Velodrome

CS636: Project Report

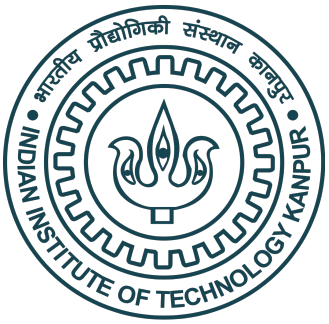
Aniket Sanghi
Roll no: 170110
sanghi@iitk.ac.in

Guntas Singh Brar
Roll no: 180274
gbrar@iitk.ac.in

Paramveer Raol
Roll no: 170459
paramvir@iitk.ac.in

Sarthak Singhal
Roll no: 170635
ssinghal@iitk.ac.in

May 1, 2021



Contents

1 Introduction

We have re-implemented Velodrome[1], a sound and precise dynamic analyzer that detects atomicity violations. It is implemented on a platform called RoadRunner which is a dynamic analysis framework that supports Java. Velodrome detects atomicity violations at the granularity of method calls and generates an exclusion list that tells us which methods aren't atomic.

The ideas and algorithms have been taken from the Velodrome paper directly but have been modified based on the needs. Like the garbage collection optimisation have been modified to operate lazily to save on memory. Similarly, the non-transactional optimisation has been modified. Each have a detailed explanation in the following sections.

2 Implementation Details

2.1 Fundamental Data-Structure's Implementation

The main constructs that are important in the implementation of Velodrome are as follows:

- $C : Tid \rightarrow Node_{\perp}$ This gives current transaction node of the thread.
- $L : Tid \rightarrow Node_{\perp}$ This gives last transaction node of the thread released the lock.
- $U : Lock \rightarrow Node_{\perp}$ This gives last transaction node that released the lock.
- $R : Var \times Tid \rightarrow Node_{\perp}$ This gives last transaction node of each thread to read Var.
- $W : Var \rightarrow Node_{\perp}$ This gives last transaction node read Var.
- $H : Node_{\perp} \times Node_{\perp}$ This graph formed by the happens-before relationship.

Note: The out-put of these constructs can also be **NULL** if there does not exists an transaction node for the condition of interest.

The analogous structures in our implementation are as follows:

1. **VDTransactionNode**: This is the transaction node class, it contains the meta-information pertaining to the transaction like the method name, number of in-edges in the graph, unique id, etc.
2. **VDLockState**: This is the lock class which is mapped with *ShadowLock* using the decoration it contains only one field lockReleaseLastTxn, which is the reference to the last transaction to release the lock. This provides the functionality analogous to $L : Tid \rightarrow Node_{\perp}$.
3. **VDThreadState** : This provides the functionality of $C : Tid \rightarrow Node_{\perp}$ and $L : Tid \rightarrow Node_{\perp}$. This contains 2 fields which are as follows:
   - **currentTxnNode** : The current transaciton going in the thread.
   - **lastTxnNode** : The last transaciton that was completed in the thread.
4. **VDVarState** : This provides the functionality of $R : Var \times Tid \rightarrow Node_{\perp}$ and $W : Var \rightarrow Node_{\perp}$ it extends the class $ShadowVar$. This contains 2 fields which are as follows:
   - **lastTxnPerThreadToRead** : This is the list of the last transaction to read for each thread for a given variable.
   - **lastTxnNode** : This gives the last transaction to write to the given variable.
5. **VDTransactionGraph** : This provides the functionality of $H : Node_{\perp} \times Node_{\perp}$ it contains numerous other graph related methos like the addEdge(), isCyclic(), GarbageCollection() etc.

2.2 Cycle Detection

We have used Depth-first search traversal to detect cycle in the transaction graph. Every time we add a new edge to the graph, invoking the **addEdge()** method, we check for cycle in the graph. If a cycle is found, we invoke the **dump()** method. The **dump()** method dumps the transaction graph into a file as a dot script with the blamed edge and blamed method highlighted.

Note: The transaction graph will always be a DAG, the edge that results in a cycle in the graph is never actually added. This is done to aid the detection of subsequent atomicity violations.

2.3 Important Details

This method gives a brief overview of the algorithms implemented. All the functions stated are in Velodrome-Tool.java unless stated other wise:

1. Whenever a method is entered it first goes into the **enter** which checks that it is not in the exclusion list and then makes the transaction-node by calling the function exnterTxn if all the conditions are satisfied .ie method not in the exclusion list, not recursive one, etc.

2. Then if there are write, read, acquire, release events in the transaction node then the updates are performed in accordance with the paper's description and then **addEdge** method is called, over-here if there is a cycle then blame-assignment is done to the destination transaction-node.

3. Then after the method gets completed it enters into the **exit** function where if appropriate (.ie method not in exclusion list etc.) the transaction node will then passed on to the **GarbageCollection** function whose details are described in the optimization section.

4. Whenever a method is entered it first goes into the **enter** which checks that it is not in the exclusion list and then makes the trasction-node a

5. The graphs generated at any point during execution will be quite small in order to get the entire graph turn off the Garbage-Collection of the exitTxn() function in VelodromTool.java file.

6. To get the graph just use the dump("src","dest") method this will make a dot file name "src","dest".

7. Every method is assumed to be atomic initially but as we execute Velodrome, the exclusion list is gradually built using the method of iterative refinement. The final exclusion list contains functions, excluding which, the code is atomic.

8. To be more efficient with garbage collecting, the graph is always kept as a **DAG**. This is done by not adding the edge because of which a cycle will be created. The information that a cycle was seen is communicated using dot files for micro-benchmarks and exclusion lists for macro-benchmarks.

3 Optimizations

3.1 Garbage Collection

3.1.1 Normal Garbage Collection

The first implementation in GC was done in the following manner:

1. Add a following fields in **VDTransactionNode** : lockList, readList, writeList. These lists are the locks, vars that read or wrote in the current transaction.

2. On each access' write or read, lock's acquire the corresponding **VDVarState** and **VDLockState** respectively will be added to the appropriate list of VDTransactioNode.

3. Finally when the transaction gets completed then the graph region with root equal to the exited transaction is explored. The traversal is in bfs-traversal, and the nodes that satisfy the condition (1) the number of inEdges==0 (2) the transaction is finished are deleted.

4. The lists (ie lock, read etc) of the nodes to be deleted are set to null, provided they still point to the same transaction node.

5. Finally when all the nodes that could be deleted are

Note that bfs is required to ensure that graph is efficiently garbage collected because if the exited-node is the only one that is deleted then there may be cases where the exited-node's count is not 0 and hence the graph beyond this will never be deleted. If bfs not applied. In the method described above does the removal of the graph from the node, plus the updation of read, writes, locks, last-transaction all just for one node's deletion. Because of the fact that graph operations have to be synchronized a great chunk of time will be spent in this method, and the requirement of bfs further exacerbated the situation. This is resolved in the following manner

Note: Finally Lazy GC is currently implented but with minor changes(can be seen in one of the commit) normal GC can be done too.

3.1.2 Lazy Garbage Collection

The implementation details and change for Lazy GC are as follow:

1. Add a field **isDeleted** in **VDTransactionNode**. This field is set to true as soon as the transaction node is removed from the graph(ie. adjacency-list)/
2. When the transaction gets completed then the graph region with root equal to the exited transaction is explored. The traversal is in bfs-traversal, and the nodes that satisfy the conditions (1) the number of inEdges==0 (2) the transaction is finished and (3) isDeteted flag is false are deleted.
3. Now whenever last-transaction of thread, lock, variable (read or write) is read , then the last transaction's isDeleted checked. If it is true then the reference is set to null. Hence this is a lazy implementation, the advantage over here is that the node deletion is quite fast and the adjacency list remains small because of the faster method.
4. In case of a write to the last-transaction of thread, lock, variable (read or write) reference is any-way changed so does not require any change.

The current method although does garbage-collection to a lesser degree, as even if one meta-data points to the transaction node then the node wouldn't be garbage collected. However, the graph traversal becomes quite fast thus the number of nodes in the graph remains quite small and all this is done without the list traversals of the previous method. In programs where the variables are uniformly accessed intuitively, the given method should perform GC to the same degree as the previous method.

3.2 Non-Transactional Operations

For the operations that are outside transactions, the nodes are allocated at an extremely fast rate which leads to a long sequence of unary transactions. To avoid this problem optimized rules are used for non-transactional operations: acquire, release, write and read.

Algorithm:

1. If current operation is a non-transactional operation then first find a list of nodes (let's denote it by L) from which an edge would be added to current operation if current node would have been inside a transaction. E.g. last node to write current variable, last node to unlock current lock, etc.
2. Now based on the above information decide whether it is necessary to create a node for current operation or not.
3. If there is a node in L which follows a happens-after relation from all the other nodes in L then don't create a new node for the current operation. Instead merge the happens-after node with the current node.
   (a) To get the happens-after node first the graph is reversed i.e. the direction of edges is reversed.
   (b) Then for each of the nodes in L it is checked whether we can reach to rest of the nodes in L using depth-first search.
   (c) For the nodes which were visited in the dfs, we don't check whether it is happens-before all the nodes because it was happens-after node for the earlier node for which dfs was called.
4. If no such node exists then create a fresh node and add an edge from each of the node in L to the freshly made node.

4 Execution Instructions

4.1 General Instructions

How to test atomicity violation using velodrome:

```
rrrun -tool=VD -isMicroBenchMark=<0/1> -exclusionList=<inputfile> \
-outputExclusionList=<outputfile> -field=FINE -array=FINE -noTidGC \
-availableProcessors=4 <classpath>
```

- VD is the abbreviation of the velodrome tool
- There are 3 additional command line parameters:
  - *isMicroBenchMark*: equals to 1 if testing on microbenchmark else 0. The difference between the two is that for benchmarks dot files are not generated.
  - *exclusionList*: file which contains the current exclusion list.
  - *outputExclusionList*: file in which the new exclusion list is to be dumped
- The final exclusion list is computed iteratively. Initially, the input exclusion list will be empty. The output exclusion list from the first run has to be manually copied to the input exclusion list. So, in the next run new non-atomic methods will get detected and again those have to be manually copied to the input exclusion list. This process will continue till there are no more atomicity violations.
- The directory structure for the benchmarks and microbenchmarks is as follows:
  - There is a *testcases* directory in the project root directory which contains directories *MicroBenchmarks* and *Benchmarks*.
  - The *MicroBenchmarks* directory contains 2 directories *atomicity* (provided by Prof. Swarnendu Biswas) and *self* (created by us).
- *classpath*: It is the relative path of the class you want to test w.r.t execution directory. E.g. If you want to test ThreadDemo2.java provided in the microbenchmarks from the project root directory then the classpath would be *testcases.MicroBenchmarks.atomicity.ThreadDemo2*

4.2 Microbenchmarks

There is a script *test.sh* in the root directory which tests on a given class and generates exclusion list and transaction graphs.

How to execute:

```
./test.sh <classpath> <inputExclusionList> <outputExclusionList>
```

- *classpath*: relative path of the class you want to test w.r.t execution directory. E.g. If you want to test ThreadDemo2.java provided in the microbenchmarks from the project root directory then the classpath would be *testcases/MicroBenchmarks/atomicity/ThreadDemo2*.
- *inputExclusionList*: relative path of the input exclusion list w.r.t execution directory.
- *outputExclusionList*: relative path of the output exclusion list w.r.t execution directory.
- This script generates the .dot files and their corresponding .ps files in the *output/⟨classpath⟩* directory inside the project root directory.

4.3 Benchmarks

The Velodrome's 5 benchmarks can be found in https://github.com/sarthak2007/Velodrome/tree/main/testcases/Benchmarks. To run any of the benchmark the direct procedure is -

How to execute:

```
cd <benchmark>
ant
cd classes
<execute the command given in <benchmark>/README with the required flags>
```

The flags you will need to provide are

- **-exclusionList**: The path of the input exclusion list file
- **-outputExclusionList**: The path of the file where you need the exclusion list generated to be stored

To build up the exclusion list, keep doing the below steps until the output exclusion list comes empty

```
<execute the command given in <benchmark>/README with the required flags>
cat <path to output exclusion list file> >> <path to input exclusion list file>
```

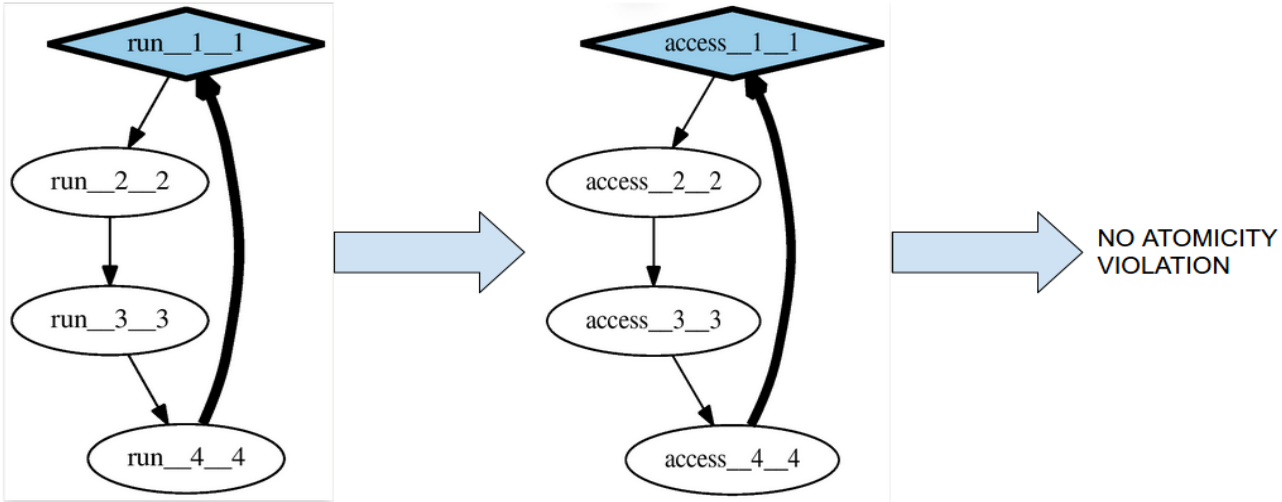
A similar procedure will be used to run the DeCapo benchmark, only the command to execute the benchmark will be modified.

5 Testing

5.1 Micro-Benchmarks

The way we get the exclusion list for the micro bechmarks is as follows:

1. Lets look at the case of ThreadDemo6. It is first ran with no exclusion list at the cyclic graph of it is as shown in the first part.
2. Then the run method is added to the exclusion list and the graph generated is the second one in the picture.
3. Finally after adding access method to the exclusion there are no more atomicity violation due to the fact that now there are only unary accesses.



NOTE: In the above graphs the bold edge signifies that it was the reason of the cycle creation and the blue colored node represents the method that is blamed for non-atomicity. The nomenclature of nodes represented is as follows *method-name_tid_labelid*, where *labelid* is unique to the node.

- **CaughtArithmeticExpression.java**
  Single threaded hence no atomicity violation and none detected.

- **Change.java**
  A cycle between **classA.run()** and **classB.run()** is detected which is due to an atomicity violation because of line no 17-18 and 27-28 .

- **Consumer.java**
  Atomicity violation is detected due to **Consumer.run()** and **Producer.run()** which is evident as both modify **Producer.messages** which results in a cycle.

- **Synchronize.java**
  There are no shared variabes hence there are no atomicity violations and none are detected.

- **test2.java**
  Single threaded hence no atomicity violation and none detected.get

- **TestArrayReadsWrites.java**
  Single threaded hence no atomicity violation and none detected.

- **TestFieldReadsWrites.java**
  Single threaded hence no atomicity violation and none detected.

- **TestVariableHiding.java**
  Single threaded hence no atomicity violation and none detected.

- **Update.java**
  Atomicity violation is detected due **Update.run()** as both the instances of Update modify **Update.acc** which results in a cycle.

- **ThreadDemo.java**
  Multiple methods of MyObject3 are called in the 2 thread's run function thus creating the dependency amongst the run methods of the threads however it run method is added to the exclusion list no more atomicity violation will be there as the methods of the MyObject3 are synchronized.

- **ThreadDemo2.java**
  The sleeps are such that any parent of access method (ie. main and run) will always violate atomicity, so these method must be in the exclusion list thus no method is in atomic.

- **ThreadDemo3.java**
  Get and Set methods are not properly synchronized .ie the synchronize(this) is does not entirely cover get method and hence there is no method that is atomic.

- **ThreadDemo4.java**
  The sleeps are set such that the atomicity violations between the access methods are always exposed hence no method is atomic.

- **ThreadDemo5.java**
  The sleeps are set such that the atomicity violations between the access methods are always exposed hence no method is atomic.

- **ThreadDemo6.java**
  Same as above only more threads 4 node cycle.

- **ThreadDemo7.java**
  Same as above only more threads 3 node cycle.

5.2 Benchmarks

5.2.1 Performance

Benchmarks	Base time on tool=N	Execution Time of Velodrome
elevator	23.8s	23.9s
philo	2.8s	3.1
sor	0.26s	0.32s
luindex	7.4s	174.9s

Table 1: Velodrome's performance on benchmarks

5.2.2 Exclusion lists obtained

All of the exclusion lists can be found at https://github.com/sarthak2007/Velodrome/tree/main/exclusionLists

- **elevator**
  - elevator/Elevator.main([Ljava/lang/String;)V
  - elevator/Lift.run()V
  - elevator/Lift.doIdle()V
  - elevator/Elevator.begin()V
  - elevator/Controls.claimUp(Ljava/lang/String;I)Z
- **philo**
  - Philo.run()V
  - Table.getForks(I)I
- **sor**
  - sor/sor_first_row_odd.run()V
  - sor/Sor.main([Ljava/lang/String;)V
  - EDU/oswego/cs/dl/util/concurrent/CyclicBarrier.barrier()I
  - EDU/oswego/cs/dl/util/concurrent/CyclicBarrier.doBarrier(ZJ)I
- **luindex**
  - RRBench.iterate()V
  - Index.main(Ljava/io/File;[Ljava/lang/String;)V
  - org/apache/lucene/index/IndexWriter.optimize()V
  - org/apache/lucene/index/IndexWriter.optimize(Z)V
  - org/apache/lucene/index/IndexWriter.optimize(IZ)V

5.2.3 Other Benchmarks

Our Velodrome tool's memory usage overpowered the systems we were testing on for benchmarks like **avrora**, **hedc**. They enter a phase where GC is tries to clean-up but keeps failing. For such cases we tried to find the exclusion list by force stopping the program manually. Ctrl + C triggers the printXML function of our tool which outputs the exclusion list created till then. By following this way we found the exclusion lists for avrora and hedc to some extent.

- avrora
  - avrora/sim/SimulatorThread.run()V
  - avrora/sim/Simulator.start()V
  - avrora/sim/AtmelInterpreter.start()V

  - avrora/arch/legacy/LegacyInterpreter.runLoop()V
  - avrora/arch/legacy/LegacyInterpreter.fastLoop()V
  - avrora/sim/AtmelInterpreter.advanceClock(J)V
  - avrora/sim/clock/MainClock.advance(J)V
  - avrora/sim/clock/DeltaQueue.advance(J)V
  - avrora/sim/clock/DeltaQueue.advanceSlow(J)V
- hedc
  - hedc/PooledExecutorWithInvalidate$Worker.run()V
  - hedc/Tester.run()V
  - hedc/MetaSearchRequest.go()V
  - hedc/MetaSearchImpl.search(Ljava/util/Hashtable;Lhedc/MetaSearchRequest;)Ljava/util/List;
  - hedc/PooledExecutorWithInvalidate.getTask()Lhedc/Task;
  - EDU/oswego/cs/dl/util/concurrent/LinkedQueue.take()Ljava/lang/Object;

References

[1] Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. https://dl.acm.org/doi/10.1145/1379022.1375618

[2] Prof. Swarnendu Biswas' Github repository with exclusion lists for benchmarks. https://github.com/swarnendubiswas/velodrome