Memory Management Replay in DejaVu

Tom Ghesquiere
Ghent University, dept. ELIS
St. Pietersnieuwstraat 41
B-9000 Gent
Tel +32 9 264 3367
Fax +32 9 264 3594
tghesqui@elis.ugent.be

Jong-Deok Choi
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
Tel +1 914 784 7961
Fax +1 914 784 7455
jdchoi@us.ibm.com

Koen De Bosschere
Ghent University, dept. ELIS
St. Pietersnieuwstraat 41
B-9000 Gent
Tel +32 9 264 3406
Fax +32 9 264 3594
kdb@elis.ugent.be

***Abstract*— Multithreading makes it hard to use cyclic debugging techniques, due to the non-determinism related to the concurrent behaviour of such applications. A well known solution consists of recording an execution and debugging it during replay. In order to produce a faithful replay, the recorded information needs to contain enough information to cover all non-deterministic choices. Java has an extra source of non-determinism related to the garbage collector of the JVM. This paper discusses the problems that arise herefrom and describes the solutions we developed in order to implement a fully operational memory management replay module, based on the scheduling replay of DejaVu.**

I. Introduction

DejaVu is a record/replay framework, developed at IBM [CS98], [CAN$^+$01]. In its most recent version it replays an application by recording the points where thread switches were made and forcing them to happen again at the very same points during replay. Furthermore, wallclock values are also recorded and replayed, because these values can also influence thread scheduling. Other forms of nondeterminism, like file-input, are not handled because of efficiency, although functionality to do so is present. Remark however that replaying the thread switch points only works on a single processor.

DejaVu has been developed for, among others, Jikes RVM [Jik02]. This virtual machine is almost entirely written in Java and can therefore benefit from cross-optimization to improve its performance. Cross-optimization means that the runtime environment and the application are analysed together, clearing the road for more intrusive optimization techniques. Co-running a record/replay functionality with such a virtual machine can also benefit from the same performance improvement. This is exactly what DejaVu does. Unfortunately, this means that DejaVu, as part of the bigger picture, also has to behave similarly during record and replay. However, this imposes that DejaVu should replay itself, which is by definition impossible. The rest of the paper presents how to deal with this contradictio in terminis.

II. Problem statement

Cyclic debugging of multithreaded applications is difficult, due to the non-deterministic nature of their concurrent behaviour. A lot of record-replay mechanisms have been introduced to cope with this kind of difficulty [CGC$^+$03]. Known sources of non-determinism are: actions by the scheduler, interrupts, input (e.g. the clock values), and data races. In Java however, there is an additional source of non-determinism, namely the invocation points of the garbage collector in a particular execution. This non-determinism could further influence the application, for instance in the presence of address-based hashing techniques.

There are two ways to deal with the non-determinism of the GC: one could record the execution points at which the garbage collector is invoked, and re-invoke it during replay at the very same points, or one could enforce the application to behave deterministically in accordance to the memory manager, such that the garbage collector will automatically be invoked at the same execution points. Both have their own problems: re-invoking the garbage collector requires that the replay phase does not use more memory than the original one. Furthermore, this method can never guarantee the same memory layout and object references. Deterministically replaying all the memory allocation operations effectively solves this problem, but it is not straightforward either: it must be done for both the application and the instrumentation code. Especially the latter one is difficult since the instrumentation code for recording and replaying is necessarily different.

The rest of the paper presents the difficulties of the latter choice, namely the deterministic memory manager, and the solutions we developed in order to obtain a fully operational memory management replay module.

III. Enforcing identical behaviour

It seems pretty straightforward that code handling the recording phase must be as similar as possible to the code that handles the replay phase. Any inconsistency between them can influence the virtual machine, causing different behaviour during record & replay.

However, the record phase and the replay phase are bound to do different things. For instance, record writes data to a trace file, while replay reads from it. This leads to unavoidable differences in code. Luckily, as stated before, in order to produce a faithful replay, all one needs to do is enforcing the non-deterministic items to behave deterministically. In the case of the scheduler for instance, this means enforcing the thread switches at the same logical execution points, independent of the different instrumentation. In case of the garbage collector, this means allocating the same amount of memory at the same logical execution points, also independently of different instrumentation code. The next sections show how to enforce these rules in some of the most interesting situations.

A. Lazy compilation

Usually in Java, methods are not compiled until their first execution[1]. Executing different code during record & replay leads as a consequence to compiling different methods. For instance reading some trace information from a file uses different methods as writing that same data. As one may expect, this also influences the garbage collector. Different methods will consume a different amount of memory, leading to different invocation points for the garbage collector, or even non-deterministic out-of-memory errors.

Former solutions to this problem were based on eager compilation [CAN$^+$01]. This technique suggests to precompile every method that could be called from the current executed one. Since no contemporary JVM supports this type of JIT-compilation (if this is still JIT after all), and this technique is not efficient at all either, we chose for another solution. We precompile every single method that ever could be called by the instrumentation code, both record and replay. This way, the memory is still filled exactly the same way in both modes, while the inefficiency of unnecesary compiled methods remains restrained to those needed by the instrumentation code, wich is a very small set.

B. Execution frequencies

Optimizing a particular part of the code in terms of speed often results in using extra memory. One particular and often used technique is inlining a method call. An advanced compiler could choose to make this trade-off between space & time only for hot paths, resulting in gaining lots of time while loosing a small amount of space.

To implement such a strategy, the compiler must collect some information about the execution frequencies of the different paths. This could be done at runtime, but to compile a particular method for the first time, guessing these values is the only solution.

Knowing all of this, an inconsistency between record and replay could arise as follows: Suppose one of the gambling techniques is dividing the frequency at every if-test by 2. Also suppose the record mode needs more if-tests than the replay phase. This could result in the code following the recording instrumentation as being cold, while the code following the replaying part to be labeled as hot.

[1]In the presence of an interpreter, even less methods will be compiled

While the hot path through the replay instrumentation continues, it meets a method call to method X, which can nicely be inlined. The same method was seen in the previous recording execution, after following a cold path through the record instrumentation. But in the latter case, the compiler didn't find it necessary to inline, and leaves the method call as is. This finally could result in compilation of method X at different execution points during record or replay, or even worse, in an extra method compilation during the record phase, leading to the same troubles as in the previous section.

The solution we present herefore is to give the control flow graph of the recording code and the replaying code an equal design. This off course follows the general rule that the recording instrumentation and the replaying instrumentation must be as similar as possible.

C. Allocation of objects

In fact the above problem can be generalized. Not only methods need space during a program execution, but also objects and local variables. The problem of local variables was already solved in [CAN$^+$01]. Different instrumentation code could use a different amount of local variables & stack slots. [CAN$^+$01] presented to preallocate the stack with a certain estimated size that certainly covers all needs during both the record and replay phase.

The allocation of objects is another problem however. Using a similar technique as for local variables would be to do a garbage collection before entering the record/replay dependent code. One will certainly notice that this is a very expensive solution, that won't even work.

Executing dummy allocations of the objects that are only used in the other mode may look as a possible solution at first sight, but there is more. Allocations must also be done in the exact same order, especially in relation to the possible thread switch points. Suppose you're 20 kiB away from a new garbage collection. The order of allocating 2 objects of 10 and 30 kiB each can make a big difference for the execution behaviour. Calls to library code, like writing to the tracefile, make this solution even harder, as you cannot insert dummy allocations in library code. The only way out here is to restrict the use of library code to non-allocating methods. As one might notice, this becomes pretty implementation dependent. In our solution however we were not restricted by this demand.

IV. Evaluation

In order to prove that this replay mechanism actually works, we have tested it on several multithreaded benchmark applications. The different execution times of these applications can be found in Table 1.

MTRT is actually _227_mtrt, the multithreaded Ray Tracer from the JVM98 benchmark suite. The mentioned timings in Table 1 is however not the time reported by the JVM98 application, but the total execution time. This is because, as

	original execution (s)	recording execution (s)	recording overhead (%)	replaying execution (s)	replaying overhead (%)
MTRT	26.44	37.67	42.46	37.21	40.72
Monte Carlo	39.47	43.91	11.24	43.58	10.42
Moldyn	170.48	193.99	13.79	193.04	13.23
JBB	1311.02	1345.14	2.60	1346.23	2.69

TABLE I
TIMING MEASUREMENTS

DejaVu replays the wall clock, the replayed benchmark test reports that is has run exactly as long as the recorded one. It doesn't know any better.

Monte Carlo and Moldyn are two multithreaded applications from the Java Grande suite, which we ran both with 10 threads. Here too, we didn't use the reported performance measurements, but the actual execution time. Remark however that the performance slowdown of the record phase resembles pretty much the overhead in execution time.

JBB is the JBB2000 benchmark, ran with standard input parameters, namely 8 warehouses, 30 sec. rampup & 2 min. of transaction time.

As one can see, the overhead significantly drops with longer execution time. DejaVu has a high impact on the compilation time. While most of the applications are pretty small programs, they spend relatively much time compiling, wich explains the high overhead. A small overhead of 2% remains however during large applications.

Although one might think that performance measurements are important, the most important thing is that the recorded execution and the replaying execution actually produce the same results. The above timing measurements are definitely susceptible for optimization, but one thing is not changeable, namely that all applications generate exactly the same output during record and replay. For the Java Grande and the JVM98 benchmarks for instance, this means that they produce exactly the same performance measurements. With JBB, this means the same amount of transactions, thread spread, heap usage, sequence of started warehouses and so on.

V. Conclusions & future work

With this paper, we have revealed some of the difficulties of a record/replay tool in the presence of a garbage collector. For each one, we have presented a solution. We have also shown that our proposals actually work for modern virtual machines, and with an acceptable overhead.

In the future, we should design a debugging platform on top of our record/replay framework. This way one could easily follow variables, place breakpoints and so on. The most difficult obstacle will probably be the influence of the debugger on the virtual machine, as this could also disturb the replayed execution. A good start is already given by [ACN+01].

References

[ACN+01] Bowern Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, John Vlissides, and Hytm-Gyoo Yook. A Debugging Platform for Java Server Applications. Technical Report RC22036, IBM Research Division, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, April 2001.

[CAN+01] Jong-Deok Choi, Bowen Alpern, Ton Ngo, Manu Sridharan, and John Vlissides. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *Proceedings of the 15th IEEE International Parallel & Distributed Processing Symposium*, San Fransisco, April 2001.

[CGC+03] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. De Bosschere. A Taxonomy of Execution Replay Systems. In V. Milutinovic, editor, *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, pages CD–ROM paper 59, L'Aquila, 7 2003.

[CS98] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Welches, Oregon, August 1998. ACM Press.

[Jik02] The Jikes^{TM} Research Virtual Machine, User's Guide, December 2002.