DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications

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

http://www.elis.rug.ac.be/~ronsse/diota/

***Abstract*— In this paper, we describe *DIOTA*, a novel method for instrumenting binaries. 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 implemented for the IA32 architecture running the Linux operating system.**

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. In some cases it can also be used to really modify the semantics of the program, e.g., to temporarily fix errors without recompiling or to circumvent authentication routines. The former type of program instrumentation is called *passive instrumentation*, while the latter form is called *active instrumentation*. In what follows, we will focus on passive instrumentation. Our results can however be generalized to active instrumentation too.

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. The former two are actually an instrumentation of the environment that must execute the program while the latter two are instrumentations of the program itself, leaving the environment intact.

In cases were absolute non-intrusiveness is of paramount importance (real-time systems, or non-deterministic systems) the only viable option is instrumenting the hardware. The most common way to do this is to use an emulator, which means that a processor is replaced by a complete computer system that behaves like the processor. If the emulator is cycle-true, it will execute a given program in exactly the same number of cycles as the processor it is emulating. The advantage of using an emulator is that full program traces can be recorded and stored without slowing down the program execution. The disadvantage is that this solution is expensive and not available for the fastest processors (the emulator must always be faster than the processor it emulates). It is however common practice for the development of sophisticated embedded applications that do not require extremely fast processors.

A less expensive solution is to build custom hardware to observe the various computer busses [TFCB90], [MN89], [BG91]. The problem with this solution is that modern state of the art processors have on-chip caches so it is extremely hard to discover what is going on inside the processor by just observing the bus. Disabling the caches eliminates this problem, but causes a substantial slowdown, and therefore makes this technique intrusive. Another alternative for extra hardware is to use an architectural simulator to execute the program (e.g. the multiprocessor simulators Mint [VF93], Tango [DGH91], Proteus [BDCW91] and Shade [CK90] or the workstation simulator SimICS [MDG+98]). This is much slower than a real execution, but it can produce exactly the same (timing) results as a real execution. Obviously, this is not feasible for instrumenting real-time systems that must interact with their environment. With hardware-based instrumentation, the instrumentation is clearly a property of the execution, not of the program. A fairly straightforward technique for instrumentation with low granularity such as instrumenting the synchronization or the input/output of a program, is to instrument the corresponding library routines. Linking an application program with an instrumented library is enough to produce an instrumented version of the program. This technique has been used successfully in our RecPlay [RDB99] system to collect synchronization traces. When using dynamic libraries, the relinking can be done at

load-time which makes this kind of instrumentation a property of the execution, not of the program.

Another straightforward technique is to instrument the source code (examples are Insight [Par96], Trapeds [SJF92], and lock_lint [Sun94]) or the assembler output from the compiler [EKKL90]. This can only be done when the full source code is available. The instrumentation code is inserted and compiled, producing an instrumented version of the program. The major limitation of this technique is that the source code of the program must be available, which is not always the case, especially for libraries. In this case, the instrumentation is a property of the program, not of the execution.

The final technique, instrumenting at the machine instruction level, does not have this limitation. It starts from the executable form of a program and adds the necessary instrumentation code to it. This instrumentation technique is very flexible and can be implemented as a property of the program (when done statically by e.g. the linker), or it can be a property of an execution when the instrumentation is done at e.g. runtime, in which case the dynamically loaded libraries can also be instrumented. Examples of static tools are Purify [HJ92], IPS-2 [MCK$^+$90], Cedar [Mal87] and the basic block counting tools Pixie, Epoxy, QPT [BL92] and Mtool [GH91], EEL [LS96] for Sun SPARC machines, OM [SW93], ATOM [SE94] and Alto [DBD96] for Digital Alpha machines and ETCH [LRV$^+$] for Intel machines.

A dynamic binary instrumentation tool is Paradyn and its derived DynInst API [MCC$^+$95], [HMG$^+$97]. The tool runs on a number of processor architectures and was developed for performance measurements. The system allows instrumentation (in the form of *code patches*) to be added ('spliced') at procedure entries, procedure exits and individual call statements.

A novel and dynamic instrumentation tool for instrumenting kernels is described in [TM99a], [TM99b]. Although the tool is dynamic, it performs the same analysis as static tools to create an interprocedural control flow graph of basic blocks and finding live registers for each basic block. As such the tool has the same problems as static tools: data in code, code in data and handwritten and self-modifying code. However, relocation is not needed as the tool replaces instructions with a branch to *springboards* to jump to the actual instrumentation code.

A very recent instrumentation tool for the Intel A32 architecture is Valgrind [Sew02]. Valgrind is a tool that detects memory-management problems. When a program is run under Valgrind's supervision, all reads and writes of memory are checked and calls to `malloc()/new()/free()/delete()` are intercepted. Valgrind uses techniques that are very similar to the methods we use (e.g. instrumented code is generated on the fly), but currently does not support threads or MMX, SSE, SSE2, ... instructions. Valgrind attaches itself to any dynamically-linked ELF x86 executable, without modification, recompilation, or anything. No performance data is given, but we expect its overhead to be quite high given the fact that all IA32 instructions are translated to a number of simpler RISC-like instructions, instrumentation is added to these instructions and the resulting instruction sequence is then translated back to IA32 instructions.

There are however also a number of problems with dynamic instrumentation:

1) it requires that the executable program is first disassembled and a control flow graph is constructed before instrumentation code can be inserted. This step is a weakness as it is not always easy to reconstruct the original program (for reasons such as code in data or data in code);
2) inserting instructions requires the modification of code addresses (absolute as well as relative addresses). Since the code size is increasing, some relative addresses may grow beyond their maximal size;
3) not all instructions are easy to instrument. Instructions in delay slots of some RISC-processors are the most notable example;
4) self-modifying code is almost impossible to deal with.

In practice, all existing systems can be broken by adding some hand-written assembly language to it such as code in data or data in code. Sometimes, the instrumented programs will simply crash, on other occasions they will return wrong results.

This paper describes *DIOTA* (Dynamic Instrumentation, Optimization and Transformation of Applications), an instrumentation technique that can overcome these problems. As the name suggests, *DIOTA* is not only able to instrument applications during their execution, but *DIOTA* can also be used to transform or optimize a running application. This paper discusses the common base that is used by *DIOTA* to implement these three goals.

DIOTA is built using a modular approach and can be easily 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 at its original location in memory and to generate (instrumented, optimized or transformed) code on the fly somewhere else. 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 the *clone*) is used for all code accesses.[1] 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 hurt 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

[1] *DIOTA* is a Latin word that means vase or drinking cup having two handles or ears as used by the Romans. It literally means *two handles*, and this reflects the way our system works: we use one handle for data accesses and a second handle for code accesses.

(re)generate an instrumented version of the data.[2]

This paper describes *DIOTA* in full detail. The tool has been implemented for the Intel IA32 architecture running the Linux operating system, but 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.

The paper now continues with a detailed description of *DIOTA*, followed by a description of the currently available backends in section III. We conclude with a small evaluation part in section IV and with the final conclusions in section V.

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 (disassembled) until a branch is encountered. Those with immediate target addresses can be followed to their destination, but the ones with a variable successor (indirect branches) 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.

Once a condition to stop instrumenting is met, *DIOTA* inserts a trampoline in the generated code. This trampoline will, when executed, push the address of the instruction that should have been executed next on the stack and then return control to *DIOTA*. After generating this trampoline, *DIOTA* restores the stack pointer to its original position and jumps to the code it just generated (by pushing the address of this code on the stack and executing a `ret` instruction), which is then 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 block 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 still run correctly. The return instruction is handled almost the same as other branches with a variable destination. The difference is that it is just replaced in the clone by a jump to *DIOTA*, since the target address is already at the top of the stack in this case. The end result is that the amount of extra code that is inserted in the instrumented version of the program does not matter (since the original code is left alone and as such does not have to be relocated) and in most cases the original program cannot notice it is being instrumented. The exception to this last rule are intercepted routines from dynamic libraries, which are described later.

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. Finally, this also conflicts with programs that wish to use this instruction and the associated signal handler for their own purpose.

Multithreading support has been realized by intercepting the `pthread_create()` function. The replacement function generates a new trampoline, which passes the start address of the function that the new thread has to execute to *DIOTA*, and then calls the original `pthread_create()` function with the address of this trampoline as argument. This means that as soon as the new thread is started, *DIOTA* will be entered through the trampoline and it can start instrumenting the necessary code, since it knows the address of the function that had to be executed. The only requirement for this technique to work is that the *DIOTA* library must be thread-safe, which it is. The main difficulty here was to determine which externally used libraries were re-entrant, since this property is not always documented very well.

Signal handlers are caught in a similar way to threads: the `sigaction()` function is intercepted and when a program tries to install a new handler, this handler is replaced by a trampoline that passes the address of the original handler to *DIOTA*. The address of a previous handler returned by the `sigaction()` routine is checked and if it is the address of a a previously installed stub, the address of the real handler is passed on to the program instead. The signal support allows *DIOTA* to instrument C++ exceptions coming from events such as segment violations as well.

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-

[2] This is not yet implemented.

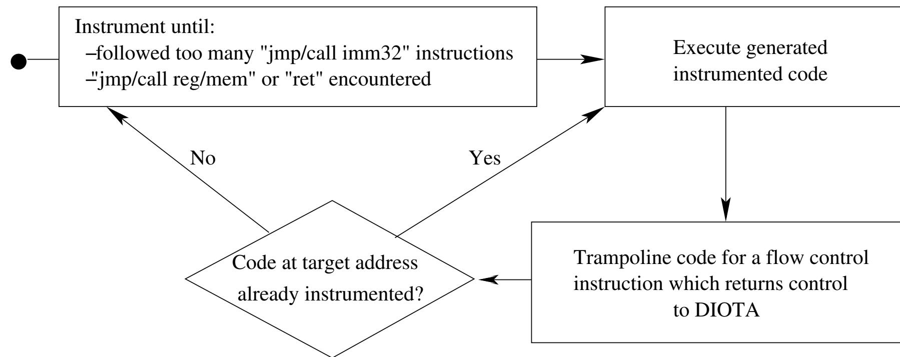

Fig. 1. General overview of *DIOTA*'s operation

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. For parallel programs, the thread identifier is also provided.

- *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. In this context, a basic block denotes a sequence of instructions which is guaranteed to have been executed from the first till the last instruction when the instrumentation routine is called. It is possible that in the original program, these instructions do not form a basic block, because there may be a control flow instruction somewhere that jumps to the middle of this sequence. When this instruction is encountered by *DIOTA* later on, it will generate a new instrumented block starting at the target of that branch instead of jumping to the middle of the previously generated block, as it would do in other modes. This means then when code instrumentation is on, generally a lot more instrumented code blocks are generated.
- *Interception of function calls DIOTA* allows the interception of calls to routines in dynamically linked libraries used by the program. This interception includes calls from the main program, from other libraries and direct calls from within the library that contains the function itself. In the future, *DIOTA* will also be able to intercept statically linked function calls, either in a library or in the main program. Of course, this will require the library or program to contain symbol information.
- *Optimization* This is an attempt to perform dynamic optimization of a program and the libraries it uses, both using static (addresses which are not yet known at compile or link time can be calculated here) and dynamic (inlining and optimizing of code which is executed a lot) techniques.

Memory instrumentation obviously can be used to keep track of all memory accesses done by a program. This is not only useful to record traces, but also to perform data race detection in multi-threaded programs, provided the synchronization routines are also intercepted. When it is combined with the interception of memory management routines, it can function as an advanced debugging heap manager as well.

Code instrumentation is most useful to find out which code of a program and the libraries is executed during a particular run. As mentioned previously, the interception of routines is mostly combined with other functionality, though it can also provide an easy way to perform simple profiling by counting the number of times a certain function is called, as long as this function resides in a dynamically loaded library.

Finally, the optimization feature is normally used on its own, since the objective in this case is improved performance and the other functionality only slows down the adapted program. It is the least mature of all modes and currently does not manage to achieve any speed-ups yet. In the best case, it causes a 5-10% slowdown at this time. The main reason is that no optimizer has been implemented yet to optimize the code traces [BDB00] which are selected during the execution.

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. More information on this topic is presented in section III.

C. *Instrumentation*

A general overview of the data structures used by *DIOTA* can be found in Figure 2. As shown in that figure, *DIOTA* keeps track of the addresses in the original program at which it has already processed code, so in case another instruction jumps to that place, the generated code can be reused.

DIOTA has been implemented for IA32 systems running Linux and can be downloaded from `http://www.elis.rug.ac.be/~ronsse/diota/`. It is a dynamic library that can be attached to a program using the `LD_PRELOAD` environment variable. If this variable is set, the dynamic linker

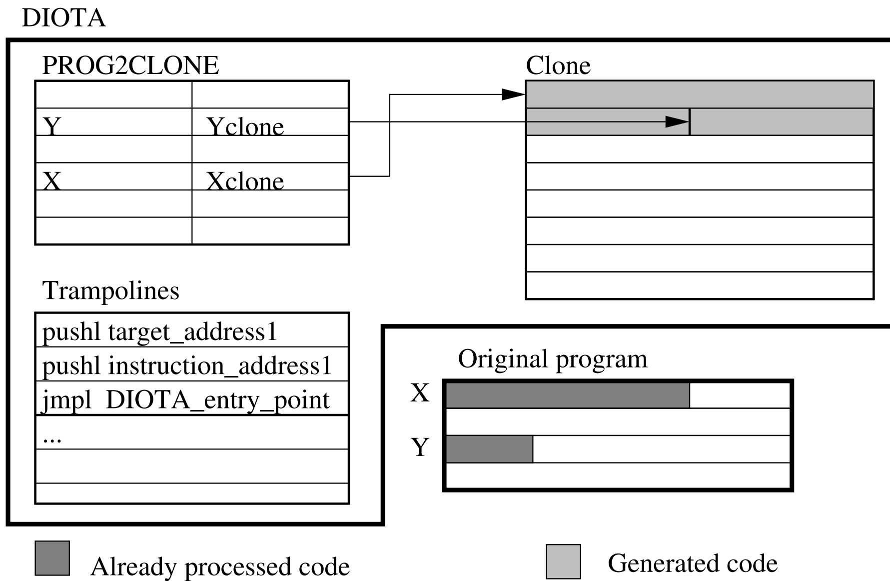

Fig. 2. General overview of the used data structures

will load the designated library although the application is not explicitly linked against it.[3] 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 first instructions of the program. In this context, disassembling means deciding what each individual instruction means, but not performing a sophisticated analysis, e.g. building a control flow graph. Also note that code is instrumented if *DIOTA* detects that it will probably be executed in the near future, therefore data is not instrumented (if an application generates code on the fly it will be instrumented though).[4] During this instrumentation phase, 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 control flow 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 direct branches are followed to their 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.

Conditional branches are more tricky. In this case, a decision must be made about whether or not to follow them and care must be taken so that when the other path is followed, the appropriate instrumented code gets executed (and first generated, if necessary). Currently, *DIOTA* always follows the fall-through path.[5] The second problem is solved by using the construct presented in Figure 3. This is only necessary if the code at the target address of the conditional jump has not been instrumented earlier, otherwise just a `jcc .LprevGeneratedCode` is added.

The first time the conditional branch is taken, the `jcc_taken_handler_` is entered, which links back to *DIOTA* and allows it to process the code at the target of the original `jcc`. It uses the return address on the stack to get the address of the original `jcc`, which is placed in the code right after the call. A better way might be to simply push this address on the stack though. Once it is done generating

[3] Note that this is the only prerequisite for *DIOTA*: the program should be dynamically linked (this covers almost all applications though; only a few critical programs that run at boot time are statically linked). The application can be compiled and linked in the usual way and the final executable can be stripped as no debugging or symbol information is used.

[4] Neither the initialization nor finalization code of libraries can be instrumented currently.

[5] Although one could contend that this is not very efficient in the case of certain kinds of loop constructs emitted by compilers, this simplifies the instrumentation engine and avoids code bloat by minimizing loop unrolling. An optimizer module (not backend) can later on rectify this situation.

```
  jcc    .Ltaken
  jmpl   .LnotTaken
.Ltaken:
  call   jcc_taken_handler_
  .long address_of_org_jcc_instruction
.LnotTaken:
```

Fig. 3. Code in the clone for jcc instructions

code, it adapts the target of the `jcc .Ltaken` instruction so that it points to this newly generated code. This way, the `jcc_taken_handler_` will be executed only once per conditional jump.

The last control flow instructions with an immediate operand are the `loop` and `jecxz` instructions. Figure 4 shows how they are handled. There are two reasons for this special construct. One is that that these instructions are limited to 8 bit displacements and due to the insertion of instrumentation code the target can lie beyond that limit. Another reason is that it is possible that at the moment such an instruction is processed, the code at its target address has not been encountered yet. In that case, a trampoline will be put in place of the `jmpl .LloopStart`. This is not very efficient, but then again, neither is the `loop` nor the `jecxz` instruction on modern processors.

```
  loop .Lloop
  jmpl .LloopDone
.Lloop:
  jmpl .LloopStart
.LloopDone:
```

Fig. 4. Code in the clone for loop instructions

The target address of control flow instructions with absolute targets (such as `ret, jmp %reg`) cannot be used directly, since in that case the execution would return to the original code and *DIOTA* would lose control over the program. After all, the calculated address is independent of the instrumentation of the program (otherwise, the program could notice it is being instrumented), so it will lie somewhere in the original code instead of in the clone. These instructions can also not be evaluated once and then replaced with a direct jump to somewhere in the clone, since every time they are executed, the target address can be different.

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. In the case of a `call`, an additional push of the return address is inserted, while for a `ret` the return address must still be removed from the stack. As an example, the code generated for a `jmp *mem` is shown in Figure 5. After such an instruction has been encountered and processed, *DIOTA* stops instrumenting and jumps to the start of the block instructions it has just generated, since it cannot know where the program will continue afterwards.

```
  pushl  $org_instruction_address
  pushl  target_address
  jmpl   jmp_mem_handler_
```

Fig. 5. Code in the clone for ret instructions

The requirement of the usage of the above construct for all indirect branches, and in particular for `ret` instructions, is responsible for the largest part of the total overhead of *DIOTA*. After a program has run for a while, most code has already been instrumented, but for each execution of such an instruction a hash table lookup is required to get the target address in the clone, which requires much more time than executing the instruction that is replaced.

What is also interesting, is that the current version of *DIOTA* is able to deal with all contemporary user mode instructions found on IA32 architecture implementations, such as MMX, MMX2, SSE, 3DNow!, ... opcodes. *DIOTA* has no knowledge of instructions that require special privileges (e.g. kernel mode). This is however no problem as these instructions cannot be used by applications, only by the kernel. *DIOTA* can also be easily adapted to future architecture extensions thanks to its table-based instruction analyzer. The structure of these tables is shown in Figure 6. They are indexed using the first byte of the opcode, after which the function whose address is stored at that location is called with the arguments in the same row (no argument means "0"). The *extended0F* function uses another table indexed by the second byte of the opcode, since there are many opcodes starting with that byte. All functions have access to generic information about the current instruction, such as its address, through global variables.

D. Interception

DIOTA supports intercepting any routine imported from a dynamically linked library, whether it is used directly in the target program or only by libraries it links to. This is possible thanks to the fact that a shared library is only loaded once in memory and as such checking against one address per intercepted routine is enough to catch all calls to it. Before explaining the way the interception works, a small introduction to the way dynamic libraries work under Linux on the IA32 architecture may be appropriate. First, the situation of the applications that use them is described.

Since it is impossible to determine at link time where in memory a shared library will be loaded, it is impossible to resolve the references to symbols at that stage, especially since those libraries are external and as such can be upgraded separately from the program. Furthermore, since multiple programs may use a shared library at the same time, this relocation cannot even be done at load time, since the absolute addresses will be different for each program: all programs share the same code,

opcode	function	param1	param2	param3
0x00	check_mod	IMM_LEN(0)	MEM_LEN(1)	TYPE(MODIFY)
0x01	check_mod	IMM_LEN(0)	MEM_LEN(4)	TYPE(MODIFY)
0x02	check_mod	IMM_LEN(0)	MEM_LEN(1)	TYPE(LOAD)
0x03	check_mod	IMM_LEN(0)	MEM_LEN(4)	TYPE(LOAD)
0x04	no_mem	INSTR_LEN(2)	IMM_LEN(1)	
0x05	no_mem	INSTR_LEN(5)	IMM_LEN(4)	
...				
0x0f	extended0F			
...				

Fig. 6. Structure of the tables used by the instruction analyzer

but it is almost always mapped at a different address, and they all get a private copy of the data.

```
.PLT0:
  pushl GOT+4    // library identifier
  jmpl  *(GOT+8) // resolution routine
.PLT1
  jmpl  *(GOT+Function1Index)
  pushl Function1ID
  jmpl  .PLT0
.PLT2
  jmpl  *(GOT+Function2Index)
  pushl Function2ID
  jmpl  .PLT0
...
```

Fig. 7. Structure of the PLT of an application

One of the consequences is that when calling a routine in a shared library from inside a program, some level of indirection is necessary. This layer is provided by the so-called *Procedure Linkage Table* (PLT). This is a block of code that has the structure shown in Figure 7 (*DIOTA* will have to be adapted if the layout of the PLT changes. This event is however unlikely to take place in the near future). For each exported function that is used by the program, an entry is provided in this block. The PLT is added to the application and every call to a function imported from a dynamic library has its target set to the appropriate entry inside this PLT.

When the program is started, the entries for all functions in the *Global Offset Table* (GOT), which is part of every private copy of the data, are initialized with the address of the `push` coming after the respective `jmp *(GOT+...)` instructions in the PLT. The result is that the first time such a function is called, the `jmp *` will simply jump to the next instruction, the index of the called function will be placed on the stack together with the identifier for the shared library and then the dynamic linker is called. This last one will then lookup where in memory the routine is located, update its entry in the GOT with that address and then transfer control to this function. The next time this same function is called, the `jmp *` will immediately jump to the routine instead of calling the dynamic linker once again.

In the loaded ELF image of the program, the PLT entries all have an attribute that contains the name of the imported function. Using this information, *DIOTA* can replace the addresses stored in the GOT of functions, specified by name, with user-defined ones and thus implement the basic interception. Before all calls to such a functions can be intercepted however, more work must be done.

The first problem are direct calls to this routine from within the shared library the function is located in. Since the target address of a direct call is known while it is being instrumented, we can compare this address with the addresses of the functions that must be intercepted. However, the latter addresses are not necessarily known at the time such a call is encountered, since if the function has not been called yet from the program at that point, its GOT entry still contains the address of the `push` instruction in its PLT code. To get around this problem, the environment variable `LD_BIND_NOW=YES` must be set. This instructs the dynamic linker to resolve all GOT entries for functions when the program is started. The reason this is not the default, is that this prolongs the startup time of a program and that it is possible that not all imported routines will be called during a particular run of a program. With the default behavior, only the functions that are actually used are resolved.

The second problem occurs when a backend wants to intercept a routine that is not used directly in the program, but which used (internally) by one or more libraries the program links to. In this case, the address will not be found in the program's PLT. The solution that was chosen here, is that the backend in that case must link to the library that contains the function it wants to intercept and then pass the address of that function to *DIOTA*. Since *DIOTA*, the backend and the program to be instrumented all run in the same address space, this address will be the same address that will be used during the program's execution.

The third problem is the hardest: calls to a function inside one shared library from inside another shared library. The reason is that although such calls also happen via a PLT, this PLT (and the associated GOT) is part of the global data of the calling library and not of the program that uses it (possibly indirectly), since every time a shared library is upgraded, its PLT changes with it. As mentioned previously, every running program gets a private copy of this data. Since the code is shared between

all running programs, the only way to address this data is by using addresses that are relative to the code. After all, the distance between two addresses in a library, whether they point to code or data, is always the same. Such code that addresses data with offsets relative to itself is called *Position Independent Code* (PIC).

```
call   .+5               call   helper
popl   %ebx              addl   $offset,%ebx
addl   $offset,%ebx      ...

                       helper:
                         movl   (%esp),%ebx
                         ret
```

Fig. 8. Calculate the value of the instruction pointer

Since it is impossible to read the instruction pointer on the IA32 architecture, one has to use one of the code snippets presented in Figure 8. The leftmost code was used in the past, but nowadays it is more efficient to use the other sequence. The reason is that modern IA32 processors contain an internal hardware stack which keeps track of the return addresses of `call`'s, so that they can better predict the targets of `ret` instructions. The old sequence messes up this stack, since there's a `call` without a `ret`, but both snippets are functionally equivalent. The used offset is calculated at link time so that afterwards, `ebx` contains the start address of the GOT of the library.

This piece of code can be found at the start of any shared library routine that uses a static or global variable from this library, or which calls a routine imported from another library. The PLT now looks like Figure 9. It is almost the same as the PLT of the program, except that all used addresses are now relative to `ebx`. To still be able to check for intercepted routines at instrumentation time instead of at execution time, *DIOTA* has to keep track of the contents of the `ebx` register. This is done by passing the value of the `ebx` register to *DIOTA* from all trampolines and by calculating new values assigned to it using the instruction sequences presented in Figure 8.

```
.PLT0
  pushl  0x4(%ebx)  // library identifier
  jmp    *0x8(%ebx) // resolution routine
.PLT1
  jmpl   *(%ebx+Function1Index)
  pushl Function1ID
  jmpl   .PLT0
.PLT2
  jmpl   *(%ebx+Function2Index)
  pushl Function2ID
  jmpl   .PLT0
...
```

Fig. 9. Structure of the PLT of an application

With this final modification, *DIOTA* is able to intercept all direct calls to shared library routines with little overhead. Note that the above solution will not catch indirect calls to such routines (e.g. using function pointers). It may also fail in case manually coded assembler routines are used which modify `ebx` in another way or in case the PLT calls would use another base register (which is very unlikely, since the PLT is generated by the linker). Although these problems can be solved, their solution does not come for free.

The indirect calls can be caught by modifying *DIOTA* so that when it is instructed to lookup the instrumentation code corresponding to the target of a branch with a variable argument, it first compares this target with any routines that should be intercepted. This would cause quite a noticeable slowdown however, since then these comparisons would have to be done every time such an instruction is executed. The solution for the `ebx` value problem is to either implement an emulation engine inside *DIOTA* which really simulates the modification of at least this register while it processes instructions, or by replacing the calls to these dynamic PLT entries with some trampoline code which can then use the actual value of the `ebx` register at the moment the call is made. None of these suggested solutions has been implemented yet due to the fact they have not been needed yet.

E. Limiting the memory consumption

Given the description about the way *DIOTA* instruments programs on the fly, it should be clear that a lot of memory will be consumed. Indeed, the executed instructions will be placed twice in memory, instrumentation is added, loops are unrolled and function calls are inlined (in a limited way), stubs are created, . . .

The total amount of memory consumed can grow very rapidly during an execution, especially if memory operations are instrumented: in that case, a simple `mov mem,reg` results in 20 bytes being added. However, this does not prevent us from running large programs as *DIOTA* can garbage collect the code it created. Initially *DIOTA* allocates a large amount of memory it will use for the code it creates. If *DIOTA* runs out of space, it simply throws away all the code it already generated, releases the allocated memory, allocates a new block and starts instrumenting again, *from this point in the execution instead of from the start of the execution*. In order to accommodate programs with varying working sets of instructions, the newly allocated block of memory can be smaller, equal in size or larger than the previous one, depending on the amount of time that passed since the last allocation.

Note that this scheme naturally adapts to programs with changing working sets: e.g. after a initialization phase a program can start executing a totally different set of functions, to which *DIOTA* can react by throwing away the already instrumented (and now obsolete) instructions.

F. Optimization

When doing optimizations at runtime, it is crucial to decide which parts of the code shall be optimized, since the gain from those transformations must outweigh the cost of doing them. The granularity at which this selection occurs (per procedure, per basic block, per instruction, . . .) is another important factor. Finally, the detail of the profiling information that is collected also plays an important role, since more information can result in better optimizations, but its gathering slows down the execution of the program.

The first step of the optimization process is determining the so-called *hot spots* of the code and with how much precision they should be pin-pointed. Since *DIOTA* is always entered when a branch with a variable target address is taken (since the target address must be looked up), this lookup routine was deemed to be a good place to collect some profiling information without causing to much extra overhead. *DIOTA* keeps track of the "favorite" target addresses of such instructions and the number of times the favorite target address was the actual target. Per target address, *DIOTA* also keeps track of which instruction jumps to it the most, which will be useful later on for determining when to stop following a path through the code.

```
  jcc    .Ltaken
  jmpl   .LnotTaken
.Ltaken:
  call   jcc_taken_handler_
  .long  address_of_org_jcc_instruction
.LnotTaken:
  movb   $1,jcc_taken_table[constant_index]
```

Fig. 10. Code in the clone for jcc instructions

DIOTA also profiles conditional jumps. One way is inherent to the way conditional jumps are handled: if a conditional jump has never been taken yet at the moment it is examined for inclusion in a block, its counterpart in the instrumentation code will jump to a call to the `jcc_taken_handler_`. This strategy does not allow *DIOTA* to check whether a conditional jump has never been "not taken" however, so for this purpose the code presented earlier in Figure 3 is changed to that from Figure 10.

No counting of the number of times a conditional jump is (not) taken occurs, because this would require the saving and restoring of the flags, causing a significant slowdown. The simple `mov` instruction which is used now on the other hand, does not make the program noticeably slower and provides enough information to the optimizer to take an appropriate decision regarding whether or not a conditional jump should be followed. What is also interesting about this construct, is that no separate table holding the relationship between a particular conditional jump and a certain *jcc_taken_table* entry is necessary, because the optimizer can get the appropriate address straight out of the `mov` instruction.

Before the optimizing can start, first a block of code, which preferably consists of several basic blocks of the original program and of dynamic libraries it uses, must be generated. *DIOTA* starts with the selection of such a block (called a *trace* in [BDB00]) when inside the lookup routine the usage count of a favorite address of a branch with a variable target is deemed high enough. The block starts at this target address and then successive instructions are added from the most executed path. This means that indirect branches are followed to their favorite destination. If only one of the paths following a conditional jump has been executed until now, that path is chosen, otherwise backward jumps are followed the first time they are encountered and forward ones are not, with this decision being reversed the second time such an instruction is encountered in the same optimization block. This continues until one of the following conditions is met.

- An instruction with a variable target address is encountered, but it does not have a favorite target, or its usage count is too low.
- The current block can be linked to a previously generated optimized block.
- Too many direct or indirect branches have been followed to their (favorite) destination (avoid endless loops, again).
- Previously, an indirect jump was followed to its favorite destination, but this destination address is more often jumped to by another branch and the optimizer has arrived at yet another indirect branch.

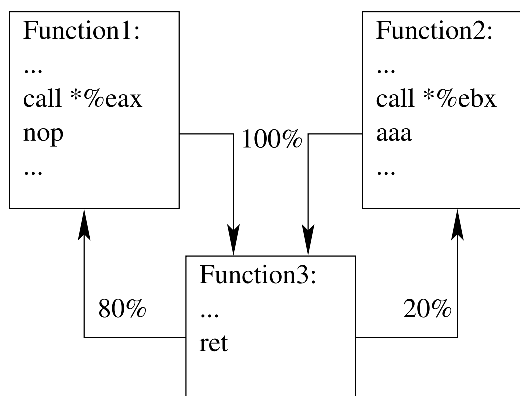

Fig. 11. Usage of the relationship between target addresses and jumps.

An example of the last case is shown in Figure 11. `Function1` and `Function2` both contain an indirect call to `Function3`. Assuming that `Function3` is only called from these two places and that `Function1` is executed four times as much as `Function2`, the majority of the calls comes from the former.

After a while, the profiling information will show that the calls in both `Function1` and `Function2` most of the time jump to the start of `Function3`, but the information about this target address will indicate that most jumps come from within `Function1`. The favorite target address of the `ret` instruction at the end of `Function3` will also be the instruction coming after the call in `Function1`.

Now, when following a path starting from within `Function2`, the `ret` at the end of `Function3` will not be followed back to `Function1`, but the path will simply end at that instruction thanks to the target-source relationship information that is gathered. However, when starting inside `Function1`, the `ret` does get followed back. If for some reason an incorrect prediction is made and a wrong instruction is followed, the resulting code will still be correct, but it will simply take up space and never be executed. This safeguard is provided by the code shown in Figure 12, which is inserted every time an indirect branch is followed to its most likely target.

```
pushf
cmpl  $expected_target,actual_target
jne   .LexitStub
popf
```

Fig. 12. Compare expected and actual target of indirect branches

As mentioned earlier, a real optimizer has not yet been implemented. Basically, the optimization was added in order to make the slowdown caused by *DIOTA* and the backends less annoying. Currently, only one optimization is performed: often executed paths through the code are organized in contiguous blocks. In that process, indirect branches are replaced by a combination of a comparison of their current target with the most likely one and a conditional jump to the original instrumented code in case it was mispredicted. These blocks have only one entry-point, some inefficiencies of the regular instrumentation code (such as the constructs required for conditional jumps) are eliminated and a few small optimizations are performed (e.g. light loop unrolling). The advantage of *DIOTA*'s approach over Dynamo's [BDB00] is that unoptimized code is not emulated, but executed by the underlying processor. Hence, it should be possible to achieve better results for a broader range of applications.

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. As an example, Figure 13 shows a very simple backend that instructs *DIOTA* to instrument all data accesses and to call the provided `trace_data()` function each time data is accessed.

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 (the `pthread` functions) 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.
- 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. 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

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

```
#include "diota.h"

static void trace_data(unsigned ip, unsigned type, unsigned len, unsigned address){
  fprintf(diota_tty, "%.8x %d %d %.8x\n", ip, type, len, address);
}

static init_trace_data(){
  diota_callback("#DATA#", 0, trace_data);
}

INIT(init_trace_data);
```

Fig. 13. A simple backend.

[BDCW91] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. Proteus: A high performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[BG91] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206, Santa Cruz, CA, May 1991.

[BL92] T. Ball and R. Larus. Optimally profiling and tracing programs. Conference Record of the 19th ACM Symposium on Principles of Programming Languages, pages 59–70, 1992.

[CK90] Bob Cmelik and David Keppel. A fast instruction-set simulator for execution profiling. volume ACM SIGMETRICS, pages 128–137, May 1990.

[DBD96] Koen De Bosschere and S. Debray. `alto`: a Link-Time Optimizer for the DEC Alpha. Technical Report TR 96-15, Computer Science Department, University of Arizona, 1996.

[DGH91] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1991 Conference on Parallel Processing*, pages II–99–II–107, Augustus 1991.

[EKKL90] S. Eggers, D. Keppel, E. Koldinger, and H. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 8, May 1990.

[GH91] A. Goldberg and J. Hennessy. Performance Debugging of Shared Memory Multiprocessor Programs with MTOOL. In *Proceedings of Supercomputing '91*, pages 481–490, November 1991.

[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.

[HMG+97] J. Hollingsworth, B. P. Miller, M. Goncales, O. Naim, Z. Xu, and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proceedings of Intl. Conference on Parallel Architectures and Compilation Techniques*, San Fransisco, July 1997.

[LRV+] Dennis Lee, Ted Romer, Geoff Voelker, Alec Wolman, Wayne Wong, Brad Chen, Brian Bershad, and Hank Levy. Instrumentation and Optimization of WIN32/Intel Executables. URL= `http://etch.cs.washington.edu/`.

[LS96] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. SIGPLAN Conference on Programming Language Design and Implementation, June 1996.

[Mal87] A. Malony. Program tracing in cedar. Technical Report 660, Center for Supercomputing Research and Development, University of Illinois, April 1987.

[MCC+95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Tools. *IEEE Computer*, 28(11):37–44, November 1995. Special issue on performance evaluation tools for parallel and distributed computer systems.

[MCK+90] B. Miller, M. Clark, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, (2):206–217, April 1990.

[MDG+98] Peter S. Magnusson, Fredrik Dahlgren, Hkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenstrm, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. 6 1998. USENIX '98 Annual Technical Conference.

[MN89] A. Mink and G. Nacht. Performance measurement of a shared-memory multiprocessor using hardware instrumentation. Proc. of the 22nd Hawaii Int. Conf. on System Sciences, pages 267–276, January 1989.

[Par96] ParaSoft Corporation. Insure++: Automatic runtime debugger, 1996.

[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.

[SE94] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. Research report 94/2, Digital-WRL, 1994.

[Sew02] Julian Seward. The design and implementation of Valgrind. http://developer.kde.org/∼sewardj/docs/techdocs.html, March 2002.

[SJF92] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs. Address tracing of parallel systems via trapeds. *Microprocessors and Microsystems*, 16(5):249–261, 1992.

[Sun94] SunSoft. lock_lint User's Guide, 1994.

[SW93] A. Srivastava and D. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Language*, 1(1):1–18, March 1993.

[TFCB90] Jeffrey J. P. Tsai, Kwang-Ya Fang, Horng-Yuan Chen, and Yao-Dong Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. In *IEEE Transactions on Software Engineering*, volume 16, pages 897–916. August 1990.

[TM99a] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Third Symposium on Operating Systems Design and Implementation*, pages 117–130, New Orleans, February 1999.

[TM99b] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *Parallel Processing Letters*, 1999.

[VF93] Jack E. Veenstra and Robert J. Fowler. Mint tutorial and user manual. Technical Report 452, The University of Rochester, Computer Science Department, Rochester, New York, November 1993.