A Comparison of Online and Dynamic Impact Analysis Algorithms

Ben Breech, Mike Tegtmeyer and Lori Pollock
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{breech, tegtmeye, pollock}@cis.udel.edu

Abstract

Impact analysis is the process of determining the effect, or impact, of a change to a software system. Dynamic impact analysis uses data obtained from executing a program to perform analysis after program termination for determining impacts more in line with how a program is used. Online impact analysis has the same goal, but is performed concurrently with program execution. While some of the tradeoffs between dynamic algorithms have been studied, no such study has been performed for online algorithms. In this paper, we present such a study by comparing two online algorithms and two previously published dynamic algorithms in terms of their space overhead, time for computation, computed impact sets, and scalability. Our results indicate that performing impact analysis online can be more scalable than the dynamic counterparts.

1. Introduction

Software change impact analysis has the goal of determining the potential effects (or impacts) on a software system resulting from a program change [4]. This knowledge 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 a change is made allows a software engineer to determine what components may be affected by the change and gauge the cost of the change. After a change is made, impact analysis can be used to guide regression test efforts by reducing the test cases to be run to those cases that traverse the change.

Many impact analyses (such as [5, 18, 23]) rely upon *static* techniques such as slicing and call graph traversals, which take into account all possible program inputs and behaviors. These conservative analyses can cause the results ("impact sets") to be overly large with respect to how the program is actually being used.

Recent work has focused on *dynamic* impact analysis, which addresses these concerns by analyzing information obtained from the program during particular executions [6, 13, 14, 16, 17]. The resulting impact analysis is not conservative due to the reliance upon program execution, which can lead to different results for different inputs, for the same proposed change. However, the dynamic impact analysis provides results that more accurately reflect how a program is actually being used which can decrease the amount of time a software engineer has to spend retesting different components.

Law and Rothermel developed `PathImpact` [14], which utilizes whole path profiling [12] to obtain a compressed representation of the function execution trace of a program. The program's trace representation is then analyzed, after the program has finished executing, to obtain the impact sets. Relative to static techniques, `PathImpact` can provide more precise impact sets for a given set of execution profiles at the cost of having to wait for the compression to take place and storage of the compressed trace. These drawbacks may make the technique infeasible for collecting information from the field because, as noted in [16], "(1) the size of execution traces generated during execution can easily approach thousands of megabytes, and (2) algorithms that compress the traces to a reasonable size can be computationally expensive and cannot be straightforwardly used on-line, while the execution traces are produced."

`CoverageImpact` is a dynamic impact analysis algorithm, which was put forth by Orso et al. [16]. Their approach is to gather function coverage information from a particular execution of the program, perform static slicing, and merge the results together to obtain the impact sets. The instrumentation is very lightweight and can be gathered quickly without adding large overhead to the execution of the program. However, the impact sets produced may lack precision. Recently, Orso et al. [17] performed an experimental study that examined the tradeoffs between `PathImpact` and `CoverageImpact` in terms of precision and overhead required to use them. They

found that PathImpact, while usually more precise than CoverageImpact, may be too impractical due to large temporal overhead.

In this paper, we present *online* dynamic impact analysis algorithms and compare them with their dynamic counterparts. By online, we mean that the impact analysis operates concurrently with the execution of the program being analyzed. No post program execution phase is required. Two of the online algorithms we present utilize *dynamic* compilers to analyze the program as it is executing. The cost paid for this analysis is that the program executes for a longer period of time than it otherwise would.

In this paper, we empirically compare four different algorithms; two of them dynamic and two online, in terms of temporal and spatial overhead, and their scalability. All of the algorithms work at the relatively coarse level of functions. This may cause conservative overestimation of impacts due to lack of detailed information, but the algorithms provide useful results, and show more promise for scalability to larger programs. We are interested in how the algorithms scale with increasing program size, how they generally perform for a variety of applications, and what types of program characteristics may make one algorithm more useful than others. An algorithmic analysis can be used to gain some insight into the tradeoffs involved in using these algorithms, but experimental studies must be performed to provide a better sense of the tradeoffs in practice.

This paper is organized as follows; sections 2 and 3 provide an overview of the dynamic and online impact analysis algorithms, respectively, targeted by this study. Section 4 presents an analytical comparison of the algorithms. Section 5 describes the experimental study and results, while sections 6 and 7 provide related work and conclusions, respectively.

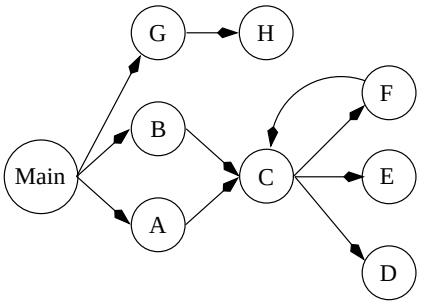
2. Dynamic Impact Analysis

In this section, we discuss two dynamic impact analysis algorithms, PathImpact [14] and CoverageImpact [16]. By dynamic, we mean that the algorithms obtain data from an execution of the program and then perform analysis after the program has finished ("off-line").

2.1. Path-based Impact Analyses

PathImpact (PI) is a dynamic impact analysis algorithm, proposed by Law and Rothermel [13, 14], that uses the execution trace of a program's functions to calculate the impact sets. These traces are obtained by instrumenting the source (or binary) prior to execution. As execution traces can become quite large and inconvenient (or impossible) to store, PI makes use of the SEQUITUR [15] compression algorithm, which is an online (i.e., can be run as the program is executing), linear compression algorithm that builds a context free grammar for a given input string. Larus [12] demonstrated how to use SEQUITUR to build a DAG representation of execution traces of a program.

Once SEQUITUR is finished, PI scans the DAG representation to calculate the impact of a proposed function change. Included in 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" [14]. PI starts at the first invocation of p and then moves forward through the trace, identifying functions called after p, until the end-of-trace symbol is found. PI then moves backward to identify functions that p could impact through its return.



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

Figure 1. Example call graph

Consider the example call graph and execution trace given in figure 1. To calculate the impact set for G, PI first adds G and then scans forward for the end-of-trace symbol, x. Functions A, C, F, D and E would be added. PI then scans backward to find functions that G could possibly return into. This scan adds Main into the set giving a final impact set for G as {Main, G, A, C, F, D, E}. Note if no data dependencies exist between G and the later functions, A, C, F, D and E, then extra impacts are reported where none exist. PI, however, will not miss impacts and, thus, is safe relative to the execution profiles used. Applying PI to E, we obtain {Main, E, D, A, C}. F and G are not added here because E cannot return into them.

2.2. Coverage-based Impact Analysis

CoverageImpact (CI) is a dynamic impact analysis algorithm developed Orso et al. [16]. Rather than analyzing an entire trace, CI analyzes function coverage information (i.e., what functions were called during execution) combined with slicing information to calculate the impact sets. The coverage information can be obtained through lightweight instrumentation of either source or binary.

To calculate the impact on a function m, a *static* forward slice is first computed for all the variable definitions

contained in m. The set of all functions found in the slice is then intersected with the set of all functions called during a particular execution involving m. The impact of m for that execution is the resulting set. CI is thus a hybrid approach that utilizes dynamic information to make a static technique more sensitive to a particular execution profile.

For the example given in figure 1, the function coverage information would be {Main, G, A, C, F, D, E}. To calculate the impact for function G, we first compute the functions in a static forward slice starting from all the variables in G. Let us assume that all the functions in the example are included in this slice. The intersection of the slice functions and covered functions gives us an impact for G of {Main, G, A, C, F, D, E}, which, for this example, is the same as the set calculated using PI. Note, however, that if there are no data dependencies between G and the later functions, A, C, F, D and E, then they would not be included in the impact set of G. We can do the same calculation for E and, assuming the slice functions are Main, A, B, C, D, E and F, we would obtain {Main, A, C, F, D, E} as the impact set for E. Note that, in this example, CI added an item to the impact set that PI would not include, because CI only examines coverage information and not relative ordering of function executions.

A modification to CI, as described in [17], approximates slicing on program code by computing reachability on the interprocedural control flow graph representation of the program. This modification, which will be used throughout the rest of the paper, is easier to implement and promises more scalability than slicing, but results in reduced precision.

3. Online Impact Analyses

Online impact analysis performs all analysis as the program is executing. This is an important difference from the dynamic counterparts, which collect data as the program executes, but process that data after the program is finished. This difference suggests that the online algorithms may be more scalable. As is the case with dynamic impact analysis, the online algorithms may not be as conservative as static techniques because of their reliance on a particular program input. We present two classes of online algorithms; the first makes use of program instrumentation and the second uses a dynamic compiler to perform the analysis.

3.1. Using Instrumentation

Figure 2 presents the algorithm for PathImpact_Allin1 (PI_Allin1), which has motivation identical to that of PI in that the impact of function f is any function that f can return into as well as any function that is called after f. In contrast to PI, PI_Allin1 calculates the impact set of every function executed using only one pass through a trace. Furthermore, PI_Allin1 does not need to scan through a trace, making it ideal for online analysis.

Algorithm: PathImpact_Allin1
Input: Function calls and returns of program P
Output: Impact sets of all functions in P

Let M be a matrix of impacts, initially empty
  /*M_{ij} = 1 if function $j \in$ impact set of function i.*/
Let BackFuncCnt be an array of integers, initially empty
  /*BackFuncCnt[i] = # times function i is currently on the call stack */

Begin executing program P
While not program exit, DO
  When function f is called, DO
    IF first execution of f, THEN
      Add one row and one column to M, labeled f
      Set all entries of M_f to 1
      FOR ALL functions i in BackFuncCnt, DO
        If BackFuncCnt[i] == 0, then M_{fi} = 0
      Add one element to BackFuncCnt array
      BackFuncCnt[f] = 0
    FOR ALL rows i in M, DO
      M_{if} = 1
    BackFuncCnt[f]++
    Push f onto the call stack

  When function return occurs, DO
    Let f be function on top of call stack
    Pop call stack
    BackFuncCnt[f]−−
End executing program

Figure 2. PathImpact_Allin1 algorithm

For the example in Figure 1, PI_Allin1 starts by initializing a matrix and an array to be empty. The matrix, an array of bit vectors, grows as each function is seen for the first time. The interpretation is that $M_{ij} = 1$ if function j is in the impact set of function i. BackFuncCnt is an array of integers that grows as each function is seen for the first time. BackFuncCnt[i] gives the number of times function i appears on the call stack at the moment. If BackFuncCnt[i] = 0, then i has returned. This enables PI_Allin1 to discover which functions are not currently executing, and, therefore, which functions cannot be impacted by the return of the currently executing function.

When function main starts executing, the matrix is resized to a 1 x 1 matrix, initialized to 1 (i.e., a change to main would impact itself), and BackFuncCnt[main] is set to 1. G then starts executing so the matrix is resized again with the new elements set to 1, and BackFuncCnt[G] = 1. When finished executing, G is popped off the call stack, and its entry in BackFuncCnt is decremented. The situation after G returns is shown in figure 3(a). Now A starts execut-

ing; the matrix is resized with new elements initialized to 1. Since G is no longer executing, BackFuncCnt[G] = 0, and we set $M_{AG} = 0$ (i.e., the return from A cannot impact G). The final matrix for the example is shown in Figure 3(b). When the program terminates, the impacts for a given function can be found simply by traversing the appropriate row in the matrix. In this case, we find the impact for G to be {Main, G, A, C, F, D, E}, the same as PI. For E, we find {Main, A, C, D, E}.

m	impact(m)	
	Main	G
Main	1	1
G	1	1

BackFuncCnt:[Main]=1
BackFuncCnt:[G]=0

(a) After G returns

m	impact(m)						
	Main	G	A	C	F	D	E
Main	1	1	1	1	1	1	1
G	1	1	1	1	1	1	1
A	1	0	1	1	1	1	1
C	1	0	1	1	1	1	1
F	1	0	1	1	1	1	1
D	1	0	1	1	0	1	1
E	1	0	1	1	0	1	1

(b) At program exit

Figure 3. PI_Allin1 example

To implement PI_Allin1, a program is instrumented to generate a call trace, as is done for PI. However, rather than sending the trace to SEQUITUR for compression, the trace is sent, through a Unix named pipe, to an analysis program that implements the PI_Allin1 algorithm. The benefit of this approach is that the impacts of all functions can be calculated without ever compressing or storing the trace. The drawback is that significant overhead can be incurred through the use of the pipe. However, the overhead is not likely to be as much as using SEQUITUR to compress the trace.

3.2. Using a Dynamic Compiler

A downside to using instrumentation is that significant overhead can be incurred either through using a compression algorithm, such as SEQUITUR, or through using a Unix pipe. These overheads are not directly caused by the algorithm, but rather by ancillary features.

A different approach is to use a dynamic compiler [3, 7, 9]. These systems were originally developed to perform dynamic optimization of programs using information obtained during execution. A side effect is that these systems can also be used to modify or analyze a program's instructions during execution for other purposes. This allows impact analysis to be performed during program execution without incurring the overheads mentioned above, and without requiring the program to be instrumented beforehand. Naturally, overhead is incurred, but the overhead comes solely from the impact analysis and not any external factors such as compression algorithm or operating system features.

PI_Allin1 can be easily adapted to a dynamic compiler, as it does not require the entire trace before executing. The algorithm can be applied by examining calls and returns during program execution. More details about adapting impact analysis algorithms to a dynamic compiler can be found in [6].

OnOpt [6] is another online impact analysis, that is more optimistic than PI_Allin1. By optimistic, we mean that OnOpt assumes impacts travel only along dynamic call paths. The impact for a function f is taken to be all functions that are on the call stack when f begins executing along with any function that is called either directly or indirectly by f. This is different from PI and PI_Allin1 which assume that any function called after f executes could possibly be impacted even if they were not directly, or transitively, called by f. This optimistic approach can cause OnOpt to potentially miss some impacts, but may be more useful for programs where global variables are not frequently used (such as object-oriented programs).

Algorithm: OnlineOptimistc
Input: Function calls and returns of program P
Output: Impact sets of all functions in P

Begin executing program P
While not program exit, DO
  When function f is called, DO
    If first execution of f, impact(f) = {} //init
    FORALL functions s on the call stack,
      impact(s) = impact(s) + {f} // forward
      impact(f) = impact(f) + {s} // backward
    Push f onto the call stack

  When function return occurs, DO
    Let f be function on top of call stack
    Pop call stack
End executing program

Figure 4. OnOpt impact analysis algorithm

The complete algorithm for OnOpt is given in figure 4. For the example call graph and trace in figure 1, OnOpt would report the impact set for G as {M, G} because those are the only functions on the call stack during the execution of G. Note that potential impacts may be missed here because data dependencies are not examined. For E, OnOpt gives the set {M, A, C, E}.

3.3. Impact Analysis as Software Evolves

An important topic for software engineering is the evolution of a software system and the test suite that accompanies the system. Each of the algorithms discussed here associates extra information with each test case, which is then merged together to find the impacts of the entire test suite. `PI` concatenates the call traces of the different test cases together, with each trace being identified by a unique key. The DAG formed from all the call traces can then be traversed to find the impact sets [13]. `CI` stores function coverage information related to each test case. When computing the impact of function m, the coverage information from all test cases that traversed m are unioned together. `CI` uses the resulting coverage information to calculate the impact sets. The online algorithms associate an impact file, which contains the impact sets of all functions in the program, with each test case. These sets are unioned together to find the final impact sets.

The extra information associated with each test case is updated as the program or test suite changes. If we remove or change a component of the system, we would re-execute all test cases that traverse that component. Adding a new component requires re-executing all test cases that traverse functions that call the new component. For test suite modifications, removing a test case requires removing the associated extra information. Modifying a test case requires re-executing that test case. Adding a new test case requires executing it to obtain the extra information needed by each algorithm.

In summary, all of the approaches can accommodate evolution of the software system and its associated test suite. We note, however, that the extra information required for impact analysis may not necessarily be easy to obtain or to store (see sections 4 and 5).

4. Analytic Comparison

Table 1 gives a summary of each algorithm in terms of the phases of program execution. We should also note that for `CI`, the static slicing could be performed either before or after the program execution. Comparing the algorithmic complexity of these algorithms provides some insight into the tradeoffs involved in using them. Let F be the number of functions in the application that are to be analyzed for potential impacts. T is the size of the call trace generated by those F functions. A is the maximum number of activation records that could appear on the stack at any one time (i.e., maximum depth of the call tree), and L is the number of lines in the program.

For `PI`, the program must first be instrumented and then executed. During execution, the call trace is gathered and `SEQUITUR` is used to compress the trace and form the DAG. We will neglect the time needed to collect the trace as it is much smaller than the time `SEQUITUR` needs to run. `SEQUITUR` itself is linear in the size of the call trace, $O(T)$, in both time and space to generate the grammar [15]. The generated grammar, which can be stored as a DAG, is, at worst, $O(T)$ in size, and, at best, $O(\log T)$.

Once the program and `SEQUITUR` are finished, `PI` can then run to calculate the impact of a given function by scanning forward and backward in the DAG. This process can take $O(T)$ time to complete. The algorithm, as presented in [14], only calculates the impact for one function at a time. It could, therefore, take up to $O(F * T)$ time to find the impacts for all functions.

`CI` requires very little overhead to gather the coverage information. All that is required is a bit vector for the F functions. Once the program is finished, `CI` can then calculate the impact for a given function by intersecting the set of covered functions with the set of functions from static slicing. Approximating static slicing with an interprocedural control flow graph traversal means the algorithm will take, for one function, time proportional to the number of nodes traversed which, at worst, is on the order of the number of lines in the program, $O(L)$. The worst case total time to compute the sets for all functions would, therefore, be $O(F * L)$. The time required for generating the control flow graph will also be $O(L)$ [2], which may not be insignificant. The graphs themselves will take $O(L)$ space.

`OnOpt` first requires generation of the address map, which is negligible as it only scans the symbol table stored in the binary. This mapping, with size of $O(F)$, is required because `OnOpt` works on the instruction stream level, which contains only addresses. The mapping makes the final results human readable. Binaries may contain references to more functions than are actually defined in a program, in particular, library functions. These can be added or removed from the address map as desired. Only functions found in the address map are included in the analysis.

`OnOpt` performs impact analysis for all functions during execution of the program. At every call, `OnOpt` looks up the address in the address map and performs the algorithm given in figure 4, which could take $O(A)$ time. The map is implemented as a hash table, so the lookup could, potentially, be $O(F)$. Total worst case time would therefore be $O(T * (A + F))$. In practice, the hashing is efficient and can be done in constant time. Furthermore, programs that are not heavily recursive tend to put very few calls on the stack so A can be treated as a small constant. `OnOpt` therefore requires $O(T)$ time to gather the impact information for all F functions. We stress that this is not necessarily the case for heavily recursive functions. See the discussion concerning 130.li in section 5.5.

In terms of space overhead, `OnOpt` requires memory for the impact sets themselves, the address map and the call

	Phases of Program Execution		
Algorithm	Before Execution	During Execution	After Execution
`PI` (`PathImpact`)	Instrument for call trace	Run `SEQUITUR` to produce the DAG	Analyze DAG for impact of 1 func
`CI` (`CoverageImpact`)	Instrument for function coverage	Obtain coverage information	Static slicing and intersection
`PI_Allin1` (w/ inst.)	Instrument for call trace	Perform online impact analysis	
`PI_Allin1` (under DynamoRIO)	Obtain address map	Perform online impact analysis	
`OnOpt` (`OnlineOpt`)	Obtain address map	Perform online impact analysis	

Table 1. Algorithm comparison by phases

stack. The map takes $O(F)$ space while the stack requires $O(A)$ space. Both can be neglected compared to the space needed for the impact sets, which, at worst is $O(F^2)$ assuming the pathological case of all functions being in the impact set of all other functions.

`PI_Allin1` requires space for the matrix, `BackFuncCnt`, and call stack. The matrix requires $O(F^2)$ bits, and the `BackFuncCnt` $O(F)$ integers. The call stack requires $O(A)$ space. The online version requires an additional $O(F)$ space for the address map. At every function call, `PI_Allin1` performs at most F bit operations. Thus, the time requirements are $O(T * F)$.

In summary, we can expect `CI` to add the least amount of temporal overhead to the execution of a program. We also expect `CI` to, on average, add more functions to each impact set than either `PI` or `PI_Allin1` because `CI` does not keep track of the call order among the functions.

Due to its optimistic nature, `OnOpt` is expected to deliver the smallest sized impact sets of all the algorithms. As discussed earlier, this may mean some impacts are missed. We also note that heavily recursive programs may cause `OnOpt` to perform badly, whereas the other functions should not be affected.

In terms of time, `PI`, `PI_Allin1` and `OnOpt` all scale approximately linearly with the number of function calls made. Thus, empirical experiments are needed to observe the behavior in practice.

5. Experimental Study

5.1. Experiment Definition and Context

The objective of our empirical study was to gain insight into the tradeoffs involved in using the various impact analysis algorithms in practice. This requires evaluating the spatial and temporal overheads, scalability, and the sets generated by each algorithm experimentally, in addition to the analytical comparison in section 4.

We chose several programs from the SPEC [20] application suite as subjects of analysis. These applications are small to medium sized real world applications, whose inclusion in this work is relevant because they generate large traces which allows us to examine scalability concerns. 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. All the applications considered are C programs. However, all the impact analysis algorithms discussed here can be applied to programs in any language so long as appropriate infrastructure is available.

The applications were run for a set of test cases that had already been provided. `Space` was run on 1,000 randomly generated test cases.

Program	Description	Source LOC	funcs
008.espresso	Boolean function minimizer	9,844	363
026.compress	Compression program	1,043	19
099.go	Plays the game of go	25,080	374
126.gcc	GNU C compiler (2.5.3)	131,811	2,015
130.li	Lisp interpreter	4,888	366
132.ijpeg	JPEG compressor	15,925	476
147.vortex	Object Oriented database	40,242	925
space	ESA ADL Interpreter	6,230	136

Table 2. Subjects of Analysis

5.1.1. Research Questions The research questions we sought to answer through our empirical study follow:

1. How does each algorithm, in practice, scale in terms of program execution time and space requirements?
2. How do the sets computed by each algorithm compare?

5.2. Variable Selection

The independent variable in our experiment is the impact analysis algorithm applied. In all, we implemented six different impact analyses; `PI` and `CI` are our implementations of `PathImpact` and `CoverageImpact`, respectively, `OnOpt` is the `OnlineOptimistic` algorithm, and three versions of `PI_Allin1` which differ in their inputs were studied. The three versions of `PI_Allin1` are `PI_Allin1`, which expects an uncompressed trace, `PI_Allin1_pipe` which obtains input through a Unix named pipe, and `OnPI_Allin1`, the online version with a dynamic compiler. Each of the algo-

rithms were run on each subject using the available test inputs. The dependent variables were the time overhead, measured as the difference in running times of the program with and without the analyses being performed, space overhead, measured in terms of disk storage required, and the computed impact sets.

5.3. Setup

We implemented each of the impact analysis algorithms and executed them on each subject application with the available test inputs. Where applicable, we used DynamoRIO [7] as the dynamic compiler, which allows users to write a client module to inspect running code. As the program is run, DynamoRIO creates blocks of instructions that are given to the module, which can then inspect or modify the instructions. Once the module is finished, the instructions are given to the CPU for execution.

A DynamoRIO module was written for `CI` to calculate function coverage. The control flow graphs were generated using the CodeSurfer application from GrammaTech Inc. (`http://www.grammatech.com`). We implemented `CI` to use this control flow graph and function coverage information.

For `PI`, we obtained a call trace by instrumenting the source code using SUIF [21]. The trace was compressed using `SEQUITUR` to gather timing and overhead information necessary for `PI` to run. Our implementation of `PI` follows the algorithm given in [14], except that it uses an uncompressed call trace. This has the effect of speeding up the algorithm since it does not incur overhead to parse the DAG.

`PI_Allin1` itself operates on an uncompressed trace. `PI_Allin1_pipe` uses a Unix named pipe to get the trace from the executing program. This version requires no storage or compression of the trace. The last version, `OnPI_Allin1`, is a DynamoRIO module. Finally, we implemented a DynamoRIO module for the optimistic algorithm, `OnOpt`.

It should be noted that the source code is not strictly required for any of the algorithms. Binary instrumentation tools, such as Vulcan [19] and Dyninst [8], easily allow binaries to be instrumented to gather the information required by `CI` and `PI`. The online algorithms require a dynamic compiler or a JIT, both of which are easy to obtain.

The subject programs and impact analyses were run on an Intel Pentium 4 2.0 GHz machine with 512 MB of memory running Linux. Timing information was obtained using the `time` utility. Timing overheads were calculated by running the program twice, once with the instrumentation or online module activated and once without.

Each program was run with multiple test inputs. We only report the largest test case, in terms of how many application function calls were made.

No algorithm was allowed to run for more than two hours. This affects `SEQUITUR` more than anything else as the generated execution traces could grow to gigabytes in length. The justification for this limit is that the programs under study are small to medium sized applications that run for a few minutes. Waiting hours for impact analysis results seems unreasonable.

5.4. Threats to Validity

Threats to internal validity concern our ability to draw conclusions about the different techniques and what we observe during their use. The primary issue here is the implementations used. Algorithms that we did not devise were implemented using their published descriptions. We made some modifications to speed up execution in an effort to be as fair as possible. For example, we used the uncompressed call trace for `PI` so that the implementation did not have to spend time parsing the DAG. For `CI`, we approximated slicing by performing reachability on the interprocedural control flow graph. Nonetheless, there may be other modifications that we did not consider, and were not published, that could possibly affect timing and scalability.

Threats to construct validity relate to the appropriateness of the measures. To gain an insight into the scalability of the different techniques, we measure time and disk space usage, but memory utilization, which is more difficult to reliably measure due to sensitivity of a particular implementation, may also be an important factor to consider in choosing an appropriate impact analysis algorithm.

Threats to external validity involve generalization of our results. We have chosen C applications that vary in size from 1,000 lines of code to 131,000 lines. 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.

5.5. Data and Analysis

Table 3 gives the total time needed to run each algorithm on the applications for the input set that produces the largest trace. For the online algorithms, this is the overhead incurred during the execution of the program. For the dynamic algorithms, `CI` and `PI`, this is the execution overhead combined with the post execution analysis. The breakdown between these two phases is also shown.

In two cases, 130.li and 147.vortex, the applications generated a trace larger than 15 GB, which was beyond the available disk space on the machine, which meant the algorithms requiring the trace, `PI` and `PI_Allin1`, could not be run. In the case of 126.gcc, instrumentation could not be added due to problems encountered using SUIF. Interprocedural control flow graphs also could not be generated for

	CI			PI			PI_Allin1	PI_Allin1_pipe	OnPI_Allin1	OnOpt
	Exec. Over.	Post Exec.	Total	Exec. Over.	Post Exec.	Total				
008.espresso	1s	16s	17s	2m 46s	1h 14m	1h 16m	20s	24s	4s	2s
026.compress	$<$ 1s	$<$ 1s	$<$ 1s	7s	15s	22s	2s	1s	2s	1s
099.go	2m	1.2s	2m 1s	$>$ 2h	$>$ 2h	$>$ 2h	24m	27m	4m 12s	10m
126.gcc	NA	NA	NA	NA	NA	NA	NA	NA	24s	40s
130.li	3m	10s	3m 10s	$>$ 2h	NA	NA	NA	$>$ 2h	14m	$>$ 2h
132.ijpeg	6s	3s	9s	31m	$>$ 2h	$>$ 2h	3m 42s	4m 30s	27s	1m 8s
147.vortex	3m	5m	8m	$>$ 2h	NA	NA	NA	$>$ 2h	14m	55m
space	$<$ 1s	1s	1s	$<$ 1s	$<$ 1s	$<$ 1s	$<$1s	$<$ 1s	1s	$<$ 1s

Table 3. Timing comparisons

this application. Note that, despite these problems in obtaining traces, and graphs, OnPI_Allin1 always succeeded in obtaining results.

Table 4 shows the disk space requirements for our implementations of the dynamic algorithms. CI requires space for storing the interprocedural control flow graphs (CFGs) while PI requires space for storing the DAG. For reference, the size of the uncompressed trace is also listed, but we stress that this is not needed for PI as the compression can be done online. The uncompressed traces can grow quite large with two of the programs exceeding the 15 GB filesystem size. In cases where SEQUITUR was able to finish within 2 hours, it did an excellent job of compressing the traces down to more manageable sizes.

Table 4 also provides a sense of scalability in the benchmarks. In all the algorithms, except CI, the time required was related to the number of function calls made during program execution, not the number of lines of code. This becomes evident in the case of space, which had 6,000 lines of code, but generates a call trace of only 2KB. 130.li, which has few lines of code, generates a much larger call trace. It is the size of generated traces that determines scalability, and not necessarily the number of lines of code.

As expected, and can be seen in table 3, CI adds very little execution overhead. The overhead for PI, which is almost entirely due to SEQUITUR, is quite high and unable to complete within 2 hours for several of the applications. These findings are generally consistent with the timing overhead reported in [17], which examined smaller applications.

Regardless of the execution overhead, PI does not scale well to larger applications, as can be seen in table 3. The post execution time listed in that table is solely due to running the analysis programs. In the case of 099.go, which generated a call trace of 3.5 GB, PI could not complete within 2 hours. PI could not be run on 130.li and 147.vortex because the generated traces were larger than the available disk space (15GB). On the basis of the experience with 099.go, we feel it is likely that PI would not have completed within 2 hours on these 2 applications either.

CI appears to scale very well to larger programs. CI's time, unlike the other algorithms, is based on the number of functions and not how many function calls are made during program execution, which is very noticeable in table 3.

The execution time for the online algorithms are also given in table 3. In general, PI_Allin1 performed better than PI_Allin1_pipe. Since both perform the same algorithm, the differences in times are due to the overhead from using a Unix named pipe. Note that an advantage of using the pipes is that PI_Allin1_pipe could still be used, even though a trace could not be obtained due to disk space limitations, as was the case for 130.li and 147.vortex. In these cases, however, the overhead from the pipe caused PI_Allin1_pipe to run longer than 2 hours.

OnPI_Allin1, the version of PI_Allin1 that runs under DynamoRIO, performed well for all the applications, especially the larger programs. This provides an important motivation for using dynamic compilers; results can be obtained in a much more scalable fashion than other online techniques such as using named pipes. Overhead is solely incurred by the algorithm and not external factors such as reading disk files, or pipes.

Finally, OnOpt, the optimistic algorithm running under DynamoRIO, also seems to perform well in most cases, though not nearly as well as OnPI_Allin1. OnOpt performed poorly for 130.li, a lisp interpreter, which is expected as 130.li is heavily recursive causing OnOpt to spend too much time scanning the call stack.

	Trace Sz.	DAG Sz.	CFG Sz.
008.espresso	58MB	331KB	2.3 MB
026.compress	2.8MB	475B	124 KB
099.go	3.5GB	NA	2.8 MB
130.li	$>$15GB	NA	1.6 MB
132.ijpeg	672MB	1.2 MB	2.5 MB
147.vortex	$>$15GB	NA	7.6 MB
space	2.2KB	396B	844 KB

Table 4. Disk space requirements

We performed a close examination of the impact sets for 008.espresso, which was the largest program that all anal-

ysis algorithms were able to process. Out of the 363 functions found in the program, the test case covered 186 of the functions.

For all of the functions, PI, and all the PI_Allin1 algorithms computed the same impact sets. We expect this to be true in the general case, although PI and PI_Allin1 may differ in the event of abnormal program termination or exceptions. For 172 functions, CI reported, on average, 30 more functions in their impact sets than PI computed. This is an example of the imprecision that is possible with CI. For 10 functions, CI did compute sets that were smaller than the corresponding sets computed by PI, but upon further investigation, we found that the difference was due to one function that was called solely through function pointers. The implementation of CI could not resolve the call, and thus missed the impact. PI and PI_Allin1 were able to disambiguate the call from the runtime information.

As expected, OnOpt reported much fewer elements in each set due to its optimistic approach. OnOpt computed smaller sets for all functions, except for main. On average the sets were smaller by 128 functions. However, the optimistic approach may be missing impacts.

In answer to our research questions, we find that OnPI_Allin1 scales the best as programs increase the number of function calls made, while maintaining precision. PI, as currently described, takes too long to perform the analysis. It is also clear that CI adds more functions to the impact sets, and thus, tends to be less precise than the PI algorithms. OnOpt, which is expensive when the program being studied is heavily recursive, produces the smallest impact sets, but may also be missing impacts.

Finally, table 5 gives a brief summary of the qualitative aspects of each algorithm discussed. Most of the values in the table were discussed earlier in this paper. We note that the values for 'potentially missed impacts' are speculative. It is not immediately clear how each algorithm would respond in the presence of exceptions.

6. Related Work

Impact analysis can be performed on a software system using various approaches. The most common automated techniques, other than the ones we studied, are transitive closure on call graphs [5], static slicing [22, 24], and dynamic slicing [1, 11]. All these techniques have advantages and disadvantages. Namely, transitive closure on call graphs might be inexpensive, but it can be highly inaccurate. Static slicing can perform safe analysis, but may yield impact sets that are too large to be used in practice. Dynamic slicing can produce impact sets of reasonable size, related to specific execution profiles, but it does not guarantee safety. Similar to static slicing, dynamic slicing is computationally expensive (thus not scalable) relative to call graph based approaches due to the data and control dependence level of analysis.

To our knowledge, this is one of the first uses of a system like DynamoRIO to involve online analysis for software engineering applications. DynamoRIO has been used previously for adaptive optimization and dynamic monitoring for security enforcement. Bruening et al. [7] provide an interface for constructing addons to the DynamoRIO system that can be used to manipulate programs by performing actions such as optimization, instrumentation, profiling, dynamic translation, etc. Kiriansky et al. [10] have implemented a form of control flow monitoring that they call program shepherding, for enforcing security policies.

7. Conclusions and Future Work

We have described a comparative study of several dynamic and online impact analysis algorithms. The dynamic algorithms, PI and CI, collect data during program execution and then process that data after the program is finished. In contrast, the online algorithms, PI_Allin1, and OnOpt, perform all analysis as the program is executing. Our results indicate that performing impact analysis online, using a dynamic compiler, scales better than current dynamic impact analysis algorithms. In particular, we find the online version of PI_Allin1, which has the same motivation as PI and produces the same sets under normal program termination, to perform the best, while still being safe relative to the program's execution.

Based on the added scalability of online dynamic impact analysis, we plan to investigate ways to increase the precision of online analyses by exploiting the analyses already made available by dynamic compilers for dynamic optimization. We are also exploring other program analyses for software engineering that may benefit from exploiting dynamic compilation technology.

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation*, June 1990.

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

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Object Oriented Programming, Systems, Languages and Applications*, 2000.

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

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

	CI	PI	PI_Allin1 (w/ inst.)	OnPI_Allin1	OnOpt
instrumentation needed	function coverage	function entry/exit	function entry/exit	none	none
infrastructure required	instrumentation, CFG generator	instrumentation string compression	instrumentation	dynamic compiler	dynamic compiler
static analysis required	ICFG slicing or reachability	none	none	none	none
analysis time	low	high (SEQUITUR)	high (due to using pipe)	medium	medium to high
spatial overhead	low to medium (CFG)	high (DAG)	low	low (address map)	low (address map)
potentially missed impacts	exceptions, function pointers	exceptions	exceptions	exceptions	exceptions, global variables

Table 5. Algorithm summary

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

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, 2003.

[8] B. Buck and J. K. Hollingsworth. An API fro runtime code patching. *Journal of Supercomputing Applications and High Performance Computing*, 2000.

[9] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: a new run-time control point. *MICRO*, 2002.

[10] V. Kiriansky, D. Bruening, , and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[11] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29, 1988.

[12] J. R. Larus. Whole program paths. In *Programming Language Design and Implementation*, 1999.

[13] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. *International Symposium on Software Reliability Engineering*, 2003.

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

[15] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference*, 1997.

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

[17] 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.

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

[19] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binaray transformation in a distributed environment. Technical report, Microsoft Research, 2001.

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

[21] SUIF Group. The SUIF compiler system. http://suif.stanford.edu.

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

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

[24] M. Weiser. Program slicing. In *International Conference on Software Engineering*, 1981.