Software instrumentation using dynamic techniques

Michiel Ronsse Jonas Maebe Koen De Bosschere
Department of Electronics and Information Systems
Ghent University, Belgium
{ronsse|jmaebe|kdb}@elis.rug.ac.be

Abstract— **In this paper, we describe *DIOTA*, a dynamic instrumentation technique. The technique correctly deals with programs that contain traditionally hard to instrument features such as data in code and code in data. The technique does not require reverse engineering, program understanding tools or heuristics about the compiler or linker used. The basic idea is that instrumented code is generated on the fly, while the original process is used for data accesses. *DIOTA* comes with a number of useful backends to check programs for faulty memory accesses, data races, deadlocks, . . . and perform basic tracing operations, e.g. tracing all memory accesses, all code being executed, to perform coverage analysis, . . . *DIOTA* has been completely implemented for the IA32 architecture.**

I. Introduction

PROGRAM instrumentation is a general term to denote the techniques used to modify existing programs in order to collect additional data during a program execution (profile information, tracing, cache simulation, etc.). The basic technique of program instrumentation is to insert instrumentation code at points of interest in the program. During the execution, the instrumentation code is then executed together with the original program code.

An important characteristic is the *abstraction level* at which the program instrumentation is done. This abstraction level can be the hardware level, the library level, the source code level or the machine instruction level. It turns out that the only true viable instrumentation method is dynamic instrumentation: instrumentation code is added to the executable while it is running [HMC95]. This way, also the dynamic linked libraries can be instrumented.

This paper describes *DIOTA* (Dynamic Instrumentation, Optimization and Transformation of Applications), a dynamic instrumentation technique. *DIOTA* is built using a modular approach and can easily be extended with additional functionality. This is e.g. necessary to do something functional with the added instrumentation code. To deal with this *DIOTA* uses (user written) backends that instruct *DIOTA* what and how to instrument. In order to demonstrate the usefulness of *DIOTA*, a number of backends have been written so far.

The basic idea of *DIOTA* is to keep a running program unaltered in memory and to generate (instrumented, optimized or transformed) code on the fly in memory. The generated code will be constructed in such a way that it will use the original program for its data accesses while the generated code (called clone) is used for all code accesses. During the execution, the code is taken from the clone, while the data is taken from the original process. Hence, instrumenting data in the code does not harm as data is always taken from the original program. Code in data is also treated correctly because data is also instrumented as if it were code. Self-modifying code can be processed elegantly by instrumenting the store-operations in such a way that they are not limited to writing the data in the original program, but that they also (re)generate an instrumented version of the data.

Although *DIOTA* has been implemented for the Intel IA32 architecture the ideas can be easily applied to other architectures. *DIOTA* is loosely based on *JiTI* [RDB01], an instrumentation tool we developed for the SPARC architecture.

In this paper, we focus on the techniques used by *DIOTA* to instrument individual instructions. Although *DIOTA* can also intercept dynamic linked library functions, this functionality is not discussed in this paper.

II. *DIOTA*

A. Overview

Two major difficulties when inserting code into binaries are (i) correctly relocating the code and data after inserting instrumentation code and (ii) correctly distinguishing between code and data (especially when code is located in data or when data is interspersed with code). In existing systems, these two difficulties could only be solved by applying a sophisticated analysis (disassembly) of the binary, using assumptions about the origin of the code. Most systems can be broken by offering hand written machine code to it.

DIOTA, on the other hand, has no such limitations. The reason is that it just follows the code as it is executed, so that it does not have to do any guesswork. As shown in Figure 1, the general operation is quite simple: once the instrumentation has started, successive instructions are processed until a branch is encountered. Those with immediate target addresses can be followed to their destination, but the ones with a variable successor must be evaluated at the moment they are executed. The limit imposed on the number of times immediate branches are followed to their destination prevents endless loops in the instrumentation engine.

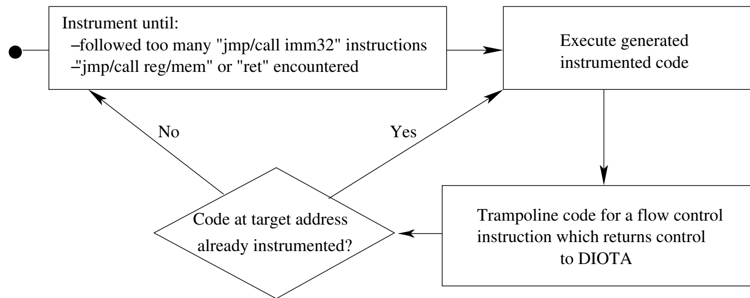

Fig. 1. General overview of *DIOTA*'s operation

Once a condition to stop instrumenting is met, *DIOTA* inserts a trampoline in the generated code, which will return control to *DIOTA* when it is reached. Next, the generated code is executed at full processor speed. Of course, it will not be as fast as the original code, with the slowdown factor depending on the amount of extra instrumentation code that has been inserted, but it will still be much faster than emulation. Once *DIOTA* is reactivated through a trampoline, it checks whether it has already instrumented the code at the target address. If that is the case, it simply jumps to this code, otherwise it generates a new bock of code.

Since the original program is left untouched and the data is not moved either, no relocation has to be done during the instrumentation. When a subroutine call is encountered, the return address in the original code will be pushed on the stack, so that code which uses this information (e.g. for the purpose of generating a backtrace) will also still run correctly. The return instruction is handled like any other branch with a variable destination, so instead of returning to the original program code at the end of a function, *DIOTA* stays in control. The end result is that the amount of extra code that is inserted in the instrumented version of the program does not matter and at no time the original program notices it is being instrumented.

Another advantage of not modifying the original code, is that the variable length property of the IA32 instruction set does not cause any problems. Although it is always possible to use the `int3` (breakpoint) instruction, which needs only 1 byte, to interrupt a program at any time, it is not optimal in terms of speed or usability. The reason is that such an instruction causes a signal, which means that the kernel must be involved, and not all functions of `libc` can be called safely from within a signal handler.

B. Functionality

DIOTA has several modes of operation, each of which can be used separately, but most can be combined as well. The currently implemented modes are listed below.

- *Memory instrumentation* In this case, all instructions that access memory are preceded by a call to one or more user-defined instrumentation routines which have access to the address of the original instruction, the type of memory access (load, store, modify) and the address and size of the memory location that is being accessed.
- *Code instrumentation* When activated, this feature inserts a call to one or more user-defined instrumentation routines at the end of every basic block in the machine code.
- *Interception of function calls* *DIOTA* allows the interception of calls to routines in dynamically linked libraries used by the program. This interception includes both calls from the main program and other libraries and direct calls from within the library that contains the function itself.

Apart from the aforementioned modes, *DIOTA* also supports the concept of so-called backends. These are user-written modules that link to the main *DIOTA* library and which select the modes to activate (at run-time), which callbacks should be installed and which functions should be intercepted. Every module resides in its own shared library, allowing several random backends to be used at the same time.

C. Instrumentation

DIOTA has been implemented for IA32 systems running GNU/Linux. It is a dynamic library that can be attached to a program by using the `LD_PRELOAD` environment variable that forces the dynamic loader to load *DIOTA* even though the application is not explicitly linked to it. Through the use of an `init()` function, *DIOTA* manages to be activated before the program is started. It then replaces the start address of the loaded ELF image with an address in the clone, so that the code generated by *DIOTA* is executed instead of the original program.

Initially, the clone only contains a *trampoline*. This is a small piece of code that first pushes information about the original instruction corresponding to it on the stack and then passes control on to *DIOTA*. Such constructs are used a lot throughout *DIOTA* in different forms, but the principle is always the same: transfer control from the currently executing thread back to *DIOTA* along with any necessary context information, so that it can decide what to do next.

In the case of the start of the program, this means disassembling the instructions of the `main()` function. All successive instructions are copied to the clone, possibly together with some

instrumentation code or in an altered form, depending on the used modes. This can continue until a flow control instruction is encountered. The IA32 architecture contains such instructions both with relative, immediate and absolute, indirect addresses.

Since the original address of the instruction that is being worked on is known, the target in the original code of a branch with an immediate operand can be calculated at once. This means that such a branch does not have to be included in the instrumented code, the instrumentation can simply continue at its target. In case it was a `call`, the (original) return address still has to be pushed on the stack. To prevent endless loops, a limit is imposed on the number of times a direct branch is followed to its destination. Once that limit is reached, a trampoline is put in the clone which links back to *DIOTA* and the code generated until now is executed.

For conditional branches, the fall-through path is instrumented and a trampoline that is the target of the branch is created. This trampoline will transfer control to *DIOTA* if the branch is taken, and at this moment the branch will be instrumented.

The target address of flow control instructions with absolute targets cannot be used directly, since in that case the execution would return to the original code and *DIOTA* would lose control over the program. As a result, all such instructions are replaced by a trampoline which passes the actual target address and information about this instruction to *DIOTA*. This way, every time the instruction would be executed, *DIOTA* can check whether it has already processed the code at the current target address and if not, do the processing at that moment.

III. Backends

In order to show the usefulness of *DIOTA* as an instrumentation tool, several backends have been developed. Such a backend is implemented as a dynamic library that is loaded in the same way as *DIOTA* itself.

The available backends at this moment are:

- a number of simple tracers that collect all data accesses and all code accesses. These tracers can be used to implement a cache simulator, to get a list of all instructions executed, to perform coverage analysis, . . .
- a simple memory sanity checker that checks for pointer errors, array indexes that are too high, erroneous `malloc()/free()` combinations, . . .
- a record/replay module that is able to replay the synchronization operations in a parallel program. This can be used to enable cyclic debugging techniques for non-deterministic parallel programs.
- a module that checks a parallel execution for the occurrence of data races. This module is normally used in combination with the record/replay module. More information about the methods used by this backend can be found in [RDB99].
- a module that checks a parallel execution for (possible) deadlock or livelock.

It is possible to load a number of backends at the same time, e.g. to collect a number of traces for the same execution.

IV. Evaluation

The basic instrumentation functionality of *DIOTA* is relatively fast. Programs are only slowed down by 20 to 400% when running under control of *DIOTA* compared to a normal execution. The more the already instrumented code gets executed and the less return instructions there are in the often executed parts of the code (like e.g. in *bzip2*), the smaller the slowdown becomes. When using memory or code instrumentation, the overhead naturally increases, especially with the former since in that case every instruction that accesses memory is accompanied by a call to the installed callbacks (a huge slowdown is reported by all tools that intercept all memory operations, such as [HJ92]). The total speed decrease will then also largely depend on the amount of work done in these callbacks.

The implementation of *DIOTA* is quite solid, which is demonstrated by the fact that fairly complex programs such as the *Mozilla* and *Konqueror* browsers work fine when they are running under its control. *DIOTA* has already been used to (successfully) detect data races in a number of programs. To do complex tracing, very fast processors and much time are still required however. The memory requirements on the other hand are quite reasonable, since only the executed code needs to be instrumented and duplicated. Most of the time, the data takes up much more space.

V. Conclusions

In this paper, we have described *DIOTA*, a program instrumentation technique that is able to correctly instrument hard to instrument features such as data in code and code in data and that requires no recompilation, relinking or the usage of symbol or debugging information. *DIOTA* is implemented as a dynamic library, allowing it to be applied to existing applications. Hence, the instrumentation is a property of an execution, and not of the program itself. *DIOTA* and a number of backends have been implemented for the IA32 architecture. In the future, we will focus our attention on the usage of *DIOTA* as a dynamic optimizer. The basic infrastructure and a number of small optimizations are already present in *DIOTA*.

References

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. Proceedings of the Winter USENIX Conference, pages 125–136, January 1992.

[HMC95] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. Computer Sciences Department, University of Wisconsin-Madison, May 1995. SHPCC.

[RDB99] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[RDB01] Michiel Ronsse and Koen De Bosschere. Jiti: A robust just in time instrumentation technique. volume 29 of *Series Computer Architecture News*, chapter Proceedings of Workshop on Binary Translation - 2000, pages 43–54. ACM Press, March 2001.