Database Management for a Trace Oriented Debugger

Introduction

Table of Contents

❖❖❖

Presentation

It is often harder to find the cause of a bug than to fix it. This is due to the fact that traditional debugging tools provide a very limited view of the computations, and usually force the programmer to repeatedly run the program, narrowing down the focus area until the cause of the problem is found.

TOD is a debugger that, instead of controlling execution, records every meaningful action of the debugged program, ie. those that change its state or alter its control flow. The user interface then permits to navigate in the entire history of the program and answer questions such as “why does variable X has value Y at time T” or “why does list X contains element Y at time T”.

TOD is currently implemented in Java and is targeted at debugging Java programs, but the idea can be applied to other OO programming languages, and even other programming paradigms.

Challenges

Developing such a tool raises three main concerns: how to efficiently capture events, how to store and query them, and how to present them to the user. There is abundant literature about each of these topics taken separately, but very few about all of them as a whole. We aim at filling this gap with TOD. This document, however, only deals with database requirements.

A little bit of arithmetic will illustrate the problem we are facing: roughly speaking, a program running on a 1GHz machine could generate about 10-100 million events per second. With an average event size of 20 bytes, that means 200MB-2GB of data per second! Simply storing the data, without any processing, is far beyond the capacity of a current workstation. Hopefully we can reduce the amount of events by configuring the debugger so that some *trusted* pieces of code don't generate events. Typical trusted code is, for instance, JDK classes or parts of the program that are known to work correctly.

At this stage of the project we only have debugged toy applications that run for a very short period of

time; events are recorded in memory. Our next goal is to scale up: we will try to support 100.000 events per second (ie. 2MB of data per second), which is a rather low estimate of what a real program could generate. This amount of data requires the adoption of an adequate *database management system*.

Solution space

Let's discuss two obvious but inappropriate solutions so as to better understand the requirements of the system.

Raw storage. Given the high data rate, one solution could be to store the data packets representing the events to a file, without any processing. Any recent workstation easily handles a sustained 2MB/s disk write rate. The problem with this solution is that the data must be queried when the user starts analyzing the trace. A common query is to reconstruct the program's control flow, which is a tree structure spanning the entire event trace. Although the user is usually interested in only a small portion of it, the entire trace must be taken into account. In other words, without proper indexing an algorithm that reconstructs a portion of the control flow would have to read the whole trace.

Classical RDBMS. With the necessity to execute such complex queries, one would be tempted to use a classical "SQL database". This has actually been tested, and the outcome is that although queries might perform very well, *insertion times* are completely prohibitive. Unfortunately no formal benchmark has been realized.

Having exhausted the most obvious classical data management solutions, we have to devise another approach. We know of at least two alternative kinds of database management systems that could help us reach our goal:

Temporal databases store not only the value of a given property at time *t*, but all the history of the property. This seems a good fit for our system because a frequent query is to reason about the state of objects over time. We don't know the performance of such databases.

Streaming databases aim at processing in real time continual inputs from many sources, like sensors. They support the execution of workflow-oriented queries. The ability of these databases to handle high input rates is attractive, however to the extent of our knowledge the kind of queries they support don't really fit our needs.

Problem specification

Data

Data consists of a stream of timestamped tuples which correspond to registered events. Each type of event has different attributes. Attributes of type Object are not actually any object but either ObjectId, which identifies an object in the target VM, or String, Integer, Double, etc.

FieldWrite(long Timestamp, long ThreadId, int OperationBytecodeIndex, int FieldLocationId, Object Target, Object Value) [32]

LocalVariableWrite(long Timestamp, long ThreadId, int OperationBytecodeIndex, int VariableId, Object Value) [28]

Instantiation(long ThreadId) [8]

ConstructorChaining(long ThreadId) [8]

BeforeBehaviorCall(long Timestamp, long ThreadId, int OperationBytecodeIndex, int BehaviorLocationId, Object Target, Object[] Arguments) [28+4n]

BeforeBehaviorCall(long ThreadId, int OperationBytecodeIndex, int BehaviorLocationId) [16]

AfterBehaviorCall(long Timestamp, long ThreadId, int OperationBytecodeIndex, int BehaviorLocationId, Object Target, Object Result) [32]

AfterBehaviorCall(long ThreadId)[8]

AfterBehaviorCallWithException(long Timestamp, long ThreadId, int OperationBytecodeIndex, int BehaviorLocationId, Object Target, Object Exception) [32]

BehaviorEnter(long Timestamp, long ThreadId, int BehaviorLocationId, Object Object, Object[] Arguments) [24+4n]

BehaviorExit(long Timestamp, long ThreadId, int BehaviorLocationId, Object Result) [24]

BehaviorExitWithException(long Timestamp, long ThreadId, int BehaviorLocationId, Object Exception) [24]

ExceptionGenerated(long Timestamp, long ThreadId, int BehaviorLocationId, int OperationBytecodeIndex, Object Exception) [28]

ExceptionGenerated(long Timestamp, long ThreadId, String MethodName, String MethodSignature, String MethodDeclaringClassSignature, int OperationBytecodeIndex, Object Exception) [?]

Output (long Timestamp, long ThreadId, Output Output, byte[] Data) [?]

Queries

The set of queries that must be supported is quite restricted. We cannot present them all in this document because no formal set of queries have been defined yet. We can however outline the most prominent ones.

Control flow

The idea is to present the stream of tuples as a tree where each method entry event defines the start of a new node, and method exit event the end of the current node.

Object history

The goal of this query is to be able to retrieve the value of the fields of a given object over time. The value of field F at time T is determined by searching the latest FieldWrite event on field F whose timestamp is less than T.

Proposed work

In order to reach our goal we need a database management system that is able to handle high data input rates and that supports a finite set of non trivial queries that can potentially span the entire database. Being a research work, it is difficult to guarantee that we will be able to implement a working solution during the semester. We can however propose the following tasks:

- Provide benchmarks for the obvious but inappropriate solutions mentioned above.

- Perform an extensive bibliographic research so as to know and understand related work.
- Formalize the set of canonical queries needed by the trace analyzer and evaluate their complexity.
- Design and motivate a solution based on an existing DBMS or on a custom implementation.
- If possible implement the solution and provide benchmarks.