Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems

Mithun Acharya, Brian Robinson
ABB Corporate Research
Raleigh NC USA 27606
{mithun.acharya, brian.p.robinson}@us.abb.com

ABSTRACT

Change impact analysis, i.e., knowing the potential consequences of a software change, is critical for the risk analysis, developer effort estimation, and regression testing of evolving software. Static program slicing is an attractive option for enabling routine change impact analysis for newly committed changesets during daily software build. For small programs with a few thousand lines of code, static program slicing scales well and can assist precise change impact analysis. However, as we demonstrate in this paper, static program slicing faces unique challenges when applied routinely on large and evolving industrial software systems. Despite recent advances in static program slicing, to our knowledge, there have been no studies of static change impact analysis applied on large and evolving industrial software systems. In this paper, we share our experiences in designing a static change impact analysis framework for such software systems. We have implemented our framework as a tool called *Imp* and have applied Imp on an industrial codebase with over a million lines of C/ C++ code with promising empirical results.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Languages, Measurement, Experimentation.

Keywords

Change Impact Analysis, Static Program Slicing, Empirical Study.

1. INTRODUCTION

Many software systems undergo frequent changes during the software maintenance phase. These changes may introduce new functionality to the program, fix existing defects, or adapt the system to changes in its environment. While these types of changes are useful to the end users, they also introduce a risk that the changes have unintended impacts that may cause the software to behave in an undesirable way. These undesirable behaviors may include injecting a new defect, breaking existing functionality, or decreasing the performance of the application.

Due to these risks, it is essential to understand the potential impact of a software change as early as possible. While this information is important, it is often difficult to obtain. In industry, the most commonly used technique for change impact analysis is for developers to inspect the code. Developers may get some support for this task from their development environment, in the form of calling relationships or textual search capabilities, but the majority of the task is manual. The amount of time required to determine the overall impact of a change can be large, as can the error rate.

Static program slicing is an attractive option for performing routine change impact analysis of newly committed changesets. For small programs, static slicing transparently integrates with software builds and developer IDEs, enabling quick and efficient *what-if* or risk analysis for software changes. Unlike its dynamic counterpart, static slicing is not constrained by the availability and the quality of regression test suites and can be used even before the program is execution ready.

Unfortunately, static program slicing, in spite of being researched for many years, suffers from performance issues when analyzing large systems. Change impact analysis needs to be computed at least nightly, as developers will need fast access to the impact information for several critical software engineering tasks such as risk analysis, effort estimation, and regression testing. In addition to these performance issues, program slicing can have accuracy issues as well. Recent research by Binkley et al. [8] shows that many programs contain central data structures that, if affected by a change, cause the slice size to grow too large to be useful to developers. In order to reduce the time required to determine the impact of a change and improve the accuracy, we present a fully automated framework based on static program slicing.

Our paper makes the following contributions:

- We identify and outline (with a preliminary study) the unique problems encountered by static program slicing when directly used for change impact analysis of large and evolving industrial software.
- We propose a novel framework for change impact analysis based on static program slicing, while addressing its performance and accuracy issues.
- We have implemented our framework in a tool called *Imp*[1], built over a commercial static analyzer. Imp

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...$10.00

[1] Imp is a mythological creature fond of harmless pranks

seamlessly integrates with developer IDEs and also with the nightly build environments.

- We empirically evaluate Imp on a large ABB codebase containing over a million lines of code with 10 years of development.

The rest of the paper is organized as follows. In Section 2, we present the results from our preliminary study, which motivates the design of our framework, presented in Section 3. The evaluation results are presented in Section 4. We discuss the limitations, future work, and threats to validity in Section 5 and related literature in Section 6. Finally, Section 7 concludes our paper.

2. PRELIMINARY STUDY

In this section, we present our observations from a preliminary study conducted on a large ABB software product. Our observations emphasize the problems in using static program slicing directly for change impact analysis of large and evolving industrial software. Before proceeding further, we provide a brief introduction to static program slicing.

2.1 Static Program Slicing

Static program slicing [22] refers to the computation of program points that effect or are affected by other program points. The forward slice from program point *p* includes all the program points in the forward control flow affected by the computation or conditional test at *p*. Program points (or vertices) are the most basic fragments of the source code. A program may contain multiple files, a file may contain multiple functions, a function may contain multiple lines, and a line may contain multiple vertices. In this paper, for convenience, we report the impact at the line granularity. For example, in Figure 1, the statements in the forward slice of the assignment statement `sum=0` (in Line 7) are highlighted. The assignment of zero to the variable `sum` *impacts* line 10 and 13 in the example code. We next present the experimental settings for our preliminary study and discuss our primary observations.

2.2 Subjects

We used a commercial static analyzer called *CodeSurfer* [1] from GrammaTech to directly apply static program slicing for change impact analysis. We analyzed the impact of changes between two release versions of component X of a large real-time embedded software system (Y) developed at ABB. Let us call the two release versions as version V1 (old version) and version V2 (new version). Y has been under active development for more than 10 years at ABB. The development teams were spread geographically across 2 countries with a total of roughly 100 developers. X is a major component of Y and consists of 1.18 MLOC written in C/C++. The teams used Microsoft Team Foundation Server (TFS) for version control. Roughly, a year's time had elapsed between the releases of the two versions considered in our preliminary study.

2.3 Statistics

A *change* is a contiguous block of lines either added to the newer version or that has changed between the older and the newer versions. CodeSurfer computes the impact of a given change in terms of the *number of lines impacted* in the software. For a given change, CodeSurfer uses forward slicing to compute the set of impacted lines (henceforth, called the *impact set*, denoted by *IS*).

To compute the forward slice, CodeSurfer performs several static program analyses, such as pointer and dependency analysis, by transparently integrating with the building and linking of the software. We refer to the *build-link-analysis* stage simply as the *build stage*. Once the build stage is completed, CodeSurfer, for a given change, computes the impact set as the forward slice of that change in the software using the program analysis data computed during the build stage.

Build time (denoted by *BT*) is the time required for the build stage. S*lice time* (denoted by *ST*), which is per-change, is the time required to compute the impact set of a change as its forward slice. Multiple changesets can be committed prior to a nightly build and each changeset can have multiple changes. *Average slice time* (denoted by *AvST*) is the mean and the *Total slice time* (denoted by *TST*) is the sum of slice times for all the changes to be analyzed.

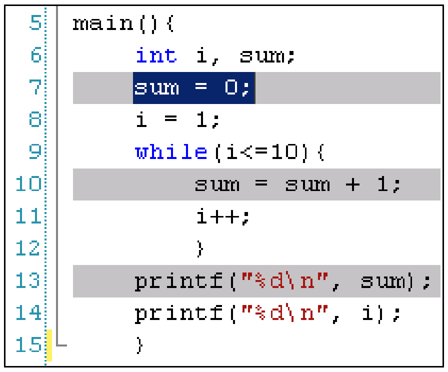

```
 5  main(){
 6      int i, sum;
 7      sum = 0;
 8      i = 1;
 9      while(i<=10){
10          sum = sum + 1;
11          i++;
12          }
13      printf("%d\n", sum);
14      printf("%d\n", i);
15      }
```

Figure 1: The forward slice of `sum=0`

2.4 Configurations

CodeSurfer has numerous configurable static analysis parameters that affect the build time and eventually the *safety* and *precision* of the impact set. An impact set of a change is *safe* if it contains every line that is *actually* impacted by the change. A safe set is *conservative* if it is also *imprecise*, i.e., if it also contains lines that are not impacted by the change (false positives). Consider, for a given change, the impact set which has all the lines in the program. Such a set is trivially safe. However, such a set is also *over* conservative if the *actual* impact of the change is not the whole program. A set that is both safe and precise is an *accurate* set.

We consider two sets of CodeSurfer configurations (or settings), termed HIGH (also denoted by *H*) and LOW (also denoted by *L*), in our preliminary study. Each CodeSurfer configuration or setting is a set of static analysis parameters. Roughly, with HIGH setting being the most precise (with respect to the underlying static analyses), we expect to obtain a conservative (and hence, safe) impact set at the expense of longer build times. With LOW setting, we expect the accuracy of the impact set to be sacrificed for faster build times. In Sections 3, we provide more details on CodeSurfer's static analysis parameters.

2.5 Experiments

In our study, the changes between versions V1 and V2 were captured with a *diff*-like interface provided by TFS. For our preliminary study, we randomly considered about thirty changes in the source code between versions V1 and V2. Our experiments were run on a 2GHz quad-core Windows Server 2008 machine with 24GB RAM and 2.8TB of disk space. We ran CodeSurfer on the changes, first with the HIGH setting and then with the LOW setting. Of the thirty changes studied, we highlight the statistics recorded for ten changes that are representative. The recorded statistics are shown in Table 1. In the next three sections, we analyze the data from our preliminary study and discuss how static slicing-based change impact analysis, when directly applied, can impede the regular software evolution process in an industrial setting.

Table 1: Number of impacted lines (LOC), Slice Time (ST), and Total Slice Time (TST) with HIGH and LOW settings for ten changes analyzed with CodeSurfer.

ID	H (LOC)	ST (m:s)	L (LOC)	ST (m:s)
1	332616	4m:22s	758	< 1s
2	339117	5m:51s	58	< 1s
3	341542	6m:12s	25	< 1s
4	341460	5m:21s	3144	1s
5	338838	5m:20s	211	< 1s
6	338665	5m:35s	92	< 1s
7	354675	15m:17s	343758	22s
8	341772	6m:52s	309925	14s
9	96	1s	93	< 1s
10	27	< 1s	27	< 1s
	TST:	**~ 55m**	**TST:**	**< 44s**

2.6 Build Time

The changesets we studied were committed to version V2 of component X. Hence, before computing the impact sets for the changes, CodeSurfer builds version V2 of component X. With the HIGH setting, the build time for X was about four days. With the LOW setting, the build time was about three hours. Clearly, with the HIGH setting, the build time is too prohibitive and thus HIGH setting builds cannot be integrated with a nightly build.

Previous studies on program slicing focus on much smaller programs and do not consider the build times [4][6][7][8]. For very large industrial systems, such as the ones analyzed in this paper, the build times are substantially larger than the per-change and total slice times.

2.7 Total Slice Time

Once the build is complete, CodeSurfer computes the impact set for each of the committed changes. In Table 1, Column 3 displays the slice times for changes with the HIGH setting. Column 5 displays the slice times for changes with the LOW setting. For the changes studied, the per-change slice time was under half a minute with LOW setting. With HIGH setting, the per-change slice time often took several minutes.

The slice time overhead may not matter when only a few changes are analyzed. But, for industrial software under active development, hundreds of changesets (and hence, an order of magnitude more changes) are committed to the source repository daily. Under these circumstances, the total slice time (TST) easily becomes very prohibitive with the HIGH setting.

2.8 Number of Lines Impacted

For the ten changes, we recorded the size of the impact set as lines of code. In Table 1, Columns 2 and 3 display the impact set sizes for the HIGH and LOW settings respectively.

With the HIGH setting, the impact sets are often very large, encompassing more than 30% of the whole program. For a few changes (change id 7 and 8 in the Table), the impact sets are very large even with the LOW setting. Large impact sets may be a problem because they could be over conservative. Also, such large sets cannot be efficiently explored by the developers inspecting the impact of a change. Based on our preliminary observations, we next outline the main hurdles faced by slicing-based change impact analysis.

2.9 Problems

Because it is safe, HIGH setting would be our choice for change impact analysis. But the HIGH setting encounters several serious problems when used for change impact analysis. Furthermore, substituting LOW setting for HIGH directly does not ameliorate the situation as we do not know how much accuracy is sacrificed for faster build times. The statistics from our preliminary study demonstrate the several problems in using static program slicing directly for change impact analysis of large software:

- The build time and the total slice time are excessive with the HIGH setting.
- For change impact analysis, the impact set with HIGH setting is safe. But the safety with HIGH setting might come at the expense of the impact set being over conservative. Due to the two aforementioned reasons, change impact analysis with HIGH setting cannot be integrated with the nightly build.
- The LOW setting has build times small enough to be integrated with the nightly build. But it is not clear how much of the impact set's accuracy is sacrificed for faster builds with the LOW setting.
- The impact set can be very large even with the LOW setting and our question on accuracy here still applies. Large impact sets cannot be easily explored by the developers for routinely inspecting the impact of the committed changesets.

In the next section, we describe our framework, which addresses these concerns. Our framework, called Imp, is implemented on top of CodeSurfer. Imp addresses the performance and accuracy issues encountered when applying static slicing directly for change impact analysis through CodeSurfer.

3. FRAMEWORK

This section is organized as follows. In Section 3.1, we describe the various static analysis parameters and configurations available to Imp through CodeSurfer. Section 3.2 provides the overview of our framework, outlining the main intuitions and the key ideas behind Imp's design. Sections 3.3 and 3.4 describe the core algorithms of our framework. Finally, Section 3.5 provides the implementation details for Imp.

Table 2: Imp configurations or settings used in our experiments

	H	L	H_cd	H_dd	H_in	H_out
nonlocals	Yes	No	Yes	Yes	Yes	Yes
range	F	F	F	F	IN	OUT
deps.	B	B	C	D	B	B
funcptrs	Yes	Yes	Yes	Yes	Yes	Yes
heap	Yes	Yes	Yes	Yes	Yes	Yes
	L_cd	**L_dd**	**L_in**	**L_out**	**H_nh**	**H_nfp**
nonlocals	No	No	No	No	Yes	Yes
range	F	F	IN	OUT	F	F
deps.	C	D	B	B	B	B
funcptrs	Yes	Yes	Yes	Yes	Yes	No
heap	Yes	Yes	Yes	Yes	No	Yes

3.1 Static Analysis Parameters

Forward slicing, required for computing the impact sets, depends on several static analysis parameters such as precision of pointer analysis, context sensitivity, flow sensitivity, non-local dependencies, and certain slicing criteria. Imp inherits the static analysis parameters of CodeSurfer and provides them as configurable options. The static analysis parameters affect the build time and eventually the accuracy of the forward slice or the impact set. We describe these parameters next.

For the pointer analysis, the precision ranges from no pointer analysis, to Steensgard's algorithm [21], to Andersen's [2] algorithm. In our experiments, we use Andersen's algorithm with the context-insensitive and flow-insensitive version of the pointer analysis [19][26] and expanded structure fields [25] (collapsing the structure fields leads to an increase in the slice size [4]). The context-sensitive and the flow-sensitive versions [9][18][23][26][11][12][13] do not scale even for an order of magnitude smaller size programs than those analyzed in this study [8]. Imp also allows its users to shut off function pointer analysis and memory allocation on the heap altogether, if required.

Computing non-local dependencies for each function in the program is a major time consuming analysis during the software build stage. The non-local dependencies of a function include all the global variables and indirectly accessed variables used or modified by a function. With Imp, the non-local dependency computation is configurable and can be shut off.

Forward slicing has three criteria – *range, dependences*, and *summary edges*. There are three possible options for slice range: full (F), across-in (IN), and across-out (OUT). For all the three options, the intra-procedural dependences, including those that connect input and output parameters in function calls are always traversed during slicing. The option across-in indicates that the traversal should go in to functions that are called, whereas the option across-out indicates that the traversal should go out to calling functions [1].

There are three options for dependences: control-dependence only (C), data-dependence only (D), and both (B). Depending on the option, the slicing follows control-dependence edges only, data-dependence edges only, or both edges. As described in [1], there is an inter-procedural control dependence edge from each call point of a function to its entry point. For a given function, there is an intra-procedural control dependence edge (1) between each call point and the call's actual parameters, (2) between the entry point to each of its formal parameters, and (3) between the entry point to each of the function's top-level statements and conditions. There is an intra-procedural data dependence edge between two points in a program if the first point may assign a value to a variable that may be used by the second point. Finally, there are data dependence edges between a function's actual parameter and the corresponding formal parameter and between the function return value and actual out parameter.

Summary edges, which model the dependences between the actual input and actual output parameters of a function, control the inter-procedural precision [17] of forward slicing. Without summary edges, a forward slice is merely a transitive closure of successors from the slicing point [1]. Summary edges are always computed for our experiments. The various Imp settings used in our experiments are summarized in Table 2.

3.2 Overview

Figure 2 outlines the algorithm for our framework. While the formalisms and details are introduced in the next section, **the intuitions and the key ideas**, illustrated by the comments in Figure 2 are as follows:

- HIGH setting analysis is expensive and might yield over conservative sets. Hence, Imp performs the expensive HIGH setting analysis infrequently (monthly, for instance, separate from the nightly build), on a sample of changesets (lines 1–4).
- Imp performs the infrequent HIGH setting impact analysis primarily to obtain *clues* in the program for possible over conservativeness. Imp computes the *Common Global Data Structures* (*CGDS*, lines 5–8) among large impact sets and the *High Impact Functions* (*HIF*) that manipulate those common global data structures (lines 9–13).
- Imp performs the faster LOW setting analysis frequently (nightly, for instance), on committed changesets (lines 14–17).
- Imp then uses the *clues* (*CGDS* and *HIF*) obtained from the infrequent HIGH setting builds to *guide* the LOW setting builds (lines 18–24) progressively expanding a large impact set to the user, if required.

In the subsequent sections, we describe the core aspects of our framework.

3.3 Infrequent HIGH Setting Analysis

3.3.1 Notations

Let *BT(s)* denote the build time for program *P* with Imp setting *s*. After the build, for a given change *c* in *P*, let *IS(c,s)* denote the impact set computed by Imp with setting *s*. Let | *IS(c,s)* | denote the size of the impact set in terms of number of lines of code in *P*. Let *ST(c,s)* denote the slice time for the change *c* in *P* with setting

s. If there are n changes (denoted by the set $C=\{c_1, c_2, \ldots, c_n\}$) among the committed changesets during a nightly build, then the total slice time, the sum of individual slice times for each change in program P, is denoted by

$$TST(C,s) = \sum_{i=1}^{n} ST(c_i, s)$$

```
foreach(INFREQUENT interval) // monthly, for instance
    Let $S=\{c_1,c_2,c_3,\ldots,c_m\}$ // $m$ sampled changesets
    foreach($c$ in $S$)
        compute $IS(c,H)$ // HIGH setting impact analysis
        if($|IS(c,H)|>LG$) // large set
            compute $GFP(c,H)$ // global footprint
    // common global data structures among large sets
    $CGDS=\{\bigcap_{i=1}^{m} GFP(c_i,H) : |IS(c_i,H)|>LG\}$
    // compute the set of high impact functions HIF
    foreach($f$ in $P$)
        compute $GUM(f,H)$ // function global footprint
        if $GUM(f,H) \cap CGDS \neq \emptyset$
            $HIF += f$ // high impact function

foreach(FREQUENT interval) // nightly build, for instance
    Let $C=\{c_1,c_2,c_3,\ldots,c_n\}$ // $n$ committed changesets
    foreach($c$ in $C$) {
        compute $IS(c,L)$ // LOW setting impact analysis
        if($|IS(c,L)|<LG$) // not large
            output $IS(c,L)$
            output $CGDS \cap GFP(IS(c,L))$
            output $f : f$ in $IS(c,L)$ and $f$ in $HIF$
        else // truly large impact
            compute and output $IS(c,L\_dd)$ // smaller subset
            expand $IS(c,L\_dd)$ into $IS(c,L)$ // on demand
```

Figure 2: Algorithm

The total time overhead for Imp during a nightly build with setting s is denoted by

$$T(C,s) = BT(s) + TST(C,s)$$

3.3.2 *Uncomputability of Golden Standard*

Let $GS(c)$ denote the *actual* impact set (the *golden standard*) of change c, the accurate impact set with 100% precision and recall. In fact, such a set is uncomputable because of the undecidability of the required static analyses. Even with the most precise (with respect to the underlying static analysis) Imp setting HIGH, the resulting impact set ($IS(c,H)$) is at best a conservative approximation of the golden standard $GS(c)$, i.e., $IS(c,H) \supseteq GS(c)$. Hence, for the impact set with HIGH setting, the recall r=100% (safe) and the precision, denoted by $p=|GS(c)|/|IS(c,H)|$.

3.3.3 *Impact Explosion*

It is not possible to accurately determine p as the golden standard, $GS(c)$, is not computable. For the sake of discussion, let us assume that the impact set $IS(c,s)$ has low precision if $p<\omega$ for some small threshold ω. If the precision of the impact set is low, we will have too many false positives. On the other hand, if the impact set computed by Imp is large, it becomes difficult for the users to navigate the impact of a given change. We set a threshold (number of lines in code), denoted by LG, beyond which the impact set is deemed as *large*. Hence, low precision impact sets that are also large are especially a problem. We call a low precision impact set that is safe and large as an *exploded* impact set. Formally, $IS(c,s)$ is an exploded set *iff*

$$p < \omega \wedge IS(c,s) \supseteq GS(c) \wedge |IS(c,s)| > LG$$

With the HIGH setting, the impact set $IS(c,H)$, a superset of $GS(c)$, is conservative (and hence, safe). Hence, $IS(c,H)$ is exploded *iff* $p<\omega$ and $|IS(c,H)|>LG$. Since p cannot be determined, definitively establishing explosion for an impact set with HIGH setting is not possible.

However, as our empirical results demonstrate in Section 4.1, we can obtain several *hints* that indicate explosion for large impact sets computed with the HIGH setting. Our results indicate that the HIGH setting impact analysis on changes yields an exploded impact set fairly often.

In spite of large build times, performing HIGH setting impact analysis on a range of changes infrequently (for example, once a month), separate from the nightly build, provides valuable *clues* about why an impact explosion might occur. As we describe next, Imp analyzes the data structures and functions in the program under analysis to find such clues.

3.3.4 *Global Footprint of Large Impact Sets*

Let us define $GFP(c,s)$ as the set of global variables in the program P that are either used or modified by the impact set of the change, $IS(c,s)$. We call the set $GFP(c,s)$ as the *global footprint* of the change c.

Imp performs HIGH setting impact analysis on a range of m uniformly sampled changes (denoted by set $S=\{c_1, c_2, \ldots, c_m\}$) infrequently (for example, once a month, separate from the nightly build). The Imp time overhead for the analysis is $T(S,H)$. We choose m such that we can expect a sufficient number of impact set explosions.

Although an exploded set is large by definition, a large impact set does not necessarily imply explosion. As discussed before, it is not possible to definitively identify impact explosion. Hence, Imp conservatively assumes that a large impact set with HIGH setting ($IS(c,H)>LG$) is an exploded set. Imp then computes the common global data structures (denoted by *CGDS*) between the possibly exploded sets with HIGH setting. Formally,

$$CGDS = \{\bigcap_{i=1}^{m} GFP(c_i, H) : |IS(c_i, H)| > LG\}$$

3.3.5 Global Footprint of Functions

After computing the common global data structures (*CGDS*), Imp computes the set of functions in the program that manipulate (use or modify) them. Let us define *GUM(f,s)* as the set of global variables or data structures that are used or modified by a function *f* in program *P*, when *P* is built with setting *s*. For each function *f* in program *P*, Imp computes *GUM(f,H)* before each infrequent HIGH setting impact analysis. Let us define the set *HIF*, the set of *high impact functions* in program *P* built with HIGH setting, as the set of functions that manipulate some common global data structures (*CGDS*). Formally,

$$HIF = \{f \in P : GUM(f, H) \cap CGDS \neq \phi\}$$

In the next section, we describe how Imp uses the set *HIF* to display large impact sets to the user.

3.4 Frequent LOW Setting Analysis

During the regular nightly build, Imp performs impact analysis with a LOW setting build. For a given change *c* in the changeset, Imp computes the impact set *IS(c,L)*. If the impact set *IS(c,L)* is not large, Imp is done analyzing the change *c* and outputs *IS(c,L)*. Imp also outputs all the functions in *IS(c,L)* that are also in the set *HIF* (high impact functions). On the other hand, if the impact set *IS(c,L)* is large, Imp progressively expands the large impact set to the user starting with a smaller subset. The reasoning for the aforementioned choices of Imp stems from our empirical results, which we discuss next.

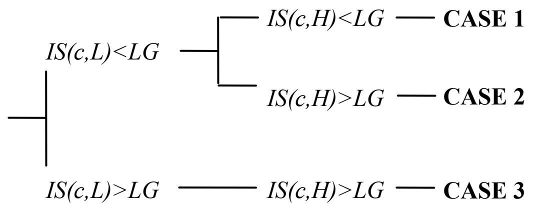


Figure 3: Various cases with LOW setting impact analysis

Recall that the HIGH setting impact analysis is only done infrequently and is not integrated with the nightly build. Our empirical results show that *IS(c,L)⊆IS(c,H)* always. This is not surprising as the local dependency computation is the same for both HIGH and LOW settings. We are interested in the set *IS(c,H)-IS(c,L)*, i.e., the impacted lines that might have been missed by the LOW setting impact analysis. In other words, we are interested in the recall of the LOW setting impact set with respect to the HIGH setting impact set. As Figure 3 shows, there are three possible cases.

Case 1: In this case, both the HIGH setting and LOW setting impact sets are not large. We empirically show that the LOW setting impact sets have reasonably high recall with respect to the HIGH setting impact sets in this case. In fact, it turns out that if both HIGH and LOW setting impact sets are less than a threshold *SM* lines, then the recall of the LOW setting impact set is almost 100% with respect to the HIGH impact set. We empirically estimate the value of *SM* in Section 4.2.

Case 2: In this case, the LOW setting impact set is not large and the HIGH setting impact set is large. The difference, *IS(c,H)-IS(c,L)*, could be substantial.

For cases 1 and 2, Imp displays the impact in three parts. First, it directly displays the small LOW setting impact set *IS(c,L)*. Next, it displays the common global data structures in the global footprint of the impact set. Finally, it displays all the functions in *IS(c,L)* that are also in the set *HIF* (*high impact functions*, discussed in Section 3.3.5). With the HIGH setting, common global data structures and *HIF* functions, if hit, are the reason why the impact set might explode. The user may further choose to expand the *CGDS* or the *HIF* functions in the impact set *IS(c,L)*.

Case 3: In case 3 of Figure 3, both *IS(c,L)* and *IS(c,H)* are large. In this case, our empirical results indicate that *IS(c,H)* is probably not exploded and the impact set is *truly* large. Roughly, an impact set IS is truly large if it significantly overlaps with the golden standard, when the golden standard itself is large.

Expanding a truly large impact set to the user: In case 3, both LOW and HIGH setting impact sets are large. This would happen if there are large control dependence clusters in the program. For example, if a function *f* directly calls most other functions in the program, then the impact of function *f* is truly large. In such cases, our empirical results indicate that the LOW setting impact sets with no control dependency (*L_dd*) should yield a smaller subset. Imp starts with the smaller subset *IS(c,L_dd)*, gradually expanding it to the full impact set *IS(c,L)* on demand to the user.

To visualize large impact sets, Imp provides a *step-through* feature similar to the ones available with the popular IDE-based debuggers. With Imp, a user can select a point (denoted by A) of interest in the program, compute its impact, and step through the impact of the program one line/function/file at a time. For expanding the view of the impact set, the next line/function/file can be selected based on proximity, along the AST, or along the control flow. The user can also specify another point (denoted by B) in the program as a *breakpoint* and find if A impacts B.

The case (*IS(c,L)>LG* and *IS(c,H)<LG*) is not possible as *IS(c,L)* is always a subset of *IS(c,H)*.

3.5 Implementation

Using the APIs provided by CodeSurfer, Imp is implemented as an AddIn that integrates with the Visual Studio IDE. The Imp screenshot shown in Figure 4 captures a usage scenario for Imp – a user selects lines 14–16 and invokes the Imp AddIn (accessible from the Tools menu) to compute the impact of the assignment statements `k=3` and `french=1`. Imp then computes the impact and highlights the impacted lines (such as lines 19 and 26) and files (such as `test.c`) in the Visual Studio IDE for the user to browse through. The user may also choose the step-through visualization feature (discussed in Section 3.4) to visualize the impact computed by Imp. Imp also presents the concise summary of the impact analysis (number of files, functions, lines, and vertices impacted) to the user. A configurable commandline version of Imp is also available, which can be transparently integrated with the nightly build environments.

4. EMPIRICAL EVALUATION

The experimental setup for our evaluation was similar to that used for our preliminary study. In our preliminary study, we had analyzed two release versions, version V1 (older version) and version V2 (newer version) of component X. For our empirical evaluation, we retained version V2 as the newer version. To cross validate our observations from the preliminary study, we used a development version (called V0) as the older version instead of version V1. Version V0 was older than version V1 by a couple of months. 70 source files had changed between versions V0 and V2 and we used Imp to analyze 147 changes between the two versions. The changes selected were uniformly spread across the changed files and we did not pick more than 3-5 changes from the same file to avoid similar impact sets.

Our empirical results validate our observations from the preliminary study regarding impact set size, as we describe next. In all our graphs, the y-axis represents the impact set size as the number of lines of code and the x-axis represents the change id. Figure 5 plots the impact set sizes of HIGH and LOW settings for the 147 selected changes (sorted in decreasing order of HIGH setting impact size). As shown in the graph, there were a number of large impact sets, both with HIGH and LOW settings, which cover more than 30% of the whole program.

Next, we outline our evaluation questions.

1. How often does the impact set explode with HIGH setting? What are the hints for a possible explosion? Why does it explode?
2. When both HIGH and LOW setting impact sets are small, what is the recall of LOW setting impact set with respect to HIGH setting impact set? How often is LOW setting impact small and HIGH setting impact large? Why?
3. How often is the impact set truly large? Why is it truly large? For truly large sets, will a user always have a small subset to start with?

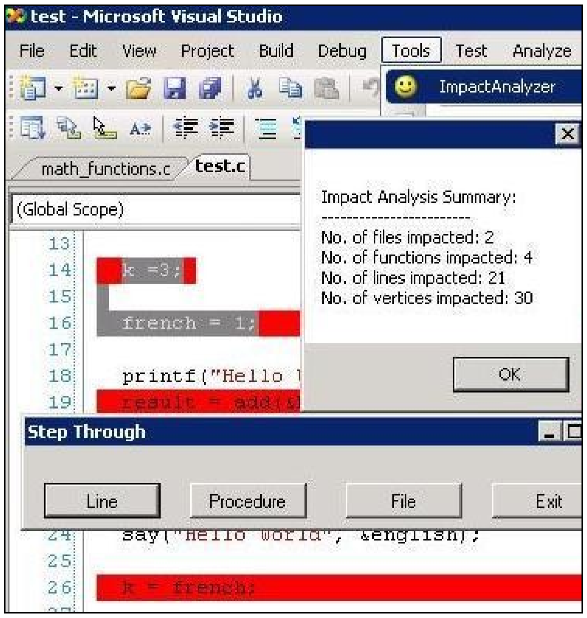
To evaluate these, in the subsequent sections, we answer a series of specific sub-questions, backed by our empirical data.

4.1 Question 1

How often does the impact set explode with HIGH setting? There were three distinct clusters grouped by size with the HIGH setting. Of the 147 changes analyzed, 66 (~45%) had impact size greater than 300,000 lines of code (~30% of the program). 13 changes (~9%) had impact set size between 25,000 and 50,000 lines of code. The remaining 68 (~46%) changes had impact size less than 10,000.

How similar are the exploded impact sets? We compared the largest impact set in the largest cluster with the next 10 largest impact sets in that cluster. The impact sets were very similar with high precision and recall with respect to the largest set in the cluster. On an average, the precision was close to 100% and the recall was close to 98.5%.

How similar are the global footprint of the changes with exploded impact sets? The global footprints were similar among clusters. For instance, the global footprint of the 15 largest impact sets from the largest cluster was exactly same with roughly 25 global variables (181, if structure fields are counted). About 20 global variables were common between all 66 changes in the largest cluster. Most global variables were large central structures.



```
k =3;
french = 1;
printf("Hello 
result = add(&
say("Hello world", &english);
k = french;
```

```
Impact Analysis Summary:
------------------------
No. of files impacted: 2
No. of functions impacted: 4
No. of lines impacted: 21
No. of vertices impacted: 30
```

Figure 4: Imp screenshots

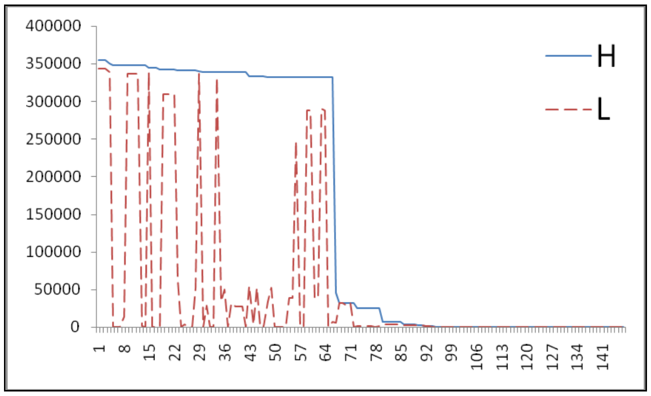


Figure 5: Impact set sizes with HIGH (H) and LOW (L) settings

Similar impact sets and global footprints for large sets strongly indicate that certain common global data structures are responsible for explosion with the HIGH setting. This role of common global data structures in creation of large slices was also confirmed by Binkley et al. [6]. Figure 6 further reinforces our observation. The impact set computed with HIGH setting with only data dependency (*H_dd*) show a strong correlation between the exploded sets and the data dependences they share.

Apart from common global data structures, are there other factors that influence the impact set size with HIGH setting? We studied the effect of function pointers and heap variables (two potential causes of large impact sets) on the impact set size. In Figure 7, we plot the impact set sizes computed without the function pointers (setting *H_nfp*) and without the heap allocation (setting *H_nh*). From the graph, it is obvious that the function pointers and variables allocated on the heap have little effect on the impact set size, ruling them out.

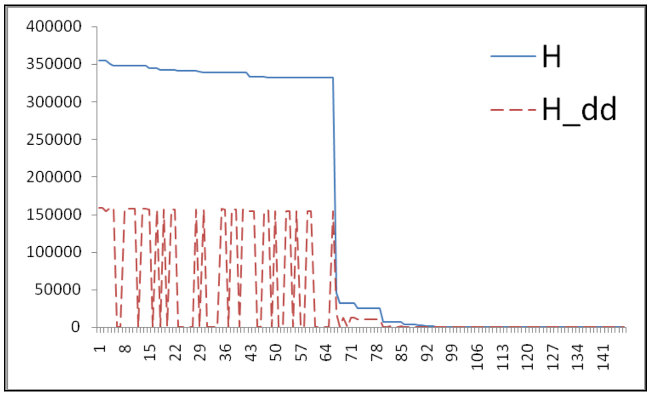


Figure 6: Impact set sizes with HIGH with only data dependency (H_dd)

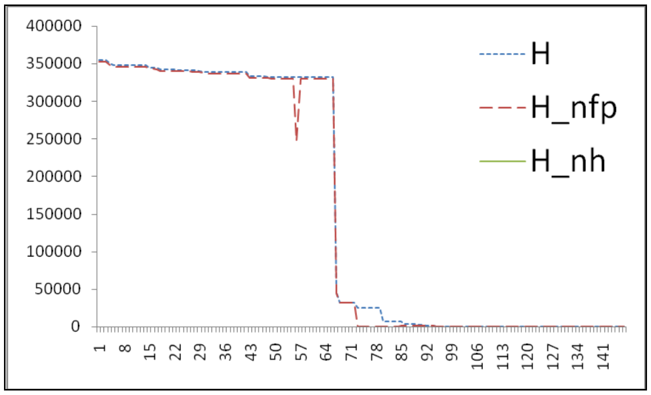


Figure 7: Impact set sizes with HIGH without function pointers (H_nfp) and without heap allocation (H_nh)

4.2 Question 2

How often are both HIGH and LOW setting impact sets small? This is case 1 of Figure 3. Of the 147 changes considered, 58 changes had impact set sizes less than 2000 for both HIGH and LOW setting. 10 changes had size between 2000 and 10000.

When both HIGH and LOW setting impact sets are small, what is the recall of the LOW setting impact set with respect to HIGH setting impact set? The precision of LOW setting impact set with respect to HIGH setting impact set was always 100% for all the changes we considered because the local dependency computation is the same for both HIGH and LOW settings. For 11 changes with LOW setting impact set size between 100 and 2000 lines, the recall with respect to HIGH setting impact set was on an average 99.6%. However, when we considered 21 changes with LOW setting impact set size between 100 and 10000 lines, the recall dropped to 76.5%. From our empirical results, the value of *SM* (discussed in Section 3.4) was about 2000 lines. The HIGH setting and LOW setting impact sets with less than *SM*=2000 lines were almost identical.

How often is LOW setting impact small and HIGH setting impact large? This is case 2 of Figure 3. There were 46 changes for which the HIGH setting impact set had more than 300,000 lines of code with LOW setting impact set having less than 300,000 lines. In such cases, all changes having LOW setting impact sets with more than 3000 lines of code had common global data structures (*CGDS*) in their global footprint, explaining why the impact set is large with the HIGH setting.

4.3 Question 3

How often is the impact set truly large? This is case 3 of Figure 3, where both HIGH and LOW setting impact sets are large. Of the 147 changes considered, 16 changes had large impact sets (more than 300,000 lines of code) with both HIGH and LOW settings. 42 changes had impact sets with more than 10,000 lines of code with HIGH and LOW.

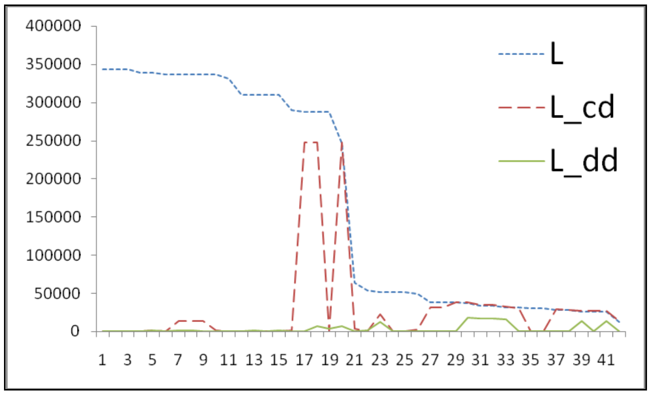


Figure 8: Decoupling control (L_dd) and data dependences (L_dd) with LOW (L) setting.

How do impact sets with settings L_cd and L_dd, compare to that with L?

We studied the effect of decoupling control and data flow dependencies with the LOW setting. A LOW setting impact set is always a superset of its decoupled versions. Figure 8 plots the impact set sizes with *L_cd* and *L_dd* for 42 changes in decreasing order of LOW setting impact set size. As the graph shows, impact sets with *L_cd* setting can be large. Impact sets with *L_cd* setting expose the large control dependency clusters, which create truly large impact sets with the LOW setting. On the other hand, the impact sets with *L_dd* setting were always small with an average size of about 3000 LOC. Hence, when the LOW setting impact sets are large, a user may start with the impact set with *L_dd* setting due to its small size.

5. DISCUSSION

Though our framework addresses the presented problems with static program slicing, it is still conservative. For example, Imp always expands a truly large set with LOW setting to the user (even if it starts with a smaller subset by performing a data dependency only analysis with setting *L_dd*). We need further investigation and empirical data to study the interplay of control and data dependences to establish a set as truly large. Additionally, when the impact set is small with LOW setting, Imp also outputs the high impact functions, if any, in the impact set.

Further research is required to devise efficient algorithms for expanding *good part* of large impact sets due to data dependences and high impact functions.

The computation of common global data structures (*CGDS*) and high impact functions (*HIF*) depend on the sampled set of changes (set *S* with cardinality *m*). Ideally, for the infrequent HIGH setting impact analysis, the best sample set is the set of all changes that is available. Hence, within the time constraints, the value of *m* should be as high as possible. Also, the changes in the sample set should be uniformly spread across the code. The number of changes and their spread might also influence the number of distinct clusters we observe with HIGH setting analysis (large plateaus in Figure 5).

Apart from refining the algorithms in Imp, we also plan to integrate Imp with visualization engines and bug repositories to obtain visual aids for software risk analysis.

Our empirical evaluation may suffer from threats to external validity as the results gathered by Imp may be specific to the system analyzed. However, researchers have confirmed some of our observations (such as large slice sizes [6]) in other types of software, including open source. Internal validity may be threatened by defects in Imp or the underlying static analysis engine. To mitigate this threat, we thoroughly tested Imp on several smaller programs, validating the impact sets manually.

6. RELATED WORK

We categorize the literature related to our work into three categories: static program slicing, dynamic impact analysis, and history-based impact analysis.

6.1 Static Program Slicing

Our work is most directly related to the recent empirical studies [4][7][8] by Binkley, Harman, and his colleagues on static program slicing. They empirically investigate the different aspects that affect slice size and the impact of different optimization techniques on slice computation time. The software analyzed by these studies are at least an order of magnitude smaller than those analyzed by Imp. Also, the studies do not factor in the build times and the total slice times. Finally, they do not investigage the time and accuracy tradeoffs required for designing a practical change impact analysis framework like Imp. They also study [5][6] the impact of global variables on predicates, program dependence, and dependence clusters. Our observations are in line with these studies in that we observe large dependence clusters because of global variables. However, beyond the identification of dependence clusters, our focus is on how to deal with these dependence clusters to facilitate practical change impact analysis.

6.2 Dynamic Impact Analysis

Dynamic impact analysis techniques [16] such as *PathImpact* [14] and *CoverageImpact* [15] rely on dynamic information such as test suite executions, operation profile data, and execution in the field. The impact computation is restricted to the *observed* program behavior. In contrast, the impact computation by Imp spans the *entire* source code. While Imp is designed to seamlessly integrate with the nightly or regular build process, dynamic impact analysis techniques are more suitable to be integrated with the regular/regression testing phase. The static and dynamic approaches, however, are complementary [3].

6.3 History-Based Impact Analysis

For a given change in a software system, approaches [9][20][24][27] based on change and bug histories exist that compute the co-changes – *What artifacts (such as files, classes, methods, and lines) should also change?* These techniques have the advantage that they can also identify non-source code artifacts that are frequenty modified with a given change.

In contrast, the goal of our framework is to compute the *complete* impact of a change in the source code as accurately as possible within the time constraints. Our framework only requires the availability of the source code for analyzing the impact of a change. History-based impact analysis techniques require fairly matured and stable software change history or bug repositories, which might not be available.

7. CONCLUSIONS

In this paper, we shared our experiences in designing, implementing, and evaluating Imp, a static change impact analysis framework for large software systems. To our knowledge, no previous studies or frameworks exist that investigate static program slicing for change impact analysis of large and evolving industrial software products. We are the first ones to identify and address the unique challenges in designing a static slicing-based change impact analysis framework for systems with over a million lines of code. For such large systems, build times and total slice times, not considered by any of the previous work, become substantial and a major road block for practical change impact analysis. To address these issues, a careful investigation of time and accuracy tradeoffs is required, which is accomplished by our framework.

8. ACKNOWLEDGMENTS

We thank GrammaTech Inc. for providing timely support for CodeSurfer, the static analysis tool used in this paper.

9. REFERENCES

[1] *CodeSurfer*. GrammaTech Inc. http://www.grammatech.com/products/codesurfer

[2] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis. DIKU University of Copenhagen, 1994.

[3] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology (IST) Special Issue on Program Slicing*, 40(11–12), pages 583–594, 1998.

[4] D. Binkley, M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 44–53, 2003.

[5] D. Binkley, M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering (TSE)*, 30(11), pages 715–735, 2004.

[6] D. Binkley, M. Harman, Y. Hassoun, S. Islam, Z. Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Software Systems (JSS)*, 83(1), pages 96–107, 2010.

[7] D. Binkley, M. Harman, J. Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1), pages 3:1–3:33, 2007.

[8] D. Binkley, N. Gold, M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2), pages 2:1–2:32, 2007.

[9] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, pages 47–56, 1988.

[10] G. Canfora, L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of Workshop on Mining Software Repositories (MSR)*, pages 105– 111, 2006.

[11] J. D. Choi, M. Burke, P. Carini. Efficient flow sensitive interprocedural computation of pointer induced aliases and side effects. In *Proceedings of the Conference on Principles of Programming Languages (POPL)*, pages 232–245, 1993.

[12] M. Emami, R. Ghiya, L. Hendren. Context sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, pages 242–256, 1994.

[13] W. Landi, B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, pages 235–248, 1992.

[14] J. Law, G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 308–318, 2003.

[15] A. Orso, T. Apiwattanapong, M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 128–137, 2003.

[16] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 491–500, 2004.

[17] T. Reps, S. Horwitz, M. Sagiv, G. Rosay. Speeding up slicing. In *Proceedings of the Symposium on Foundations of Software Engineering (FSE)*, pages 11–20, 1994.

[18] E. Ruf. Context-sensitive alias analysis reconsidered. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, pages 13–22, 1995.

[19] M. Shapiro, S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the Conference on Principles of Programming Languages (POPL)*, pages 1–14, 1997.

[20] M. Sheriff, L. Williams. Empirical software change impact analysis using singular value decomposition. In *Proceedings of the International Conference on Software Testing (ICST)*, pages 268–277, 2008.

[21] B. Steensgard. Points-to analysis in almost linear time. In *Proceedings of the Conference on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[22] M. Weiser. Program Slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 439–449, 1981.

[23] R. Wilson, M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, pages 1–12, 1995.

[24] A. Ying, G. Murphy, R. Ng, M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering (TSE)*, 30(9), pages 574–586, 2004.

[25] S. Yong, S. Horwitz, T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, pages 91–103, 1999.

[26] S. Zhang, B. Ryder, W. Landi. Programming decomposition for pointer aliasing: a step towards practical analyses. In *Proceedings of the Symposium on Foundations of Software Engineering (FSE)*, pages 81–92, 1996.

[27] T. Zimmermann, P. Weissberger, S. Diehl, A. Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 563–572. 2004.