A Survey of Dynamic Program Analysis Techniques and Tools

Anjana Gosain and Ganga Sharma

University School of Information and Communication Technology,
Guru Gobind Singh Indraprastha University, New Delhi-110078, India
anjana_gosain@hotmail.com, ganga.negi@gmail.com

Abstract. Dynamic program analysis is a very popular technique for analysis of computer programs. It analyses the properties of a program while it is executing. Dynamic analysis has been found to be more precise than static analysis in handling run-time features like dynamic binding, polymorphism, threads etc. Therefore much emphasis is now being given on dynamic analysis of programs (instead of static analysis) involving the above mentioned features. Various techniques have been devised over the past several years for the dynamic analysis of programs. This paper provides an overview of the existing techniques and tools for the dynamic analysis of programs. Further, the paper compares these techniques for their merits and demerits and emphasizes the importance of each technique.

Keywords: dynamic analysis, static analysis, instrumentation, profiling, AOP.

1 Introduction

Analysing the dynamic behaviour of a software is invaluable for software developers because it helps in understanding the software well. Static analysis has long been used for analysing the dynamic behavior of programs because it is simple and does not require running the program [6], [24]. Dynamic analysis, on the other hand, is the analysis of the properties of a running program [1]. It involves the investigation of the properties of a program using information gathered at run-time. Deployment of software now-a-days as a collection of dynamically linked libraries is rendering static analysis imprecise [32]. Moreover, the widespread use of object oriented languages, especially Java, to write software has lead to the usage of run-time features like dynamic binding, polymorphism, threads etc. Static analysis is found to be ineffective in these kinds of dynamic environments. Whereas static analysis is restricted in analyzing a program effectively and efficiently, and may have trouble in discovering all dependencies present in the program, dynamic analysis has the benefit of examining the concrete domain of program execution [1]. Therefore dynamic analysis is gaining much importance for the analysis of programs.

Dynamic and static analysis are regarded as complementary approaches and have been compared for their merits and demerits[17], [18]. Some of the major differences of dynamic analysis with static analysis are listed in Table 1. The main advantage of

S.C. Satapathy et al. (eds.), *Proc. of the 3rd Int. Conf. on Front. of Intell. Comput. (FICTA) 2014 – Vol. 1*, Advances in Intelligent Systems and Computing 327, DOI: 10.1007/978-3-319-11933-5_13

dynamic analysis over static analysis is that it can examine the actual, exact run-time behaviour of the program and is very precise. On the contrary, the main disadvantage of dynamic analysis is that it depends on input stimuli and therefore cannot be generalized for all executions. Nevertheless, dynamic analysis techniques are proving useful for the analysis of programs and are being widely used. Efforts are also being made to combine dynamic and static analysis to get benefit from the best features of both. For example, static and dynamic analysis capability has been provided in a framework called CHORD [33], a dynamic analysis tool is used as an annotation assistant for a static tool in [34] etc.

Table 1. Comparison of Dynamic analysis with Static Analysis

Dynamic Analysis	Static Analysis
Requires program to be executed	Does not require program to be executed
More precise	Less precise
Holds for a particular execution	Holds for all the executions
Best suited to handle run-time programming language features like polymorphism, dynamic binding, threads etc.	Lacks in handling run-time programming language features.
Incurs large run-time overheads	Incurs less overheads

The remainder of this paper is structured as follows. Firstly, the main techniques of dynamic analysis are described followed by a comparison of each technique. Then, a description of the most widely used dynamic analysis tools is given. Finally, in the last section, we describe the main conclusions and the future work.

2 Dynamic Analysis Techniques

Dynamic analysis techniques reason over the run-time behavior of systems[17].In general, dynamic analysis involves recording of a program's dynamic state. This dynamic state is also called as profile/trace. A program profile measures occurrences of events during program execution[2]. The measured event is the execution of a local portion of program like lines of code, basic blocks, control edges, routines etc. Generally, a dynamic analysis technique involves the following phases: 1) program instrumentation and profile/trace generation, 2) analysis or monitoring [27]. Program instrumentation is the process of inserting additional statements into the program for the purpose of generating traces. These instrumented statements are executed at the same time when the program is running. The fundamental challenge for success of dynamic analysis lies in the creation of instrumentation and profiling infrastructures that enable the efficient collection of run-time information [32]. Depending upon the instrumentation provided, required information from one or more executions would be gathered for analysis purpose. The actual analysis or monitoring phase takes place on these traces. This phase is additionally augmented to handle violations of properties. The analysis or monitoring can be performed either offline or online.

An analysis is said to be online if the target system and the monitoring system are run in parallel. But it may increase the cost. Choosing between the two, one should consider whether the purpose of the analysis is to find error or to find and correct errors. In the first case, offline analysis should be performed; while the latter should go for the online analysis.

In the next sub-sections, we will study the main techniques for dynamic analysis of programs. We have omitted the dynamic analysis of a system based on models (like UML) because we are providing only those techniques which work on source code itself or some form of source code like binaries or bytecode. The section ends with a comparison of these techniques in Table 2.

2.1 Instrumentation Based

In this technique, a code instrumenter is used as a pre-processor to insert instrumentation code into the target program[27]. This instrumentation code can be added at any stage of compilation process. Basically it is done at three stages: source code, binary code and bytecode. Source code instrumentation adds instrumentation code before the program is compiled using source-to-source transformation. It can use some meta-programming frameworks like Proteus[47], DMS[3] etc. to insert the extra code automatically. These meta-programming frameworks provide a programming language that can be used to define context sensitive modifications to the source code. Transformation programs are compiled into applications that perform rewriting and instrumentation of source, which is given as input. Applying instrumentation to source code makes it easier to align trace functionality with higher-level, domain-specific abstractions, which minimizes instrumentation because the placement of additional code is limited to only what is necessary. The disadvantage is that it is target language dependent and can also become problematic when dealing with specific language characteristics, such as C/C++ preprocessing and syntactic variations.

Binary instrumentation adds instrumentation code by modifying or re-writing compiled code. This is done using specially designed tools either statically or dynamically. Static binary instrumentation involves the use of a set of libraries and APIs that enable users to quickly write applications that perform binary re-writing. Examples of such tools are EEL[28], ATOM[9] etc. Dynamic binary instrumentation (implemented as Just-In-Time compilers) is performed after the program has been loaded into memory and just prior to execution using tools like MDL[22], DynInst[43] etc. MDL is a specialized language that has two key roles. First, it specifies the code that will be inserted into the application program to calculate the value of performance metrics. This code includes simple control and data operations, plus the ability to instantiate and control real and virtual timers. Second, it specifies how the instrumentation code is inserted into the application program. This specification includes the points in the application program that are used to place the instrumentation code. DynInst[43] is an API that permits the insertion of code into a running program. Using this API, a program can attach to a running program, create a new bit of code and insert it into the program. The program being modified is able to continue execution and doesn't need to be re-compiled, re-linked, or even re-started[43]. Dynamic binary instrumentation

has the advantage over its static counterpart in that profiling functionality can be selectively added or removed from the program without the need to recompile.

Bytecode instrumentation performs tracing within the compiled code. Again, it can be static or dynamic. Static instrumentation involves changing the compiled code offline before execution i.e., creating a copy of the instrumented intermediate code using high level bytecode engineering libraries like BCEL[11], ASM[8], Javaassist[10] etc. BCEL and ASM allow programmers to analyze, create, and manipulate Java class files by means of a low-level API. Javassist[10], on the other hand, enables structural reflection and can be used both at load-time or compile-time to transform Java classes. The disadvantage here is that dynamically generated or loaded code cannot be instrumented. Dynamic instrumentation, on the other hand, works when the application is already running. There exist various tools for dynamic bytecode instrumentation like BIT[29], IBM's Jikes Bytecode Toolkit[23] etc. There also exist tools which can provide both static and dynamic bytecode instrumentation. For example, FERRARI[4] instruments the core classes in a Java program statically, and then uses an instrumentation agent to dynamically instrument all other classes. This has the advantage that no class is left without instrumentation. Nevertheless, bytecode instrumentation is harder to implement, but gives unlimited freedom to record any event in the application.

2.2 VM Profiling Based

In this technique, dynamic analysis is carried out using the profiling and debugging mechanism provided by the particular virtual machine. Examples include Microsoft CLR Profiler[20] for .NET frameworks and JPDA for Java SDK. These profilers give an insight into the inner operations of a program, specifically related to memory and heap usage. One uses plug-ins (implemented as dynamic link libraries) to the VM to capture the profiling information. These plug-ins are called as profiling agents and are implemented in native code. These plug-ins access the profiling services of the virtual machine through an interface. For example, JVMTI[44] is the interface provided by JPDA. It is straightforward to develop profilers based on VM because profiler developers need only implement an interface provided by the VM and need not worry about the complications that can arise due to interfering with the running application. Benchmarks like SpecJVM [13] etc. are then used for actual run-time analysis. A benchmark acts like a black-box test for a program even if its source code is available to us. The process of benchmarking involves executing or simulating the behavior of the program while collecting data reflecting its performance. This technique has the advantage that it is simple and easier to master. One of the major drawbacks of this technique is that it incurs high run-time overheads[4], [39].

2.3 Aspect Oriented Programming

Aspect-oriented programming (AOP)[26] is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common

concerns. With AOP, there is no need to add instrumentation code as the instrumentation facility is provided within the programming language by the built-in constructs. AOP adds the following constructs to a program : aspects, join-point, point-cuts and advices. Aspects are like classes in C++ or Java. A join-point is any well defined point in a program flow. Point-cuts pick join-points and values at those points. An advice is a piece of code which is executed when a join-point is reached. Aspects specify point-cuts to intercept join-points. An aspect weaver is then used to modify the code of an application to execute advice at intercepted join-points [21]. The AOP paradigm makes it easier for developers to insert profiling to an existing application by defining a profiler aspect consisting of point-cuts and advice. Most popular languages like C++ and Java have their aspect oriented extensions namely AspectC++ [42] and AspectJ [25] respectively. There also exist frameworks that use AOP to support static, load-time, and dynamic (runtime) instrumentation of bytecode [7].

Some problems encountered by AOP approaches are the design and deployment overhead of using the framework[14]. AOP frameworks are generally extensive and contain a variety of configuration and deployment options, which may take time to master. Moreover, developers must also master another framework on top of the actual application, which may make it hard to use profiling extensively. Another potential drawback is that profiling can only occur at the join-points provided by the framework, which is often restricted to the methods of each class, i.e., before a method is called or after a method returns.

Table 2. Comparison of Dynamic Analysis Techniques

	Dynamic Analysis Technique			
	Instrumentation Based		VM Profiling Based	AOP Based
	Static	Dynamic		
Level of Abstraction	Instruction/Bytecode	Instruction/bytecode	Bytecode	Programming Language
Overhead	Runtime	Runtime	Runtime	Design and deployment
Implementation Complexity	Comparatively Low	High	High	Low
User Expertise	Low	High	Low	High
Re-compilation	Required	Not Required	Not Required	Required

Application-specific events occurring within a method call therefore cannot be profiled, which means that non-deterministic events cannot be captured by AOP profilers[21]. Still, AOP is getting popular to build dynamic analysis tools as it can be used to raise the abstraction level of code instrumentation and incurs less runtime overhead [18].

3 Dynamic Analysis Tools

Dynamic analysis tools have been widely used for memory analysis [35], [38], [36], [40], [30], [37], [12], invariant detection [16], [19], deadlock and race detection[35], [40], [6] and metric computation[15], [41]. These tools are being used by companies for their benefits. For example, Pin[40] is a tool which provides the underlying infrastructure for commercial products like Intel Parallel Studio suite[49] of performance analysis tools. A summary of dynamic analysis tools is provided in Table 3.

Table 3. Dynamic Analysis Tools

Technique	Tool	Language	Type of Dynamic Analysis done								
			Cache Modelling	Heap Allocation	Buffer Overflow	Memory Leak	Deadlock Detection	Race Detection	Object LifeTime Analysis	Metric Computation	Invariant Detection
Instrumentation Based	Daikon	C, C++									✓
	Valgrind	C, C++				✓		✓			
	Rational Purify	C, C++, Java				✓					
	Parasoft Insure++	C, C++		✓		✓					
	Pin	C	✓								
	Javana	Java	✓						✓		
	DIDUCE	Java									✓
AOP Based	DJProf	Java		✓					✓		
	Racer	Java						✓			
VM Profiling Based	Caffeine	Java							✓		
	DynaMetrics	Java								✓	
	*J	Java								✓	
	JInsight	Java				✓	✓		✓		

Valgrind[35] is an instrumentation framework for building dynamic analysis tools. It can automatically detect many memory management and threading bugs, and profile a program in detail. Purify[38] and Insure++[36] have similar functionality as Valgrind. Whereas Valgrind and Purify instrument at the executables, Insure++ directly instruments the source code. Pin[40] is a tool for dynamic binary instrumentation of programs. Pin adds code dynamically while the executable is running. Pin provides an API to write customized instrumentation code (in C/C++), called Pintools. Pin can be used to observe low level events like memory references, instruction execution, and control flow as well as higher level abstractions such as procedure invocations, shared library loading, thread creation, and system call execution.

Javana [30] runs a dynamic binary instrumentation tool underneath the virtual machine. The virtual machine communicates with the instrumentation layer through an event handling mechanism for building a vertical map that links low-level native instruction pointers and memory addresses to high-level language concepts such as objects, methods, threads, lines of code, etc. The dynamic binary instrumentation tool then intercepts all memory accesses and instructions executed and provides the Javana end user with high-level language information for all memory accesses and natively executed instructions[30].

Daikon[16] and DIDUCE[19] are two most popular tools for invariant detection. The former is an offline tool while the latter is an online tool. The major difference between the two is that while Daikon generates all the invariants and then prunes them depending on a property; DIDUCE dynamically hypothesizes invariants at each program point and only presents those invariants which have been found to satisfy a property. Another major difference is that Daikon collects tracing information by modifying the program abstract syntax tree, while DIDUCE uses BCEL to instrument the class JAR files.

*J [15] and DynaMetrics[41] are tools for computing dynamic metrics for Java. While *J relies on JVMPI (predecessor of JVMTI but now rarely used) interface for metrics computation, DynaMetrics uses JVMTI interface. Another major difference between the two is that *J computes dynamic metrics specifically defined by its authors for Java whereas DynaMetrics computes major dynamic metrics from various dynamic metrics suites available in literature. JInsight [12] is used for exploring runtime behaviour of Java programs visually. It offers capabilities for managing the information overload typical of performance analysis. Through a combination of visualization, pattern extraction, interactive navigation, database techniques, and task-oriented tracing, vast amounts of execution information can be analyzed intensively. Caffeine [31] is a tool that helps a maintainer to check conjectures about Java programs, and to understand the correspondence between the static code of the programs and their behavior at runtime. Caffeine uses JPDA and generates and analyzes on-the-fly the trace of a Java program execution, according to a query written in Prolog. DJProf[37] is a profiler based on AOP which is used for the analysis of heap usage and object life-time analysis. Racer[6] is a data race detector tool for concurrent programs employing AOP. It has specific point-cuts for lock acquisition and lock release. These point-cuts allow programmers to monitor program events where locks are

granted or handed back, and where values are accessed that may be shared amongst multiple Java threads.

4 Conclusion

Dynamic analysis has acquired a great importance in recent years because of its ability to determine run-time behavior of programs precisely. This paper provided the details of the techniques and tools of dynamic analysis. An attempt is made to highlight the strength and weaknesses of each technique. Aspect oriented techniques have got an edge over other techniques and are being emphasized for dynamic analysis of programs.

References

1. Ball, T.: The concept of dynamic analysis. In: Wang, J., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, p. 216. Springer, Heidelberg (1999)
2. Ball, T., Larus, J.R.: Efficient path profiling. In: Proceedings of MICRO 1996, pp. 46–57 (1996)
3. Baxter, I.: DMS: Program Transformations for Practical Scalable Software Evolution. In: Proceedings of the 26th International Conference on Software Engineering, pp. 625–634 (2004)
4. Binder, W., Hulaas, J., Moret, P., Villazón, A.: Platform-independent profiling in a virtual execution environment. Software: Practice and Experience 39(1), 47–79 (2009)
5. Binkley, D.: Source Code Analysis: A Road Map. Future of Software Engineering (2007)
6. Bodden, E., Havelund, K.: Aspect-oriented Race Detection in Java. IEEE Transactions on Software Engineering 36(4), 509–527 (2010)
7. Boner, J.: AspectWerkz - Dynamic AOP for Java. In: Proceeding of the 3rd International Conference on Aspect- Oriented Development (AOSD 2004), Lancaster, UK (2004)
8. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems, Grenoble, France (2002)
9. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching. International Journal of High Performance Computing Applications, 317–329 (2000)
10. Chiba, S.: Load-time structural reflection in java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, p. 313. Springer, Heidelberg (2000)
11. Dahm, M.: Byte code engineering. In: Java-Information-Tage(JIT 1999) (1999), `http://jakarta.apache.org/bcel/`
12. Zheng, C.-H., Jensen, E., Mitchell, N., Ng, T.-Y., Yang, J.: Visualizing the Execution of Java Programs. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 151–162. Springer, Heidelberg (2002)
13. Dieckmann, S., Hölzle, U.: A study of the allocation behavior of the SPECjvm98 Java benchmarks. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 92–115. Springer, Heidelberg (1999)
14. Dufour, B., Goard, C., Hendren, L., de Moor, O., Sittampalam, G., Verbrugge, C.: Measuring the dynamic behaviour of AspectJ programs. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 150–169. ACM Press (2004)

15. Dufour, B., Hendren, L., Verbrugge, C.: *J: A tool for dynamic analysis of Java programs. In: Proc 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 306–307. ACM Press (2003)
16. Ernst, M.D.: Dynamically Discovering Likely Program Invariants. (PhD Dissertation), University of Washington, Dept. of Comp. Sci. & Eng., Seattle, Washington (2000)
17. Ernst, M.D.: Static and Dynamic Analysis: Synergy and Duality. In: Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2004)
18. Gupta, V., Chhabra, J.K.: Measurement of Dynamic Metrics using Dynamic Analysis of Programs. In: Applied Computing Conference, Turkey (2008)
19. Hangal, S., Lam, M.S.: Tracking Down Software Bugs using Automatic Anomaly Detection. In: ICSE 2002 (2002)
20. Hilyard, J.: No Code Can Hide from the Profiling API in the.NET Framework 2.0. MSDN Magazine (January 2005), http://msdn.microsoft.com/msdnmag/issues/05/01/CLRProfiler/
21. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: Lieberherr, K. (ed.) Aspect-oriented Software Development (AOSD 2004). ACM Press (2004)
22. Hollingsworth, J.K.: MDL: A language and compiler for dynamic instrumentation. In: Proceedings of International conference on Parallel architecture and compilation techniques (1997)
23. IBM Corporation. Jikes Bytecode Toolkit (2000), http://www-128.ibm.com/developerworks/opensource/
24. Jackson, D., Rinard, M.: Software Analysis: A Road Map. IEEE Transaction on Software Engineering (2000)
25. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–355. Springer, Heidelberg (2001)
26. Kiczales, G., et al.: Aspect-oriented programming. In: Proc of the 11th European Conference on Object-Oriented Programming, Finland. LNCS, vol. 1241, pp. 220–242. Springer (1997)
27. Larus, J.R., Ball, T.: Rewriting executable to measure program behavior. Software:Practice and Experience 24(2), 197–218 (1994)
28. Larus, J.R., Schnarr, E.: EEL: Machine independent executable editing. In: PLDI 1995 (1995)
29. Lee, H.B., Zorn, B.G.: BIT: A tool for instrumenting Java bytecode. In: Proceedings of USENIX Symposium on Internet Technologies and Systems, pp. 73–82 (1997)
30. Maebe, J., Buytaert, D., Eeckhout, L., De Bosschere, K.: Javana: A System for Building Customized Java Program Analysis Tools. In: OOPSLA 2006 (2006)
31. Mines de Nantes, E.: No Java without Caffeine. In: ASE 2002 (2002)
32. Mock, M.: Dynamic Analysis from the Bottom Up. In: 25th ICSE Workshop on Dynamic Analysis (2003)
33. Naik, M.C.: A static and dynamic program analysis framework for Java (2010), http://chord.stanford.edu/
34. Nimmer, J.W., Ernst, M.D.: Automatic generation and checking of program specifications. Technical Report 823, MIT Lab for Computer Science, USA (2001)
35. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2007)

36. Parasoft Inc. Automating C/C++ Runtime Error Detection With Parasoft Insure++. White paper (2006)
37. Pearce, D.J., Webster, M., Berry, R., Kelly, P.H.J.: Profiling with AspectJ. Software: Practice and Experience (2007)
38. Rastings, R., Joyce, B.: Purify: Fast Detection of Memory Leaks and Access Errors. Winter Usenix Conference (1992)
39. Reiss, S.P.: Efficient monitoring and display of thread state in java. In: Proceedings of IEEE International Workshop on Program Comprehension, St. Louis, MO, pp. 247–256 (2005)
40. Skatelsky, A., et al.: Dynamic Analysis of Microsoft Windows Applications. In: International Symposium on Performance Analysis of Software and System (2010)
41. Singh, P.: Design and validation of dynamic metrics for object-oriented software systems. (PhD Thesis), Guru Nanak Dev University, Amritsar, India (2009)
42. Spinczyk, O., Lohmann, D., Urban, M.: Aspect C++: an AOP Extension for C++. Software Developer's Journal, 68–76 (2005)
43. Srivastava, A., Eustace, A.: ATOM: A system for building customized program analysis tools. SIGPLAN Not 39(4), 528–539 (2004)
44. Sun Microsystems Inc. JVM Tool Interface, JVMTI (2004), `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html`
45. Sun Microsystems Inc. Java Virtual Machine Profiler Interface, JVMPI (2004), `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`
46. Sun Microsystem Inc. Java Platform debug Architecture (2004), `http://java.sun.com/javase/6/docs/technotes/guides/jpda/`
47. Waddington, D.G., Yao, B.: High Fidelity C++ Code Transformation. In: Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications, Edinburgh, Scotland, UK (2005)
48. Waddington, D.G.: Dynamic Analysis and Profiling of Multi-threaded Systems. In: Taiko, P.F. (ed.) Designing Software-Intensive Systems: Methods and Principles. Information Science Reference Publishing (2008) ISBN 978-1-59904-699-0
49. `https://software.intel.com/en-us/intel-parallel-studio-xe`