AeroDrome in RoadRunner

CS636 Analysis of Concurrent Programs

Project Members:

Bhuvan Singla bhuvans@iitk.ac.in

Krishanu Singh krishanu@iitk.ac.in

Shobhit Jagga shobhitj@iitk.ac.in

Course Instructor:

Swarnendu Biswas swarnendu@cse.iitk.ac.in

Teaching Assistant:

Arun KP kparun@cse.iitk.ac.in

1 Introduction

Concurrent programs are challenging to write, debug and reason about because of a large number of possible interleavings. The number of interleavings increases exponentially with the number of instructions and threads and reasoning about correctness and performance for all of them becomes infeasible for the programmer. Since programmers tend to think sequentially while writing their code, they assume (intentionally or otherwise) certain sections to be atomic and such behaviour, more often than not, is not captured during the runtime. Now if we possess some knowledge about the atomic execution of these code sections, the number of possible interleavings drops drastically since the entire atomic section executes as if there was no interference from other threads during its execution. It makes understanding and debugging the program much easier because of the reduced interleaving space. A detected interference would indicate that programmers' expectations of atomic execution were violated and can be separately looked into. Also, atomicity violations do not have a correspondence with data races. A program may be data race free but still exhibit violations of atomicity and unexpected behaviour.

2 AeroDrome

AeroDrome is a dynamic one-pass linear time dynamic atomicity violation detector based on an alternative characterization of conflict serializability. Using this alternative characterization, conflicting events are detected via vector clock comparisons. Implementation using vector clocks is not trivial and directly intuitive as in work like DJIT$^+$ and FastTrack since happens-before relation now is over transactions instead of individual operations. Also, we can know about ordering of any event w.r.t. a transaction only after the transaction ends. The proposed algorithm assigns vector clock timestamps to individual events (constantly many events at a time) and keeps summarized information in them so as to detect atomicity violations without tracking clocks for each event. The algorithm, being linear time and single pass, is a significant improvement over previous graph based approaches like Velodrome[1] (which takes more time/memory) and DoubleChecker[2] (which is two-phased).

3 Our Implementation

Our implementation can broadly be categorized into the following 3 disjoint sections based on their functionality:

- **Atomicity Specification** - The input specification regarding which code sections need to be regarded as atomic (or as transaction equivalently) can be passed to the code in the following ways:
  - **Method Exclusion** - The source code methods which are to be excluded from consideration as atomic blocks are passed in a file. The file name can be passed via the command line parameter `-transactionFile` and is considered `transactionfile.csv` by default. In case of nested methods, all the methods are subsumed in a single transaction if none of those methods are excluded. This is implemented by maintaining the stack height of method calls in our implementation. Exclusion of outer method does not exclude the inner methods. Similarly, exclusion of inner method does not exclude the content from the bigger transaction of outer method. The `readMethodExcludeFile` method reads the file and fills the excluded methods in a hash map. The method enter and exit handlers provided by Roadrunner [3] are used as entry/exit point for transactions if the method was not excluded. This exclusion check occurs inside the `checkMethod` method call.
  - **Source Location pair** - The source locations between which the code is expected to perform atomically can also be passed as pairs in a file. The file name can be passed via the command line parameter `-transactionFile` and is considered `transactionfile.csv` by default. The command line parameter `-fileType` has to be passed to indicate that the file contains source location pairs instead of excluded methods. This input way presents an opportunity for a finer specification and tracking of atomic regions in the code than the method-based granularity. The method `readLocationPairFile` reads location pairs from file and fills them into a hash map. `CheckSourceLocation` method calls transactionBegin and transactionEnd handlers appropriately on encountering those source locations.
- **Aerodrome Logic** - We implement Algorithm 2 from the AeroDrome paper [4] inside our tool class `AeroDromeTool.java`. Algorithm 2 applies read optimization over the main algorithm and is expected to perform better than the main algorithm presented in the paper. The tool maintains and outputs the log of atomicity violations at the end of program execution. We maintain additional violation metadata in terms of variables which result in violations and their corresponding transactions. We also added counters

for events and logged them as well for validation with the data presented in AeroDrome [4].

- **Vector Clock** - The vector clock implementation basically consists of some basic functions which check if a vector clock is less than or equal to another, copies vector clocks and contains other helper functions. We have 2 working versions of vector clocks. One is implemented by us using the `VectorClock` class in RoadRunner by additionally implementing and using the relevant helper functions. The other is used from the vector clock implementation in `RAPID`. Both these versions have been tested for correctness on the microbenchmarks. Their performance evaluation is done in Section 5 below.

  AeroDrome implementation with VC1 can be found in the branch `main-vc-1` with vector clock at `/src/tools/util/VectorClock.java`

  AeroDrome implementation with VC2 can be found in the branch `main-vc-2` with vector clock at `src/tools/aerodrome/ADVectorClock.java`

4 Microbenchmarks

We validate our code for correctness by various microbenchmarks provided on Piazza and our own custom cases. These microbenchmarks can be found in the `/microbenchmarks` directory in our repository.

While executing `rrrun` in our tests we have passed the following options:

- `-noFP` - Do not use in-lined tool fastpath code for reads/writes.
- `-noTidGC` - Do not reuse the tid for a thread that has completed.

4.1 Custom Microbenchmarks

Our own custom benchmarks can be found in the `microbenchmarks/custom` with names of the form YTest1.java and NTest1.java. The benchmarks with filename starting with Y have one or more races and the ones that with start with N don't have a race. Each file contains a header describing the trace and the races which the file attempts to recreate. Note that since it may happen that the trace was not recreated successfully, each file prints out the event stream so that it could be verified manually that if the recreation was successful. If it so happens that few events are jumbled up then the sleep values could be tinkered with to fix it or running it a few times may help too.
Compile the microbenchmark with:

```
cd microbenchmarks/custom/
javac test/YTest1.java
```

Run the microbenchmark with:

```
rrrun -noFP -noTidGC -tool=AD test.YTest1
```

To run the microbenchmarks with a source location pair file with -fileType option. For instance, we can run Test.java microbenchmark with source location pair file with

```
cd microbenchmarks/custom/
javac test/Test.java
rrrun -fileType -noFP -noTidGC -tool=AD test.Test
```

We have also included a sample source location pair file for testing the microbenchmark Test.java at `microbenchmarks/custom/`

4.2 Piazza Microbenchmarks

The microbenchmarks from Piazza can be found at `microbenchmarks/atomicity` and the logs generated by RoadRunner on testing these microbenchmarks can be found at `microbenchmarks/atomicity-logs`.

Compile the microbenchmark with

```
cd microbenchmarks/
javac atomicity/ThreadDemo.java
```

Run the microbenchmark with

```
rrrun -noFP -noTidGC -tool=AD atomicity.ThreadDemo
```

5 Benchmarks

5.1 Iterative Refinement

We apply the method of **iterative refinement** as explained in [2] to come up with a list of methods which are non-atomic in an application.
Steps for iterative refinement:

1. cd into a specific benchmark directory. `touch` an empty "transactionfile.csv" file
2. Run the test script using the commands given in the next section and store the output in some file.
3. Go to the printed xml data section and find the $< tool >$ block. It will contain a list of $< violation >$ tags with the type,location and method info . Copy all the unique transaction method names and append them to the transactionfile.csv file.
4. Run the script till there are no violations left. Once there are no violations left, run the script a couple of more times (say, 10 times).

5.2 Benchmark Execution

In the execution of the benchmarks below, VC1 refers to our final vector clock implementation in RoadRunner, and VC2 refers to the vector clock implementation picked from `RAPID`. The times mentioned in tables are normalized wrt time obtained from the `empty` tool. All the log files per iteration for both vector clock implementations are present in the repository.

5.2.1 elevator

Execution command (edit the TEST script to change the name of the output file):

```
cd benchmarks/elevator/
./TEST
```

	Iter1	Iter2	Iter3	Iter4	Iter5
VC1	1.018	1.001	1.073	1.074	1.075
VC2	1.113	1.093	1.212	-	-

As can be seen in the Table above, VC1 runs better than VC2 in the first 3 iterations. We did not find any new violations after three iterations of VC2, so its 4th and 5th columns are blank. The methods removed successively after iterations are shown below:

```
Iter1: elevator/Lift.run()V, elevator/Elevator.main([Ljava/lang/String;)V
Iter2: elevator/Lift.doIdle()V, elevator/Elevator.begin()V
Iter3: elevator/Controls.claimDown(Ljava/lang/String;I)Z
```

5.2.2 tsp

edit the TEST script to change the name of the output file

```
cd benchmarks/tsp/
./TEST
```

	Iter1	Iter2	Iter3	Iter4	Iter5	Iter6
VC1	3.379	3.311	2.780	2.928	3.242	4.279
VC2	137.529	139.842	162.527	143.496	145.598	149.352

We see that our vector clock implementation, based on the vector clock built and optimized for RoadRunner, runs significantly better than the one implemented in RAPID. We do not suspect it to be related to algorithmic fault, but some internal incompatibility with RoadRunner since RoadRunner kept on running the weak resource cleaner most of the times in the second vector clock.

The methods removed successively after iterations are shown below:

```
Iter1: TspSolver.run()V, Tsp.main([Ljava/lang/String;)V
Iter2: TspSolver.Worker()V
Iter3: TspSolver.get_tour(I)I, TspSolver.recursive_solve(I)V
Iter4: TspSolver.visit_nodes(I)V, TspSolver.find_solvable_tour()I
```

5.2.3 sor

```
cd benchmarks/sor/
./TEST -noFP -noTidGC -tool=AD
```

Similar to tsp benchmarks, sor took significantly greater amount of time for VC2 as compared to VC1. We terminated this benchmark after a timeout of 10 minutes and have only collected logs for VC1. The obtained times for VC1 can be found in the git repository.

The methods removed successively after iterations are shown below:

```
Iter1: sor/SORRunner.run()V, JGFSORBenchSizeA.main([Ljava/lang/String;)V
Iter2: jgfutil/TournamentBarrier.DoBarrier(I)V, sor/JGFSORBench.JGFrun(I)V
Iter3: jgfutil/TournamentBarrier.set(IZ)V, jgfutil/TournamentBarrier.get(I)Z,
          jgfutil/TournamentBarrier.debug(Ljava/lang/String;)V,
          sor/JGFSORBench.JGFkernel()V
Iter4: sor/SOR.SORrun(D[[DI)V
Iter5: sor/JGFSORBench.JGFvalidate()V
Iter6: jgfutil/JGFInstrumentor.stopTimer(Ljava/lang/String;)V
```

5.2.4 philo

```
cd benchmarks/philo/
./TEST
```

Comparison of VC times: We can see that both VC1 performs slightly better than VC2 for

	Iter1	Iter2	Iter3
VC2	0.909	0.888	0.910
VC1	1.04	1.088	1.071

this benchmark.

The methods removed successively after iterations are shown below:

```
Iter1: Philo.run()V, Philo.main([Ljava/lang/String;)V
Iter2: Table.getForks(I)I
```

5.2.5 Avrora

```
cd benchmarks/philo/
./TEST -noFP -noTidGC -tool=AD
```

```
Iter1: avrora/sim/SimulatorThread.run()V, Main.main([Ljava/lang/String;)V
Iter2: avrora/sim/clock/RippleSynchronizer.removeNode
        (Lavrora/sim/Simulation$Node;)V,
        avrora/sim/Simulator.start()V
```

```
Iter3: avrora/Main.runAction()V, avrora/sim/AtmelInterpreter.start()V,
        avrora/sim/clock/RippleSynchronizer.waitForNeighbors(J)V
Iter4: avrora/arch/legacy/LegacyInterpreter.runLoop()V,
    avrora/actions/SimAction.run([Ljava/lang/String;)V,
    avrora/sim/clock/RippleSynchronizer.waitFor
    (JLavrora/sim/clock/RippleSynchronizer$WaitLink;)V
```

After 2-3 iterations, avrora benchmark took a lot of time in running mostly in weak-resource cleaning. We have attached the logs and non-atomic sections found in the executed runs.

5.2.6 HEDC

The hedc benchmark has execution issues with RoadRunner. The benchmark does not run for other tools like empty tool or FT2 tool. We tried resolving the issue by providing a value of 4 in the command line parameter - infThreads (since 4 threads seemed to be running infinitely) but it did not work and we did not run this benchmark after resolution efforts.

6 References

1. Flanagan, C., Freund, S. N. & Yi, J. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. *SIGPLAN Not.* **43,** 293–303 (June 2008).
2. Biswas, S., Huang, J., Sengupta, A. & Bond, M. D. *DoubleChecker: Efficient Sound and Precise Atomicity Checking* in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, Edinburgh, United Kingdom, 2014), 28–39.
3. Flanagan, C. & Freund, S. N. *The RoadRunner Dynamic Analysis Framework for Concurrent Programs* in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Association for Computing Machinery, Toronto, Ontario, Canada, 2010), 1–8.
4. Mathur, U. & Viswanathan, M. Atomicity Checking in Linear Time using Vector Clocks. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2020).