Languages for Dynamic Instrumentation

Steven P. Reiss, Manos Renieris
Department of Computer Science
Brown University
Providence, RI 02912-1910
401-863-7641, FAX: 401-863-7657
spr@cs.brown.edu

Abstract

Dynamic instrumentation has proven to be a valuable technique for a variety of different program analyses. However, developing a new analysis based on dynamic instrumentation is difficult, error prone, and time-consuming. One obvious solution is to develop a common framework that would enable quick and easy dynamic instrumentation for a variety of applications. Developing a practical solution along these lines, however, requires that one understand and effectively model how instrumentation can and should be used. We suggest that an event-oriented framework based on program analysis might be a viable approach to achieving such a practical solution.

1. Motivation

Dynamic analysis has been used for a wide variety of different applications, from simple profiling to program understanding. We alone have been using it in a variety of projects for performance analysis, visualization, program modeling, and fault location. In most of the applications of dynamic analysis, the technique has demonstrated itself to be an invaluable tool that is able to provide insights far beyond those of static analysis.

Even so, dynamic analysis still sees only limited use in day-to-day applications, in today's programming environments, and by most programmers. There are several reasons for this disparity, but most rise from the fact that dynamic analysis is expensive, both in terms of the overhead involved in collecting the appropriate data, in terms of developing practical instrumenters, and in terms of developing tools that can use the data.

What is needed is a framework to support dynamic analysis that could be used practically for a variety of different applications. If such a framework existed, it would be relatively easy to develop new applications of dynamic analysis and to incorporate them into today's programming tools. Moreover, by concentrating on implementing a framework that minimized instrumentation overhead, something that is often too difficult to do for any single application, developers would ben empowered to use dynamic-analysis based tools as part of their everyday programming.

In this position paper, we outline some of the interesting research issues that arise in attempting to define (and later implement) such a framework.

2. Requirements

A practical dynamic analysis framework has to meet a broad range of requirements. These are needed both to make it applicable to a variety of different applications and to ensure that it can be used on a variety of real systems. These requirements include:

- Low overhead. One of the key problems with dynamic analysis is the overhead it imposes on the underlying applications. Minimizing this overhead by intelligent, problem-specific instrumentation should be the primary goal of the framework. The next three requirements are based on this.
- Selectivity. The application should be able to specify what portions of the system should be instrumented and what data should be collected. This should be done at as fine a level as possible.
- Semantic basis. The selection of what to instrument and what and when to collect data should be based on the structure and semantics of the program. This implies that dynamic analysis should be predicated on some underlying static analysis.
- Temporal. Instrumentation should be limited not only by specific portions of the program, but also by those parts of the execution that are relevant. This temporal information might be determined a priori or dynamically.
- Real Programs. A second key problem with today's dynamic instrumentation tools is they often are not capable of handling the wide range of programs that developers are interested in. The next three requirements follow from this.
- Handle Libraries. Much of the work in today's applications is done inside system or user libraries. To do appropriate analysis, one often needs dynamic information from these libraries. Moreover, to understand the

semantics of the application, one must often understand the semantics of the libraries. This requirement becomes more complex when one realizes that source is often not available for many libraries.

- Handle Threads. Programs written in Java or C# are, more often than not, multithreaded. An instrumenter needs to be able to deal with the underlying complexities both in terms of collecting appropriate data and in terms of not imposing additional synchronization points on the application and thus changing its behavior.
- Extend to Systems. Many of today's programs are actually multiple-process distributed systems. The analysis and hence instrumentation that needs to be done on these systems will require correlating data accumulated from the different processes into a single analysis.
- Usable Results. The third key problem in today's instrumenters is that the data that is produced is often very specific to a particular application and not easy to reuse in other applications. What is needed is a relatively standard data format that can serve as the basis both for immediate and deferred analysis.

Meeting these requirements will be difficult. However, by using the collective experience from current instrumenters, static analyzers, aspect-oriented programming, and other areas, it should be possible to develop an appropriate framework.

3. Framework Overview

We envision a framework that is built on two "languages". The first is used to let a tool define what portions of a system should be instrumented and what information is required from those portions. This will be used by a instrumentation tool to produce one or more event streams describing appropriate portions of the execution.

The second language will let a tool define how these event streams should be processed to produce the data needed for analysis. This could involve generating higher-level events streams, accumulating information, tracking program or object states, or other analysis techniques. The framework would use this description to process the events as they were generated as efficiently as possible.

Central to this framework is the notion that both languages can make direct use of information about the system being analyzed. This means that they should be able to refer to basic blocks, to the definitions and uses of particular variables or fields, to def-use chains, and to particular packages, libraries, and routines.

4. Instrumentation Definition Language

The first part of this framework is dependent on a language that lets the developer describe the information that needs to be collected from dynamic instrumentation at a fine level of detail.

This language should be geared toward generating events streams. Events are a general purpose mechanism that closely matches the methodology of run time instrumentation. The underlying framework will have to deal with many types of parameterized events. These would include:

- Call/Return of a method;
- Definition/Use of a value;
- Enter/Exit of a basic block;
- Throw/Catch of an exception;
- Create/Start/Stop/Wait/IO/Run of a thread;
- Read/Write of a location or field;
- Allocate/Free of an object;
- Send/Receive of a message;
- Program specific events.

The set of events that are relevant to a particular program or run needs to be specified in a high level way. This will sometimes be done globally (e.g. interest in all call/return events for profiling), and sometimes very program specific (e.g. when does field X change in method Y; when is method A called with parameter B). Moreover, the set of events generally should be independent of the code.

In both cases static analysis of the program, typically done at the byte or machine code level, will be appropriate. This analysis should let one specify, for example:

- That one wants to detect the start of each basic block. The resultant instrumentation could then make use of control flow analysis to minimize the amount and size of instrumentations.
- That one wants to track field accesses for a particular set of field writes. This would require data flow analysis to determine which reads in the program might be relevant to the particular writes.
- That one wants to detect calls to a particular set of methods for objects allocated at a certain point in the program. For example, one might want to check that a particular instance of a Java iterator is used correctly.
- That one wants to detect reads and writes of shared storage. This would require static analysis to determine what fields can be accessed by multiple threads and which accesses to those fields should be considered shared.

The research in this area is to attempt to put together a language that allows an analysis application to specify what set of events it wants from the program. This could either be a language per se, an XML file describing the set

of events, or event an appropriate set of function calls and callbacks.

This "language" will have to deal with all the issues outlined above — handling a wide range of events, being able to specify those events to apply to the whole program or large portions of it, being able to restrict those events to particular locations based on semantic properties of the program, and allowing a variety of different parameters to be associated with each event.

Such a language is useless without an appropriate implementation. This is again a research problem involving what and how to do the static analysis needed to minimize instrumentation, techniques for dynamically inserting and removing instrumentation, and automatic optimization of instrumentation based on semantic information.

5. Analysis Language

While event streams are a logical conceptual output from an instrumentation front end, what is often needed is the result of analysis based on the event stream rather than the event stream itself. There are several different types of such analysis that are particular to the applications of runtime instrumentation.

For visualization and some program understanding applications, it will be desirable to map the event stream into a sequence of higher-level events. This can occur within an event stream (for example, mapping basic block event to program path events), or it might occur among multiple event streams (for example, taking information about monitor entry and exit events from multiple threads and using this to generate events denoting what threads are blocking on what other threads).

For performance analysis and related applications, it is desirable to accumulate information from the event sequence. One might want to look at the total number of calls of each method, the number of allocations of each class, the time spent in each method, or the number of calls of each method pair. This information might be further confined by accumulating information by class or package or event according to higher-level events such as user interface interactions or remote procedure calls.

Another application area for run time instrumentation is involves the dynamic checking of semantic properties of a system. These properties are typically specified using finite state systems (either using pure or extended FSMs, using regular or path expressions, or using a language such as LTL or CTL). What one wants to get out of instrumentation here is whether the actual program run satisfied or did not satisfy the specification. This implies that the sequence of run time events generated by the front end needs to be filtered and then use to check against the underlying automata.

In each of these cases, the appropriate analysis can be done either after the fact or while doing tracing. After the fact analysis is easier in that one can isolate the analysis from the instrumentation and can easily do several different analyses of the same instrumented run. This is advantageous, for example, in software visualization where the user will want to see different views of the run and the exact nature of those views might not be known in advance.

In most applications, however, the raw event streams are going to be substantially larger and more complex than the results of the analysis. Here, it is much more effective to do the event analysis on the fly, storing only the accumulated result. An ideal instrumentation environment should provide a stream-based processing language that would facilitate this. Again, this could be a real language, a high-level XML description of what needs to be done, or simply a reasonable programming interface that facilitates the appropriate processing.

We note that this language and facility will probably need to have access to the semantic analysis that was used in doing the actual instrumentation in order to correctly interpret the events. This information will either have to be recomputed or will be stored in auxiliary files as part of the instrumentation process.

The interesting research issues here are first attempting to determine the appropriate range of analyses that should be doable dynamically, in determining what is an appropriate interface for doing these analyses, and in providing a very efficient but generic implementation mechanism that will support the analyses. Other research issues that come up involve ways of combining multiple event streams in the analysis milieu and doing all this without significantly affecting the behavior of the program being instrumented.

6. Example Approaches

While we have not built anything that meets the needs outlined above, we have and continue to work on a variety of different approaches that make us believe that the general mechanisms described here can be achieved.

We currently support several different instrumenters for different applications. For software visualization, we have two instrumenters, one for C/C++ and one for Java. Both are capable of instrumenting the user's application and all the appropriate libraries. Both handle multiple threads and offer a limited degree of selectivity as to what information to obtain. While the initial data is obtained as a set of independent event streams, one per thread, this data is processed dynamically into a common sequence. Additional, on-the fly or after-the-fact processing can be done within the system for a wide variety of different resultant analyses.

For dynamic visualization of software, we have developed an instrumenter that accumulates a variety of data over millisecond time intervals and passes the accumulated data to a front end. Using a variety of techniques, we were able to limit the performance loss due to instrumentation (which includes every call, return, allocation, thread state change, and synchronization event) to a factor 2-3.

For analyzing buggy programs, we developed a variety of different instrumenters. << MANOS >>

Finally, we are developing a tool for checking finite state properties of programs through a combination of static and dynamic checking. Given a description of a program property, this tool is able to find the relevant locations in the source that affect that property, determine whether the property needs to be checked dynamically or if it can be determined statically, and, in the case where dynamic checking is necessary, it actually determines exactly what instrumentation is needed to check the property.

7. Acknowledgements

This work was done with support from the National Science Foundation through grants ACI9982266, CCR9988141, and CCR9702188 and with the generous support of Sun Microsystems.

8. References