Online Impact Analysis via Dynamic Compilation Technology

B. Breech, A. Danalis, Stacey Shindo, and Lori Pollock
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{breech, danalis, shindo, pollock}@cis.udel.edu

Abstract

Dynamic impact analysis based on whole path profiling of method calls and returns has been shown to provide more useful predictions of software change impacts than method-level static slicing and to avoid the overhead of expensive dependency analysis needed for dynamic slicing-based impact analysis. This paper presents the design, implementation, and evaluation of an online approach to dynamic impact analysis as an extension to the DynamoRIO binary code modification system and to the Jikes Research Virtual Machine. Storage and postmortem analysis of program traces, even compressed, are avoided.

1. Introduction

Software change impact analysis has the goal of determining the potential impacts on a software system resulting from a program change [2]. Predictive impact analysis seeks to predict the impact of change without making the change, thus allowing for software maintainers to make decisions with some planning and control over the impacts of their decisions. Because *static* impact analysis techniques such as transitive closure of a call graph [2] and static slicing [12], make predictions based on an analysis of the program text, they can be highly inaccurate relative to the expected operational profile of the application. Static approaches take into account all possible program inputs and behaviors, and some information, due to aliasing and polymorphism, is unknown until runtime [11]. In contrast, *dynamic* impact analysis [8, 10] predicts impact relative to a particular set of program execution behaviors by analyzing runtime information on a specified set of inputs. While impact results can be more accurate to a given operational profile, they are not safe in terms of their assessment of the impact in general.

When the safety of the impact sets is not critical to their use, the smaller sets resulting from dynamic analyses can be of more practical use to maintainers than sets generated by static techniques. By reporting impact sets specific to a particular test suite, the tester can possibly reduce the number of test cases selected to be rerun during regression testing or utilize the impact information to prioritize test cases according to the expected usage of the application [11].

Dynamic counterparts to the static techniques are dynamic slicing [1] and whole program path profiling of function calls and returns [8, 7]. Slicing provides more accurate impact sets than call graph based techniques due to the fine grain dependency information. Dynamic slicing shrinks the size of the impact sets relative to static slicing. Whole path profiling at the call graph level eliminates the need for control and data dependence computation, requires instrumentation only at the function call level, and simplifies the maintainer's specification of the change target.

Law and Rothermel [8] developed and evaluated a dynamic impact analysis technique based on whole path profiling. The program is instrumented at each method entry and exit, then the instrumented program is executed, generating a whole program trace. For space reasons, the SEQUITUR data compression algorithm [9] is applied to a set of one or more concatenated program traces; the output is the compressed trace called a SEQUITUR grammar, stored in the form of a whole path DAG representation [6]. Their impact analysis algorithm is applied to this DAG representation to estimate the impact of a given function. The approach does not rely on program source code for call graph construction or dependence analysis. Their experiments with a 6200-line C code from the European Space Agency showed that their whole program path-based dynamic impact analysis can provide more useful predictions of change impact than function-level static slicing when a software maintainer is interested in impacts related to a specific operational profile.

In this paper, we present an approach to whole program path-based dynamic impact analysis which is performed completely online (i.e., during program execution), thus alleviating the need to produce and then analyze a whole program path trace representation after instrumented execution. This is achieved by exploiting the technology of current *dy-*

namic compilers, which perform some compilation tasks, such as optimization, at runtime. Additionally, our method calculates the impact sets for all application functions called during execution. We demonstrate the general applicability of our approach by implementing it in two such compilers; DynamoRIO, a dynamic code modification system for native binaries [4], and the IBM Jikes Research Virtual Machine (RVM) [5], which can compile Java bytecode to native machine code at runtime. We have computed change impact sets for several medium-sized Java, C, Fortran, and C++ applications. In both systems, the space and analysis of a (compressed or uncompressed) whole program path trace representation of a postmortem dynamic analysis is avoided.

2. Online Impact Analysis

The heuristic used for dynamic path-based impact analysis is to assume that a changed function *f*'s impact will propagate down any and only dynamic paths that have been observed to pass through *f* [8]. Thus, any function that is directly or transitively called by *f*, and any function which is on the call stack after *f* returns, is included in the impact set for *f*.

In figure 1, the impact set computed by transitive closure on the static call graph for function B would be {B, C, D, E} regardless of what B actually calls. Extending transitive closure to include any nodes that reach function B along any path from Main would include Main in the set as well. The impact set for function C would be reported by transitive closure as {C, D, E} whereas extended transitive closure would report {Main, A, B, C, D, E} regardless of how C may actually be called. Based on a particular operational profile in which B called C which only called D, dynamic impact analysis would report the impact set for function B as {Main, B, C, D} since B calls C directly and then transitively calls D at runtime, and Main is still on the call stack after B returns. The impact set for C, under the same operational profile, reported by dynamic impact analysis would be {Main, B, C, D}, in contrast to extended transitive closure which would include A in C's impact set. Thus, more accurate impact sets for a given operational profile can be obtained because the path profiling captures the calls that actually occur.

Our general algorithm for computing the dynamic impact sets online is presented in figure 2. The key insight behind the algorithm is that any function on the call stack when f is called is a function that could be potentially impacted by the return of f, thus these functions are all added to f's impact set; this achieves the effect of the backward search through a trace representation during postmortem analysis. Any function that is added to the stack after s is called, but before s returns, could be potentially impacted

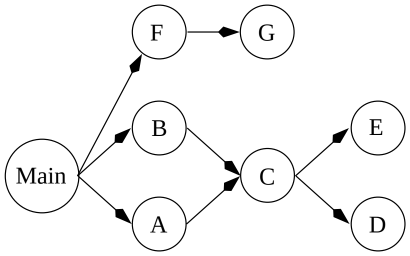

Figure 1. Example call graph

```
Begin executing program
As each call to a function, $f$, starts execution, DO
    If first exec 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

As $f$ completes execution and returns, DO
    POP $f$ off the call stack
End executing program
```

Figure 2. Online impact analysis algorithm

by s. Thus, as a function f is added to the stack, f is added to the impact set of each function, s, already on the call stack. This is comparable to the forward search during postmortem analysis of a program trace representation.

The equivalence of the impact sets computed by Law and Rothermel's analysis [8] and our online analysis, under the assumption of program execution without exceptions or abnormal exits, can be informally shown by relating how functions are added to impact sets in each approach [3]. Our approach makes use of the call stack directly to identify the same sets, adding functions upon executed calls. In the event of abnormal function returns, the two methods may differ.

In both approaches, more accurate impact sets could be constructed with more fine grained analysis, such as pointer analysis. However, these analyses require the source code of all components, which may not be feasible, or pointer analysis with missing components, which may not be worth the added analysis costs.

We have implemented our general online algorithm in two different systems: a binary code modification system for compiled native binaries from languages like C/C++ and Fortran, and a Java virtual machine.

Impact Analysis in the DynamoRIO System. DynamoRIO is a dynamic code modification system, which allows users to write client modules to inspect a running program to optimize, instrument or collect runtime information [4]. Upon startup, DynamoRIO initializes itself and calls the client module's initialization routine. DynamoRIO then enters a loop constructing and executing instruction sequences from the binary code. These sequences are basic blocks, which are consecutively executed instructions that end with a control transfer instruction, such as a conditional or unconditional jump. After each block is constructed, it is passed to the client module which can inspect or change the block. DynamoRIO then passes the block to the CPU to be natively executed.

We have implemented a module that runs under the DynamoRIO system as a separate entity, performing dynamic impact analysis online. This approach requires no modification of the program binary nor of DynamoRIO itself. Before our module can run, we must first obtain a mapping between function addresses and function names. Obtaining the mapping is the only language and compiler specific part of our system.

The initialization phase of our online analysis module reads the address map and initializes the impact sets for each function and a simplified call stack to keep track of calls. When called during basic block construction, our module inspects each basic block created by DynamoRIO looking for any function calls, performing the online impact analysis on them.

Impact Analysis in the Jikes RVM. The Jikes Research Virtual Machine (RVM) [5] is an open source Java Virtual Machine written in Java. The Jikes RVM takes a compile-only approach to method execution, where all methods are compiled from bytecode to native code before they are executed. Jikes RVM features a *baseline* compiler and an *optimizing* compiler. The baseline compiler performs quick bytecode to native translation, while the optimizing compiler performs classical optimizations and adaptive optimizations.

Jikes RVM baseline compiler translates a method from bytecode to native code when the method is called for the first time. Methods that are not called at run time will not be compiled. Our work makes use of the baseline compiler as the optimizing compiler may choose to inline methods, which could have a significant effect on the impact sets.

We implemented the dynamic analysis in Jikes RVM by slightly extending the runtime system and the baseline compiler. We modified the baseline compiler to add callbacks to our online analysis module in each method prologue as the methods are compiled. The impact analysis module was integrated into the runtime system where we can easily access the call stack maintained by Jikes RVM to perform the algorithm in figure 2.

3. Experimental Evaluation

The questions we sought to answer were: Can online dynamic impact analysis achieve the same accuracy as offline processing of the call trace? In practice, what are the space and time costs for online dynamic impact analysis, in a dynamic binary translation system and a Java Virtual Machine environment?

The applications executed under DynamoRIO were run on an 2.4 GHz Pentium 4 machine with 1.5 GB memory; the Java applications were run on a dual processor Xeon 2.40 GHz machine with 1 GB memory.

Table 1 shows space characteristics of the applications used for our experiments under DynamoRIO and Jikes RVM. The Java applications are all from the SPECjvm98 suite, while the compiled applications were taken from SPEC92, SPEC95 and SPEC2000. The SPEC applications are a variety of real world programs slightly modified to produce performance information. The native binary applications were largely written in C, although 090.hydro2d is a Fortran 77 code and 252.eon is a C++ code. They were compiled with version 3.2.2 of gcc, g77, and g++, respectively. Source LOC gives the number of uncommented, non-blank lines of source code in the application. The next column reports the number of functions present in the binary. For the native binary applications this is the number of functions in the address map, which may be more than what the program actually defines because the compiler may insert extra functions. The memory footprint was derived by running the applications under DynamoRIO with our module loaded and again without the module. The memory sizes for both executions, obtained from the *top* utility, were subtracted to obtain the memory overhead due to online impact analysis. The last two columns give the size of the impact file and the size of the compressed call trace.

The call trace compression was performed using the SEQUITUR algorithm [9]. Larus has reported that a 2GB trace was reduced to 100 MB [6]. In our case, the call traces often display repetitions that SEQUITUR can exploit. The sizes of the uncompressed traces ranged from 2.2 MB to over 10 GB while the compressed traces ranged from 2.1 KB to 8 MB.

In almost all cases, the impact file size is smaller than the compressed call trace, sometimes significantly smaller, as in the case of 099.go on DynamoRIO and 202.jess on Jikes RVM. This file is human readable and requires no more processing time, but, if desired, could be compressed to save additional space. While the impact file is the key space requirement for online dynamic impact analysis, any offline analysis would require the compressed trace.

The numbers for the memory footprint within DynamoRIO show the average memory overhead per function is small. No numbers are available for the Java applications because of the difficulty in getting accurate numbers due to garbage collection. The overhead, though, should mimic that of the DynamoRIO applications and be only a few kilobytes per method.

We measured the time costs for online impact analysis with respect to the specific base systems (Table 2). The second column gives the number of functions executed while the program was running. These numbers include only those functions mentioned in the address map for the DynamoRIO applications, and the methods defined in the bytecode for the Java applications. No library or system calls are included. The next two columns show how long the applications ran without our analysis and with our analysis performed, respectively. These two execution times were then subtracted and divided by the number of functions executed to derive the average cost of execution.

The data shows that, for most programs, the time for online impact analysis scales roughly linear with the total number of function calls. This is expected, though not proven here due to space limitations [3]. The execution time of the native binary programs is 1-2 μ seconds per function call; the Java applications are usually 30-60 μ seconds per function call.

In DynamoRIO, the exception to the 1-2 μ seconds per function call behavior is 022.li, which is heavily recursive. This recursion causes the call stack to grow much larger than in the other applications and hence more time is spent per call. In particular, the call stack for 022.li had a maximum of over 17,000 activation records while the next highest was 126.gcc which only had 135. All the other native binaries put fewer than 30 calls on the stack at their maximum.

The impact sets from our online analysis and those calculated from the call traces are the same as there was no program with abnormal function returns.

The impact sets can be computed for any number of multiple runs of the same program depending on the tool user's desired set of inputs to be used for inferring the dynamic behavior of the application. Since the goals of our experimental study are irrelevant to the number of runs, we did not unnecessarily execute multiple runs with different inputs.

Each execution gives the impact sets for all called functions in the application. If we only wanted the impact set of one or two functions, then the cost, in terms of execution time, should dramatically decrease (at least for functions other than `main`). The tradeoff, however, would be that to obtain the impact sets of any other function would require another run of the program.

We have presented an approach to online dynamic impact analysis by exploiting dynamic compiler technology. Our analysis computes accurate, course-grained change impact sets of all functions in a given running program without instrumenting the binary, while making minimum or no changes to the compiler.

We are investigating how to further reduce the time for our online analysis, especially for heavily recursive programs. We also plan to examine methods of obtaining more accurate impact sets.

References

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

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

[3] B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. Technical Report 2004-16, University of Delaware, http://www.cis.udel.edu/ hiper/hiperspace, 2004.

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

[5] D. Grove and M. Hind. The design and implementation of the Jikes RVM optimizing compiler. Object-Oriented Programming, Systems, Languages and Applications Tutorial, 2002.

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

[7] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. *IEEE Int. Symp. on Soft. Reliability Eng.*, 2003.

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

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

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

[11] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. *Program Analysis for Software Tools and Engineering*, 2001.

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

Program	Description	Source LOC	No. of Methods	Memory Footprint per func (kb)	Impact File Size (kb)	Compressed Trace Size (kb)
DynamoRIO Applications						
008.espresso	Boolean function minimization	9,844	363	3.5	64	307
022.li	Lisp interpreter	4,742	359	6.5	65	153
026.compress	Compression program	1,043	19	6.3	0.9	2.1
090.hydro2d[a]	Navier-Stokes equation solver	1,708	67	5.8	6.9	3.9
099.go	Plays the game of go	25,080	374	1.8	115	2,662
126.gcc	GNU C compiler (2.5.3)	131,811	2,015	4.4	1,026	8,500
132.ijpeg	JPEG compressor	15,925	476	4.5	102	680
147.vortex	Object Oriented database program	40,242	925	5.0	789	846
252.eon[b]	Probabilistic ray tracer	22,115	1,732	4.7	567	970
Jikes RVM Applications						
201.compress	Compression program	514	29	NA	5.5	–
202.jess	Expert system	6,123	589	NA	178	2,069
209.db	Database	644	29	NA	5	8.3
222.mpeg	MP3 file decoder	N/A[c]	221	NA	97	–
228.jack	Parser generator	N/A[c]	276	NA	194	305

[a] Fortran application

[b] C++ application

– represent cases where we are still trying to successfully compress with SEQUITUR.

[c] Derived from commercial applications; source code not available

Program	No. of Executed Appl. Calls	Base System (s)	Base with Analysis (s)	Ave. Cost of Execution (μs/func)
DynamoRIO Applications				
008.espresso	1,684,604	0.6	3.4	1.6
022.li	1,386,734	0.2	1,080	780
026.compress	250,913	0.2	2.2	7.9
090.hydro2d	51,375,647	15.7	56.4	0.8
099.go	6,127,021	1.8	12.7	1.8
126.gcc	19,931,499	5.3	48	2.1
132.ijpeg	2,499,863	1.4	5.4	1.6
147.vortex	53,329,226	4.1	124	2.2
252.eon	627,842,194	12.7	1,015	1.6
Jikes RVM Applications				
201.compress	225,925,745	22	6,944	30.6
202.jess	86,442,355	32	14,360	166
209.db	1,468,495	47	94	32
222.mpeg	108,313,227	22	6,944	63.9
228.jack	5,757,190	26	396	64.3