A THG Performance Case Study in the world of E-Commerce

Philip Wilson, Rehman Arshad, James Creedy, Adam Dad, Eloise Slater, Hannah Cusworth

philip.wilson, rehman.arshad, adam.dad, eloise.slater, hannah.cusworth @thehutgroup.com

The Hut Group
Chicago Ave, Voyager House, Manchester, UK

Abstract

THG's in-house e-commerce platform includes a micro-service aggregator and server-side template renderer (that we call THG Aggregator for this paper) that handles hundreds of THG websites. It is not uncommon in e-commerce to entertain thousands of requests per minute and THG Aggregator uses extensive in-memory (on JVM) caching to support the performance requirements. This case study revolves around analysing the performance of this in-house system on current JDK version (JDK 8) it is running on and the latest LTS (long term support) version (JDK 11) to decide the LTS version configurations before migrating THG Aggregator to that. We analysed parameters like heap utilisation, GC (garbage collector) pause time and process usage under different configurations of JDK 8 and 11. Based on this analysis, we showed different options and configurations of JVM that can enhance or decrease the performance of THG Aggregator. We have also conducted extensive analysis of all these variations on Arm VS Intel to extract the best combination of instruction set architecture and JDK variation in terms of the response time of user requests.

Key Words —E-Commerce, JDK, JVM [1]

I. Introduction

THG Aggregator is the heart of THG's e-commerce operation (Fig. 1). THG Aggregator handles hundreds of THG websites and act as a major caching and aggregating entity between the outside world and the internal resources. This case study revolves around analysing the performance of THG Aggregator on current JDK version (JDK 8) it is running on and the latest LTS (long term support) version (JDK 11) to decide the LTS version configurations before migrating THG Aggregator to that. We have defined a framework of analysis that can show variations in the values of parameters like heap usage, total duration of GC (garbage collector) pauses, app usage etc. based on different configurations of JVM. Configuration parameters include parallel GC threads (*-XX:ParallelGCThreads*), type of garbage collector, maximum heap size and *Stringdeduplication* etc.

In order to make sure that the patterns of the findings hold for different number of requests, we have conducted analysis based on 1, 10, and 100 and *n* number of requests whereas, *n* is the number of requests set via load testing. Table I shows different configurations used in the analysis, ranges from v0-v8. The configuration v4 is currently being used in THG with Java 8. We have compared v0-v4 via our framework of analysis between Java 8 and 11 to show the performance enhancements by hitting the running configurations of THG Aggregator. From v5-v8, the configurations were only run against Java 11 to analyse if these configurations can provide better performance as compared to v4. Few parameters can also show some variation based on the current usage and specifications of a machine therefore, we have conducted each type of request on multiple machines and use the average values in our framework of analysis. All the virtual and physical machines used in this analysis had the same specifications to rule out any inconsistency in data.

The instances of THG Aggregator used in this research are not set-up with load balancers and are based on a single proxy server therefore, actual values of defined parameters for system in production are way more optimised however, the test bed created for this analysis is enough to compare and contrast the different configurations and load testing for the required analysis.

The remainder of this paper is organised as follows:

[1]DOI reference number: 10.18293/SEKE2022-067

GC Configurations	
v0	Default Configurations of a JDK Version
v1	-Xmx10240M -Xms10240M
v2	-Xmx10240M -Xms10240M -XX:+UseG1GC
v3	-Xmx10240M -Xms10240M -XX:+UseG1GC -XX:+UseStringDeduplication
v4	-Xmx10240M -Xms10240M -XX:+UseG1GC -XX:+UseStringDeduplication -XX:InitiatingHeapOccupancyPercent=60
v5	-Xmx10240M -Xms10240M -XX:+UseG1GC -XX:+UseStringDeduplication -XX:InitiatingHeapOccupancyPercent=70
v6	v4 + -XX:ParallelGCThreads=16 -XX:ConcGCThreads=4
v7	v4 + -XX:G1MixedGCLiveThresholdPercent=75
v8	v5 + -XX:ParallelGCThreads=16 -XX:ConcGCThreads=4 -XX:G1MixedGCLiveThresholdPercent=75

TABLE I: GC Configurations for THG Aggregator

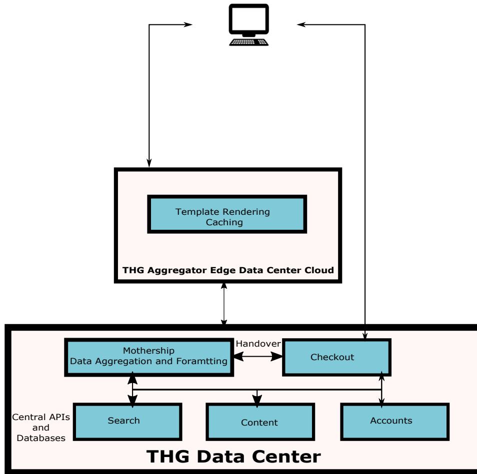

Fig. 1: THG Aggregator

Section II defines the framework of analysis used for comparing the performance of THG Aggregator. Section III discusses the findings and the analysis. This section shows multiple tables and graphs to compare and contrast the running configurations on Java 8 and 11. Section IV includes the related work which is confined to similar studies and case reports. Section V is the last section and it includes conclusion and future work.

This paper can be used as a reference in the world of e-commerce to conduct performance based analysis of different versions of JVMs under different configurations. Some pre-planned analysis like this can ensure the benefits of migration under different set of configurations to latest LTS version of Java.

II. Framework of Analysis

Based on the configurations defined in Table. I, we have divided our analysis framework into two main parts.

- First part is about analysing the important JVM parameters. Table. II is showing parameters of part 1 of analysis framework, and they are defined as follows:
  - ***GC Young Config.*** refers to the garbage collector used for young generation e.g., in JDK 8, default one is parallel scavenger whereas, in JDK 11, default garbage collector is G1 New.
  - ***GC Old Config.*** refers to the garbage collector used for old space.
  - ***GC Time Ratio.*** refers to the ratio between the time spent in GC and the time spent outside of GC.
  - ***Max. Allocated Heap.*** refers to the amount of heap set as the maximum value (-Xmx).
  - ***Max. Young generation size.*** refers to the size allocated to young generation.
  - ***Max. heap used.*** refers to the maximum heap consumed by a specific configuration on specific JDK version.
  - ***Max. heap value post GC.*** refers to the heap memory a specific configuration holds after a major GC.
  - ***Total duration of GC pauses.*** refers to the sum of all the GC pauses that take place during n number of requests under specific configurations and JDK version.
  - ***Longest Pause.*** refers to the longest pause value out of all the pauses GC takes under specific configurations and JDK version.
  - ***CPU Usage: Machine.*** refers to total utilisation of the machine's CPU while running an instance of THG Aggregator under specific configurations and JDK version.
  - ***CPU Usage: JVM + App.*** refers to the percentage of CPU utilisation takes by THG Aggregator and JVM out of the total percentage of machine usage e.g., if machine usage is 89.2% and *JVM+App* usage is 24.38% then THG Aggregator is taking 27.33% of the usage under that configuration.
- Second part (Part 2) is related to load testing i.e., the number of requests a specific configuration of THG Aggregator can handle under defined time. We ran n number of requests directed to running instances of THG Aggregator for m number of users and compare

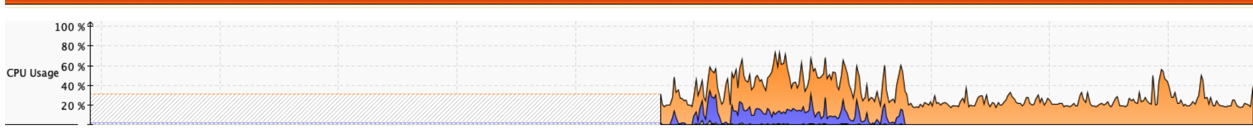

Fig. 2: Machine and *JVM+App* Usage: THG Aggregator 100 requests using JDK 11

the results handled by different configurations of JVM to analyse the impact of these configurations on performance of THG Aggregator. The performance will be elaborated via tables and graphs in section III-B.

100 Requests (no-cache) v1 Avg.		
	JDK 8	**JDK 11**
GC Young Config.	Parallel Scavenger	G1 New
GC Old Config.	Parallel Old	G1 Old
GC Time Ratio	99%	12%
Max. allocated heap	10 GB	10 GB
Max. Young generation size	3.33 GB	1.3mb
Max. heap used	3.18 GB	1.56 GB
Max. Heap value post GC	478mb	451.5mb
Total duration of GC Pauses	2828.18ms	885.73ms
Longest Pause	1298ms	281.49ms
Max CPU Usage: Machine	89.2%	84.4%
Max CPU Usage: JVM+App	24.38%	24.3%

TABLE II: THG Aggregator on JDK 8 VS JDK 11: Avg. of 100 Requests based on 2 different machines using v1

100 Requests (no-cache) v2 Avg.		
	JDK 8	**JDK 11**
GC Young Config.	G1 New	G1 New
GC Old Config.	G1 Old	G1 Old
GC Time Ratio	9%	12%
Max. allocated heap	10 GB	10 GB
Max. Young generation size	1.3mb	1.3mb
Max. heap used	2.08 GB	1.71 GB
Max. Heap value post GC	572mb	557.5mb
Total duration of GC Pauses	1210.19ms	1076ms
Longest Pause	251.11ms	332.59ms
Max CPU Usage: Machine	90.2%	73.7%
Max CPU Usage: JVM+App	38.5%	36.45%

TABLE III: THG Aggregator on JDK 8 VS JDK 11: Avg. of 100 Requests based on 2 different machines using v2

Overall, first part of the analysis was conducted with variable number of requests directed to running instances of THG Aggregator with defined configurations of specific JDK version. The final values of the defined parameters are the averages taken from running the same configurations on multiple machines under same load. Same set of requests was used for all the combinations in order to be consistent across different JDKs and configurations. The second part of the analysis was conducted against various sets of n requests with m users spawn up every second to

100 Requests (no-cache) v3 Avg.		
	JDK 8	**JDK 11**
GC Young Config.	G1 New	G1 New
GC Old Config.	G1 Old	G1 Old
GC Time Ratio	9%	12%
Max. allocated heap	10 GB	10 GB
Max. Young generation size	1.3mb	1.3mb
Max. heap used	1.91 GB	1.97 GB
Max. Heap value post GC	602.5mb	513.5mb
Total duration of GC Pauses	1501ms	759ms
Longest Pause	380.28ms	212.37ms
Max CPU Usage: Machine	89.3%	64.7%
Max CPU Usage: JVM+App	14.9%	18.5%

TABLE IV: THG Aggregator on JDK 8 VS JDK 11: Avg. of 100 Requests based on 2 different machines using v3

100 Requests (no-cache) v4 Avg.		
	JDK 8	**JDK 11**
GC Young Config.	G1 New	G1 New
GC Old Config.	G1 Old	G1 Old
GC Time Ratio	9%	12%
Max. allocated heap	10 GB	10 GB
Max. Young generation size	1.3mbGB	1.3mb
Max. heap used	3.31 GB	1.62 GB
Max. Heap value post GC	611.5mb	522mb
Total duration of GC Pauses	1243.68ms	706.70ms
Longest Pause	459.2ms	190.5ms
Max CPU Usage: Machine	76.5%	69.75%
Max CPU Usage: JVM+App	40.7%	25.45%

TABLE V: THG Aggregator on JDK 8 VS JDK 11: Avg. of 100 Requests based on 2 different machines using v4

see how JVM based optimisations can impact the end user experience by offering better performance.

III. Findings and Discussion

A. Analysis of JVM Parameters

For part 1 of the analysis and configuration v1, Table. II is showing 100 requests directed to running instances of THG Aggregator on JDK 8 and 11 respectively. In the stated table, default configurations of these JDK versions were used i..e., JDK 8 runs on *parallel scavenger* and *parallel old* GC configurations VS G1 New and G1 Old GC configurations of JDK 11. For both JDKs, maximum allocated heap was 10 GB. JDK 11 used way less heap than JDK 8 which is due to better heap management in JDK 11. Overall, JDK 11 performed considerably better

THG Aggregator 100 Requests (no-cache) JDK 11 Based on Avg. of Two Machines					
	V4	**V5**	**V6**	**V7**	**V8**
GC Young Config.	G1 New	G1 New	G1 New	G1 New	G1 New
GC Old Config.	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old
GC Time Ratio	12%	12%	12%	12%	12%
Max. allocated heap	10 GB	10 GB	10 GB	10 GB	10 GB
Max. Young generation size	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb
Max. heap used	1.62 GB	1.32 GB	1.54 GB	1.48 GB	3.41 GB
Max. Heap value post GC	522mb	514.5mb	356.5mb	552mb	530.5mb
Total duration of GC Pauses	706.70ms	1271.56ms	512.38ms	1374ms	2690.27ms
Longest Pause	190.5ms	356.24ms	242ms	398.5ms	1512.2ms
Max CPU Usage: Machine	69.75%	71.1%	68.2%	90%	78%
Max CPU Usage: JVM+App	25.45%	48.15%	51.45%	50.95%	38.35%

TABLE VI: THG Aggregator on JDK 11: 100 Requests based on different GC Configurations

in heap management, pause times and usage VS JDK 8.

For the configuration v2 (both JDK 8 and 11 were using G1 in v2) Table. III showed that JDK 11 was still better in GC total pause duration however, *JVM+App* utilisation was 42.68% (i.e., 38.5 is 42.68% of 90.2 which is total CPU usage in this case) of the machine usage in JDK 8 VS 49.45% in JDK 11 (for 100 requests) i.e., much improvement was seen in JDK 8 with G1 though overall, JDK 11 still performed better. All this data was recorded and profiled via *flight recorder* and *async profiler*. Table. IV showed that the gap between JDK 8 and 11 further narrowed down in terms of the stated parameters e.g., for 100 requests, *JVM+App* utilisation was 16.68% in JDK 8 VS 28.59% in JDK 11 i.e., 8 performed better than 11. Heap utilisation is also neck-to-neck (heap utilisation is slightly better with JDK 8 100 requests with v3). The only major difference left between JDK 11 and 8 is the total duration of GC pauses (759ms VS 1501ms in 100 requests) that is still far apart in 8 and 11.

For the configuration v4 (Table. I), it was observed that *JVM+App* usage was better in JDK 8 (difference of 14%) with 10 requests but as Table. V showed, JDK 11 took the considerable lead of more than 16% in 100 requests. Better heap utilisation and GC pause times validated that running THG Aggregator on the configuration v4 with JDK 11 is better than the current running configurations (v4 with JDK 8). From v1-v4, trend is pretty consistent in favour of JDK 11 and overall, JDK 11 always performed better than JDK 8.

It is pretty evident from the analysis so far that JDK 11 is pretty consistent in performing better than JDK 8 except few occurrences where JDK 8 performed slightly better in CPU utilisation therefore, we defined configurations v5-v8 to find out if these configurations can perform better than the configuration v4 on JDK 11. Table. VI is showing our analysis for configurations v4-v8 on JDK 11 for 100 user requests directed to running instance of THG Aggregator with stated configurations. These configurations were selected by combining the JVM flags that can help in reducing pause time, better heap utilisation and better machine usage.

According to Table. VI, configuration v6 showed best GC pause duration value of 512.38ms VS 706.70ms of v4. V6 also showed minimum value for the longest pause. The heap utilisation of v6 is slightly higher than v5 and v7. v6's *JVM+App* utilisation is 75.43% of the machine utilisation VS 36.48% in v4 which is the minimum value in the stated configurations. V6 is set up with 16 *ParallelGcThreads* and 4 *ConcGCThreads* as compared to 8 *ParallelGcThreads* and 2 *ConcGCThreads* in v4 therefore, v6 is utilising more CPU and a little bit more heap but producing the best results in terms of GC pause duration that can be really useful in case of thousands of requests at the price of higher machine utilisation that demands better hardware overall. V8 was the worst performer (v8 is the aggregation of all the GC flags from other configurations) with highest GC pause time, 49.16% of machine utilisation and 3.41 GB of heap utilisation. These results clearly stated that from GC perspective, v6 is the best configuration at the price of higher CPU utilisation and v4 has better machine utilisation at the expense of higher values for GC pause duration and heap utilisation. Overall, JDK 11 always performed better with v4 as compared to JDK 8 in terms of heap utilisation and GC total pause duration.

We went one step further and also analysed and compared 100 requests on running instances of THG Aggregator on *Arm* and *Intel*. For that purpose, we used *amazon linux2* on both instances with centos 7. One instance was

THG Aggregator 100 Requests (no-cache) JDK 11 AWS Graviton VS Intel																		
	V0		V1		V2		V3		V4		V5		V6		V7		V8	
	Arm	***Intel***	***Arm***	***Intel***	***Arm***	***Intel***	***Arm***	***Intel***	***Arm***	***Intel***	***Arm***	***Intel***	***Arm***	***Intel***	***Arm***	***Intel***	***Arm***	***Intel***
GC Young Config.	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New	G1 New
GC Old Config.	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old	G1 Old
GC Time Ratio	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%	12%
Max. allocated heap	3.89 GB	3.89 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB	10 GB
Max. Young generation size	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb	1.3mb
Max. heap used	998mb	877mb	1.73GB	1.43GB	1.46 GB	3.11GB	2.56 GB	1.45GB	1.54 GB	1.49GB	2.07 GB	1.51GB	2.55 GB	2.05 GB	3.79 GB	1.26 GB	2.35 GB	1.57 GB
Max. Heap value post GC	614mb	622mb	562mb	594mb	606mb	568mb	560mb	626mb	562mb	602mb	595mb	615mb	625mb	584mb	572mb	616mb	484mb	607mb
Total duration of GC Pauses	742.70ms	1187.77ms	722.76ms	1264.38ms	800.50ms	1030ns	796.70ms	1319.17ms	865.89ms	1095.06%	901.35ms	1260.59ms	922.78ms	1356.65ms	686.55ms	1294.8ms	989.95ms	1292.47ms
Longest Pause	80.63ms	95.49ms	153.55ms	201,02ms	179.43ms	254.93ms	242.12ms	259.72ms	240.76ms	222.30ms	208.28ms	304.54ms	283.64ms	260.95ms	177.28ms	195.47ms	265.26ms	259.50ms
Max CPU Usage: JVM + App	84.3%	92.7%	84.7%	94%	86.6%	93.8%	86.1%	91.3%	86.2%	93%	89.1%	95.8%	85.4%	94%	86.6%	99.3%	84%	96.3%

TABLE VII: THG Aggregator 100 Requests: JDK 11 (V0-V8) AWS Graviton VS Intel

running on Arm Graviton and the other one was on intel X86 architecture. Both instances spinned up with 4 vcpus and 16GB RAM. No other service was running on these instances during the profiling i.e., THG Aggregator was the only source of machine usage other than some basic OS services. Table. VII is showing the summary of this analysis. Overall, in all the configurations from v0-v8, machine utilisation was always higher in Intel as compared to Arm. Intel showed better readings for heap utilisation e.g., in v4, v6, v7 and v8 but this parameter alone cannot compensate the huge difference in total duration of GC pauses. In case of v6, the difference is 922.78ms(Arm) VS 1356.65ms(Intel) therefore, Arm in general and Arm v6 in particular still came out as the best overall configuration for THG Aggregator.

The parameters stated above are important in identifying the performance of a configuration but in the world of e-commerce, one important parameter of performance is the ability to entertain specific number of requests under defined time frame therefore, next section will evaluate THG Aggregator on v6 and v4 (on Arm based machines) to analyse how these parameters will be translated into actual performance of an e-commerce system.

B. Performance Analysis Via Load Test

THG Aggregator Load Test (Arm)						
	v4	**v5**	**v6**	**v7**	**v8**	**v6 Intel**
Time	10 mins	10 mins	10 mins	10 mins	10 mins	10 mins
Total Reqs.	52589	52235	53458	53008	52219	28335
Max. RPS	162	165.6	184.4	170.5	197.9	104.2
Avg. RPS	87.6	86.9	88.9	88.2	86.9	47.2

TABLE VIII: THG Aggregator Load Test: Max. 800 Users

For the load testing, the requests were ranging from simple user *GET* end points to the requests related to checkout basket etc. For all configurations, load test was conducted for 10 minutes, starting from zero users, spawning 10 users each second with maximum user limit to 800. We used THGs *Graviton*(in-house load tester) and *Locust* [10] for running the load test and getting the required data. Table. VIII is showing the results of our load test.

In the stated table, **Time** shows total duration for which we ran the load test, **Total Reqs.** shows number of requests entertained in defined time by a specific configuration, **Max. RPS** is showing maximum requests per second during the defined time and **Avg. RPS** is showing average requests per second during the load test. We ran the load test for configurations v4-v8 and as the table shows, v6 is a better performer i.e., better JVM metrics of v6 also translated into actual performance. V6 entertained 53458 requests with 184.4 **Max. RPS** and 88.9 **Avg. RPS**. Second closest candidate was v7 in terms of total requests but its average and maximum requests per second were lower than the v6. V8 is showing better value for **Max. RPS** but it cannot compete with v6 on other parameters and v8 was also one of the worst performers in JVM analysis.

The results were a little different with increasing number of users though e.g., when load testing was conducted for 1200 maximum users instead of 800, v4 showed slightly better results than v6 (52094 total requests on v4 VS 51870 on v6, 86.5 **Avg. RPS** on v4 VS 86.2 on v6). With 1500 maximum users, same trend between v4 and v6 still persisted as v6 has higher number of parallel and concurrent threads therefore, the slight decline in the values indicate the bottleneck in CPU usage in the running instance of THG Aggregator.

For comparison, we also ran *Intel v6* to compare it with *Arm v6* and the difference in the outcome is huge. *Intel v6* only entertained 28335 requests against 53458 in *Arm v6*. **Max. RPS** and **Avg. RPS** of *Intel v6* are also quite poor as compared to the *Arm v6* therefore, *Arm v6* seems to be the best configuration in our load test. Same trend was seen between *Arm* and *Intel* after changing user range multiple times between 400-1500. The user-range beyond 1500 was not realistic as in e-commerce, load balancers and orchestration tools are used to redirect load to different instances therefore, ceiling of 1500 was realistic for one running instance of an e-commerce platform.

IV. Related Work

There are various studies on JVM that analyse and propose options for fine tuning and optimisation. Our point of interest revolves around those approaches that investigate the performance tuning at JVM level in the domain of web-services in general and e-commerce in particular e.g., [8] investigates performance overhead of JVM in data parallel systems, [3] conducts a study to investigate ageing of JVM from memory depletion point of view and [4] looks into JVM enhancements for server-specific performance.

Few approaches are not the case studies but language oriented e.g., [5] discusses JVM from JIT (Just-In-Time) point of view to analyse JIT compiler abstraction management.

In the domain of e-commerce and micro-services, there are many case studies that include the performance enhancements and performance engineering of Java systems at JVM or framework level and these are the approaches that are closest to our approach e.g., [9] discusses the performance engineering of e-commerce systems but this approach proposes enhancement at the level of framework (EJB [2]), not at the level of JVM. [6] looks into performance enhancement of garbage collector in java based web-services. [1] proposes a simulation model to test and diagnose issues in production systems and [11] discusses the enhancements to reduce the run time overheads of JIT compiler to address the busy traffic at *Alibaba*.

Other than the optimisations based on JIT and overheads, few approaches try to tackle scalability issues via distributed JVMs. One such approach is JESSICA2 DJVM [7] that tries to cluster and scale the web application servers with distributed JVMs. *CHAOSMACHINE* [12] is another approach that does not tweak JVM itself but injects perturbations to JVM via try-catch blocks in order to extract an analysis of exception handling capabilities of a code base.

Our approach does not discuss any framework like EJB. The novelty lies in the fact that our approach involves three dimensions i.e., JVM, e-commerce and micro-services whereas, the approaches stated above involve one or two dimensions out of these three. Our approach also went one step further and analysed the implications of JVM tweaks in terms of e-commerce traffic i.e., number of requests handle by THG Aggregator under defined time frame with different configurations.

V. Conclusion and Future Work

In this paper, we presented the detailed analysis of an e-commrece system via our defined framework of analysis. The results identified the best configuration and concluded that *Arm* is the overall better performer under defined configurations to run THG Aggregator. As a part of the analysis framework, a comprehensive series of load tests showed how JVM configurations translated into actual performance of an e-commerce system.

After running and analysing THG Aggregator under different configurations, future work involves the analysis at code level to find out low-performant parts of the code base. In other words, code-based profiling should be conducted to extract call-graphs and heap dumps. Such a fine-tuning at code-level along with the JVM-based tuning will be really valuable in making an e-commerce system more robust with enhanced performance. Another dimension of future work revolves around the cost estimation of running the e-commerce operations. New *Arm* processors claim to be more energy efficient and as our results stated, performed better with THG aggregator. Further research can quantify the difference in running cost against the cost of replacing the existing hardware and further decisions can be made regarding future purchases and code optimisations against a specific hardware.

References

[1] Alberto Avritzer and Elaine J Weyuker. The role of modeling in the performance testing of e-commerce applications. *IEEE Transactions on Software Engineering*, 30(12):1072–1083, 2004.

[2] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0.* " O'Reilly Media, Inc.", 2006.

[3] Domenico Cotroneo, Salvatore Orlando, Roberto Pietrantuono, and Stefano Russo. A measurement-based ageing analysis of the jvm. *Software Testing, Verification and Reliability*, 23(3):199–239, 2013.

[4] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable jvms. *IBM Systems Journal*, 39(1):151–174, 2000.

[5] Malin Källén and Tobias Wrigstad. Performance of an oo compute kernel on the jvm: revisiting java as a language for scientific computing applications. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 144–156, 2019.

[6] Hai-Shuan Lam, GSVRK Rao, Chikkanan Eswaran, and Kok-Seong Ng. Performance comparison of various garbage collectors on jvm for web services. In *2006 International Symposium on Communications and Information Technologies*, pages 711–715. IEEE, 2006.

[7] King Tin Lam, Yang Luo, and Cho-Li Wang. A performance study of clustering web application servers with distributed jvm. In *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 328–335. IEEE, 2008.

[8] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your {JVM}: Understand and eliminate {JVM} warm-up overhead in data-parallel systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 383–400, 2016.

[9] T-K Liu, Santhosh Kumaran, and J-Y Chung. Performance engineering of a java-based e-commerce system. In *IEEE International Conference on e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004*, pages 33–37. IEEE, 2004.

[10] S Pradeep and Yogesh Kumar Sharma. A pragmatic evaluation of stress and performance testing technologies for web based applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*, pages 399–403. IEEE, 2019.

[11] Fangxi Yin, Denghui Dong, Sanhong Li, Jianmei Guo, and Kingsum Chow. Java performance troubleshooting and optimization at alibaba. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 11–12. IEEE, 2018.

[12] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. A chaos engineering system for live analysis and falsification of exception-handling in the jvm. *IEEE Transactions on Software Engineering*, 2019.