Self-modifying code: instrumentation challenges

Jonas Maebe, Michiel Ronsse, Koen De Bosschere

Abstract— **Adding small code snippets at key points to existing code fragments is called instrumentation. It is an established technique to debug certain otherwise hard to solve faults, such as memory management issues and data races. Dynamic instrumentation can already be used to analyse code which is loaded or even generated at run time. With the advent of environments such as the Java Virtual Machine with optimizing Just-In-Time compilers, a new obstacle arises: self-modifying code. In order to instrument this kind of code correctly, one must be able to detect modifications and adapt the instrumentation code accordingly. In this paper we propose an innovative technique that uses the hardware page protection mechanism of modern processors to detect such modifications. We also show how an instrumentor can adapt the instrumented version depending on the kind of modificiations as well as an experimental evaluation of said techniques.**

Keywords— **dynamic instrumentation, self-modifying code, instrumenting JVMs**

I. Introduction

THE Instrumentation is a technique whereby existing code is modified in order to observe or modify its behaviour. It has a lot of different applications, such as profiling, coverage analysis and cache simulations. One of its most interesting features is however the ability to perform automatic debugging, or at least assist in debugging complex programs. After all, instrumentation code can intervene in the execution at any point and examine the current state, record it, compare it to previously recorded information and even modify it.

The instrumentation can occcur at different stages of the compilation or execution process. When performed prior to the execution, the instrumentation results in changes in the object code on disk, which makes them a property of a program or library. This is called static instrumentation. If the addition of instrumentation code is postponed until the program is loaded in memory, it becomes a property of an execution. In this case, we call it dynamic instrumentation.

When using dynamic instrumentation, the original code is read from memory and can be instrumented just before it is executed, so even dynamically loaded and generated code pose no problems. However, when the program starts modifying this code, the detection and handling of these modifications is not possible using current instrumentation techniques. Yet, being able to instrument self-modifying code becomes increasingly interesting as run time systems that exhibit such behaviour, such as Java Virtual Machines, gain more and more popularity.

Even when starting from a system that can already instrument code on the fly, supporting self-modifying code is a quite complex undertaking. First of all, the original program code must not be changed by the instrumentor, since otherwise the program's own modifications may conflict with these changes later on. Secondly, the instrumentor must be able to detect changes performed by the program before the modified code is executed, so that it can reinstrument this code in a timely manner. Finally, the reinstrumentation itself must take into account that an instruction may be changed using multiple write operations, so it could be invalid at certain points in time.

In this paper we propose a novel technique that can be used to dynamically instrument self-modifying code with an acceptable overhead. We do this by using the hardware page protection facilities of the processor to mark pages that contain code which has been instrumented as read-only. When the program later on attempts to modify instrumented code, we catch the resulting protection faults which enables us to detect those changes and act accordingly. The described method has been experimentally evaluated using the *DIOTA* (Dynamic Instrumentation, Optimization and Transformation of Applications [**?**]) framework on the Linux/x86 platform by instrumenting a number of JavaGrande [**?**] benchmarks running in the Sun 1.4.0 Java Virtual Machine.

The paper now proceeds with how the detection of modified code is performed and how to reinstrument this code. We then present some experimental results of our implementation of the described techniques and wrap up with the conclusions and our future plans.

II. Dynamic instrumentation

Dynamic instrumentation can be be done in two ways. One way is modifying the existing code, e.g. by replacing instructions with jumps to routines which contain both instrumentation code and the replaced instruction [**?**]. This technique is not very usable on systems with variable-length instructions however, as the jump may require more space than the single instruction one wants to replace. If the program later on transfers control to the second instruction that has been replaced, it will end up in the middle of this jump instruction. The technique also wreaks havoc in cases of data-in-code or code-in-data, as modifying the code will cause modifications to the data as well.

The other approach is copying the original code into a separate memory block (this is often called *cloning*) and adding instrumentation code to this copy [**?**], [**?**], [**?**]. This requires special handling of control-flow instructions with absolute target addresses, since these addresses must be relocated to the instrumented version of the code. On the positive side, data accesses still occur correctly without any special handling, even in data-in-code situations.

The reason is that when the code is executed in the clone,

J. Maebe, M. Ronsse and K. De Bosschere are with the Department of Electronic Informations Systems, Ghent University (RUG), Gent, Belgium. E-mail: jmaebe|ronsse|kdb@elis.ugent.be .

J. Maebe is sponsored by IWT Flanders

[width=8.5 cm]exception

Fig. 1. Exception handling in the context of self-modifying code support

only the program counter (PC) has a different value in an instrumented execution compared to a normal one. This means that when a program uses non-PC-relative addressing modes for data access, these addresses still refer to the original, unmodified copy of the program or data. PC-relative data accesses can be handled at instrumentation time, as the instrumentor always knows the address of the instruction it is currently instrumenting. This way, it can replace PC-relative memory accesses with absolute memory accesses based on the value the PC would have at that time in a uninstrumented execution.

III. Detecting modifications

There are two possible approaches for dealing with code changes. One is to detect the changes as they are made, the other is to check whether code has been modified every time it is executed. Given the fact that in general code is modified far less than it is executed, the first approach was chosen. Once a page contains code that has been instrumented, it will be write-protected using the hardware page protection facilities of the processor. The consequence is that any attempt to modify such code will result in a segmentation fault. An exception handler installed by *DIOTA* will intercept these signals and take the appropriate action.

Since segmentation faults must always be caught when using our technique to support self-modifying code, *DIOTA* installs a dummy handler at startup time and whenever a program installs the default system handler for this signal (which simply terminates the process if such a signal is raised), or when it tries to ignore it. The resulting way of handling exception in *DIOTA* is shown in Figure 1.

Whenever a protection fault occurs due to the program trying to modify some previously instrumented code, we make a copy of the accessed page, then mark it writable again and let the program resume its execution. This way, it can perform the changes it wanted to do itself. After a while, the instrumentor can compare the contents of the unprotected page and the the buffered copy to find the changes.

The question then becomes: when is this page checked for changes, how long will it be kept unprotected and how many pages will be kept unprotected at the same time. All parameters are important for performance, since keeping pages unprotected and checking them for changes requires both processing and memory resources. The when-factor is also important for correctness, as the modifications must be incorporated in the clone code before it is executed again.

On architectures with a weakly consistent memory model (such as the SPARC and PowerPC), the program must make its code changes permanent by using an instruction that synchronizes the instruction caches of all processors with the current memory contents. These instructions can be intercepted by the instrumentation engine and trigger a comparison of the current contents of a page with the previously buffered contents. On other architectures, heuristics have to be used depending on the target application that one wants to instrument to get acceptable performance and correct behaviour.

IV. Handling modifications

The optimal way to handle the modifications, is to re-instrument the code in-place. This means that the previously instrumented version of the instructions in the clone are simply replaced by the new ones. This only works if the new code has the same length as (or is shorter than) the old code however, which is not always the case.

A second way to handle modifications can be applied when the instrumented version of the previous instruction at that location was larger than the size of an immediate jump. In this case, it is possible to overwrite the previous instrumented version with a jump to the new version. At the end of this new code, another jump can transfer control back to rest of the original instrumentation code.

Finally, if there is not enough room for an immediate jump, the last resort is filling the room originally occupied by the instrumented code with breakpoints. The instrumented version of the new code will simply be placed somewhere else in the code. Whenever the program then arrives at such a breakpoint, *DIOTA*'s exception handler is entered. This exception handler has access to the address where the breakpoint exception occurred, so it can use the translation table at the end of the block to look up the corresponding original program address. Next, it can lookup where the latest instrumented version of the code at that address is located and transfer control there.

V. Experimental evaluation

Table I shows the measured timings when running a number of tests from sections 2 and 3 of the sequential part of the JavaGrande benchmark v2.0 [**?**], all using the SizeA input set. The first column shows the name of the test program. The second and third columns show the used cpu time (as measured by the `time` command line program, expressed in seconds) of an uninstrumented resp. instrumented execution, while the fourth column shows the resulting slowdown factor.

The fifth column contains the the amount of protection faults divided by the uninstrumented execution time, so it is an indication of the degree in which the program writes to pages that contain already executed code. The last column shows the number of lookups per second of uninstrumented execution time, where a lookup equals a trip to *DIOTA* to get the address of the instrumented code corresponding to the queried target address. The results have been sorted on the slowdown factor.

Regression analysis shows us that the overhead due to the lookups is nine times higher than that caused by the protection faults (and page compares, which are directly correlated with the number of protection faults, since every page is compared N times after is unprotected due to a fault). This means that the page protection technique has

Program name	Normal execution (s)	Instrumented execution (s)	Slowdown	Relative # of protection faults	Relative # of lookups
FFT	40.28	95.86	2.38	2305	409609
SparseMatmult	24.29	91.09	3.75	3751	874669
HeapSort	5.25	41.03	7.82	14779	1700553
Crypt	8.91	175.15	19.66	12845	6696704
RayTraceBench	28.87	652.11	22.59	6611	8026878

TABLE I
TEST RESULTS FOR A NUMBER OF SEQUENTIAL JAVAGRANDE 2.0 BENCHMARKS

a quite low overhead and that most of the overhead can be attributed to the overhead of keeping the program under control.

The cause for the high cost of the lookups comes from the fact that the lookup table must be locked before it can be consulted, since it is shared among all threads. We have to disable all signals before acquiring a lock since otherwise a deadlock might occur (in case the thread that holds the lock receives a signal from another thread). Disabling and restoring signals is an extremely expensive operation under Linux, as both operations require a system call.

VI. Conclusions and future plans

We have described a method which can be used to successfully instrument an important class of programs that use self-modifying code, specifically Java programs run in an environment that uses a JiT-compiler. The technique uses the hardware page protection mechanism present in the processor to detect modifications made to already instrumented code. Additionally, a number of optimisations have already been implemented to reduce the overhead, both by limiting the number of protection faults that occurs and the number of comparisons that must to be performed.

In the near future, a number of extra optimisations will be implemented, such as keeping more than one page unprotected at a time and the possibility to specify code regions that will not be modified, thus avoiding page protection faults caused by data and code being located on the same page. Additionally, we are also adapting the *DIOTA* framework in such a way that every thread gets its own clone and lookup table. This will greatly reduce the need for locking and disabling/restoring signals, which should also result in a significant speedup for the programs that perform a large number of lookups.