number of uncommented, non-blank lines of code and the number of functions in the program are listed in Table 2.

Program	Description	Source LOC	funcs
008.espresso	Boolean function minimizer	9,844	363
099.go	Plays the game of go	25,080	374
130.li	Lisp interpreter	4,888	366
132.ijpeg	JPEG compressor	15,925	476
147.vortex	Object Oriented database	40,242	925
164.gzip	Gzip compression	5,604	106
300.twolf	Transistor placement	17,819	191
space	ESA ADL Interpreter	6,230	136

Table 2. Subjects of Analysis

6.1. Research Questions

The research questions we sought to answer are:

1. *Do the influence mechanisms result in more precise impact sets than performing transitive closure on an icfg (static) or using* `PathImpact` *(dynamic)?*
2. *Is the cost to build the influence graph reasonable?*
3. *Is the cost to perform the static and dynamic analyses with the influence mechanisms reasonable?*

The algorithms discussed are safe in terms of not missing impacts. All of the assumptions made in building the algorithms were conservative in nature. We thus use the metrics from Orso et al. [9] and measure precision in terms of the number of methods in the impact sets. We consider algorithm A to be more precise than algorithm B if the results of A are, overall, a subset of the results from B. That is, if we use A and B to compute the impact set of a function f, then, since A and B do not miss any impacts, we consider A to be more precise if the impact set of f has fewer members when using A.

6.2. Setup

We implemented and ran the static and dynamic algorithms on a Linux machine with 2.0 GB of RAM and an Intel Pentium 4, 3.8 GHz processor. GCC 4.0 was used to compile the applications. We modified GCC to determine the influences and callsites during compilation. The influence graph was constructed using the algorithm in Figure 2.

Implementation. For static impact analysis, we constructed three different analyses – graph closure on the icfg generated by GCC, closure on the influence graph, and `CombinedStatic`.

For dynamic analysis, we used GCC to instrument the programs to generate a call trace. The applications were executed with randomly selected test cases. `PathImpact` results were generated with an implementation used previously [5]. We implemented both algorithms from Section 5.

Many of the applications used in our study utilize function pointers, which must be disambiguated to obtain accurate results. Dynamically, all function pointers are automatically disambiguated since the address of each executed function is known at run time. Statically, another approach must be taken. While statically scanning the code, we note all functions whose address was used to initialize a function pointer. We then treat all call sites using function pointers as calls to those functions. This approach is not truly conservative due to pointer arithmetic, but is accurate enough for the applications we use.

Expected Results. On the basis of the examples in Section 4, we do not expect either closure of the influence graph or closure of the icfg to necessarily be more precise than the other. However, we expect the combination of the two to be more precise than using either alone.

We expect both dynamic algorithms to be more precise than `PathImpact`. We also expect the `InfluencedDynamic` to be more precise than `ApproxInfluencedDynamic`, although it may be more expensive in terms of time.

Threats to Validity. We have chosen C applications that vary in size from 1,000 to 25,000 lines of code. The results of our study may not necessarily generalize to other programs. However, the applications in our study are not toy programs and they span a variety of real world applications. The test cases reflect how the programs are actually used. Nonetheless, more applications and associated test cases would provide a better basis for generalizations.

6.3. Results and Analysis

Table 3 shows the precision results for the static and dynamic analysis algorithms. The average impact set size (`avg. sz`) across all functions in the program is shown for each algorithm. The average difference (`avg. diff`) shows how many functions were, on average, removed from each impact set. For the static case, the difference is how many functions the `CombinedStatic` algorithm was able to remove from the sets computed by the icfg closure. For the dynamic case, the difference is how many functions are removed from the `PathImpact` results by our algorithms. The standard deviation is also provided. The average set size columns and the average difference columns provide a measure of the precision gained. The maximum difference column (`max`) gives the maximum number of functions removed from any of the impact sets for the program.

In response to our research questions, we do find that taking account of the various influence mechanisms yields a gain in precision (question 1). From Table 3 we note that estimating impacts by performing closure on the influence graph tends to find extra impacts than when performing closure on the icfg. This is to be expected because of the lack of control flow information in the influence graph. However, if no icfg is available, the influence graph can be used to give good estimates. The best static results were obtained from `CombinedStatic`. The gain in precision is under 3%, but there can be considerable gain, as seen from the maximum difference column. Similar results are seen in the dynamic case as well, where both of our dynamic algorithms provide gains over `PathImpact`. Again, as seen in the maximum difference columns, the gain in some cases is considerable. We also note that the dynamic algorithms provided nearly 20% gain in precision for the space application.

	Static					Dynamic						
appl.	icfg	infl. graph	CombinedStatic			PI	InfluencedDynamic			ApproxInfluencedDynamic		
	avg. sz	avg. sz	avg. sz	avg. diff	max	avg sz	avg. sz	avg. diff	max	avg. sz	avg. diff	max
008.espresso	308	318	297	10.8 ± 24	331	149	143	5.8 ± 1.0	8	143	5.8 ± 1.0	8
099.go	355	370	354	0.9 ± 18	352	296	294	1.0 ± 17	307	294	1.0 ± 17	307
130.li	352	351	348	4.23 ± 26.3	355	105	99	6.0 ± 13	117	100	4.9 ± 13	117
132.ijpeg	356	421	356	0	0	170	170	0.1 ± 0.5	3	170	$0.0 \pm 0.$	0
147.vortex	905	912	902	2.9 ± 29.9	913	299	296	2.7 ± 18	363	296	2.7 ± 18	363
164.gzip	78	102	77	0.8 ± 0.35	1	49	48	1.0 ± 0.23	50	48	1.0 ± 0.23	50
300.twolf	171	186	170	0.9 ± 0.23	1	80	78	1.6 ± 0.7	122	78	1.6 ± 0.7	122
space	87	115	84	3 ± 15	124	23.3	19	4.3 ± 9.0	35	19	4.3 ± 9.0	35

Table 3. Impact Set Comparisons

We explored why the precision gains were, on average, only a few percent. We computed how many functions had a particular influence mechanism for each of our subject applications. Table 4 shows the results expressed as a percentage of the total functions in the program. A function may have more than one influence mechanism, so the numbers across the columns need not add to 100%. For "global" influence, the percentage reported is only those functions that explicitly access global variables. Masking through parameter passing or returns is not taken into account.

app	none	global	val_in	ref_in	val_out	ref_out
008.espresso	0.2%	50%	95%	93%	74%	96%
099.go	0%	97%	85%	7.8%	39%	7.8%
130.li	2.2%	46%	94%	90%	81%	93%
132.ijpeg	0%	10%	98%	96%	30%	97%
147.vortex	0%	96%	98%	96%	94%	97%
164.gzip	0%	88%	65%	36%	37%	37%
300.twolf	0%	79%	70%	34%	98%	36%
space	2.2%	62%	95%	87%	90%	87%

Table 4. % of Funcs with a Given Influence

Table 4 provides insight into why the precision gains are small. The greatest gains in precision occur for functions with no influence, which occurs in few functions. Also for most of the applications, a large percentage of functions have the ref_in mechanism, which enables propagation of a great deal of changes from the caller to callee and back.

Table 3 also shows that the `InfluencedDynamic` and `ApproxInfluencedDyanmic` algorithms performed similarly in terms of precision, for most of the applications. For 130.li and 132.ijpeg, `ApproxInfluencedDynamic` was unable to remove as many functions from the impact sets as `InfluencedDynamic`. Even in these two cases, however, the precision gains were not significantly different. Our expectation was that `InfluencedDynamic` would be more precise. We believe that there are two reasons for the similar performances. The first is that, according to Table 4, changes are more likely to be propagated due to function parameters than global variables. The two algorithms differ only in their handling of global variables so the algorithms should obtain similar results. The second reason is the high prevalence of reference parameters and returns in the applications. These reference variables allow callee functions to potentially access all global variables of the caller. The high prevalence of reference parameters and returns allows most functions to have access to most of the global variables; effectively making both dynamic algorithms similar.

We also explored the costs associated with collecting the influence mechanisms and building the influence graph (question 2). We modified GCC to provide the callsite list and influence mechanisms of each function during compilation. GCC already has that information available so there was no extra overhead incurred to gather the callsites and influence mechanisms. Building the influence graph for the applications took around one second. The space needed to store the graph was also reasonable; 147.vortex had the largest influence graph at 19 MB. The next largest was 132.ijpeg at 2.1 MB. The costs to build the influence graph certainly appear reasonable.

Table 5 shows the time requirements needed to compute a single impact set for the different algorithms we implemented (question 3), along with the size of the execution trace for each application. Times were averaged over all functions. No time is available for `PathImpact`, as the implementation we had available computes the impact sets for all functions with one pass through the trace [5]. This is more efficient than rerunning the analysis for each function separately. As we do not yet have versions of our dynamic algorithms that compute all impacts in one pass, a timing comparison would be unfair. Instead, the sizes of the execution traces are shown to provide some context for how long the dynamic analyses take.

Conservative estimates of impacts can be computed quickly using the static analysis. The dynamic algorithms took longer, but were still reasonable, even for the larger traces. As expected, `ApproxInfluencedDynamic` executed faster than `InfluencedDynamic`. It may be expensive, however, to compute multiple impact sets. The primary cost for the dynamic algorithms is the book keeping for global variables. These operations are performed at each call and return, even if no change is currently being propa-

	Static			Dynamic		
app	icfg	infl	`CombinedStatic`	Trace SZ	`InfluencedDynamic`	`ApproxInfluencedDynamic`
008.espresso	0.8	0.06	0.87	204 MB	93	21
099.go	1.4	0.29	1.81	246 MB	96	18
130.li	0.37	0.03	0.42	1.3 GB	506	122
132.ijpeg	24	0.25	24.3	2.0 GB	682	220
147.vortex	9.1	5.0	14.3	464MB	500	105
164.gzip	0.04	0.01	0.05	9.1GB	2,300	433
300.twolf	0.42	0.03	0.46	2.5GB	593	122
space	0.05	0.01	0.07	8.2 KB	0.02	0.03

Table 5. Avg. Time (in seconds) to Compute One Impact Set

gated. If multiple impact sets are needed, then we believe the efficiency of the dynamic algorithms could be improved by computing all needed results with one pass through the execution trace (see [5]).

6.4. Summary

In summary, we found that the `CombinedStatic` algorithm and both dynamic algorithms give more precise results compared to current impact analysis techniques (question 1). On average, the gain in precision tended to be a few percent, although the gain was considerably higher in some cases. Collecting the influence mechanisms and building the influence graph could be done efficiently (question 2). Finally, the time needed to perform the static and dynamic analyses was also reasonable. Thus, while the precision gains tended for this approach were small, there is no penalty incurred in performing this analysis.

7. Conclusions and Future Work

We have demonstrated the usefulness of accounting for function influence mechanisms to provide more precise results of impact analysis. These influences can be derived by a fast scan of the program's source code. We have described both static and dynamic impact analysis algorithms that take advantage of these influence mechanisms to provide better precision (3-5%, although more in some cases). The time required to perform the analysis is reasonable.

We would like to continue exploring how much precision can be gained by utilizing more static analysis. More static analysis translates into removing unnecessary edges from the influence graph, which should translate into more precise results. Probability of impacts is another avenue that could be explored. For example, if function f is changed, then is it more likely that functions one hop away from f in the influence graph are more affected by the change than functions that are ten hops away. Such analysis may help focus software maintainers attention to those areas more likely to be impacted.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. *International Conference on Software Engineering*, 2005.

[2] R. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *International Conference on Software Maintanence*, 1993.

[3] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[4] B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. *International Conference on Software Maintanence*, 2004.

[5] B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. *European Conference on Software Maintenance and Reengineering*, 2005.

[6] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *International Conference on Software Engineering*, 2003.

[7] J. P. Loyall and S. A. Mathisen. Using dependence analysis to support the software maintenance process. *Conference on Software Maintenance*, 1993.

[8] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. *Foundations of Software Engineering*, 2003.

[9] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. *International Conference on Software Engineering*, 2004.

[10] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. *PASTE*, 2001.

[11] Standard Performance Evaluation Corporation. SPEC benchmarks. http://www.spec.org.

[12] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 2, 1995.

[13] R. J. Turver and M. Munro. Early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1), 1994.