RadarScanner@Cloud Report

Diogo Moura 86976[1] - Diogo Meneses 86975[2] - Ricardo Martins 67869 [3]

Instituto Superior Técnico
Computação em Nuvem e Virtualização
31st May 2021

Abstract: Distributed systems and cloud computing are important concepts for the deployment of applications. In this work we tackled a radar scanner solver and deployed an application to run on Cloud VMs of AWS that answers requests of a simulated Raddar Scanner. The system implemented is centralized on one Master that receives requests and distribute them for several of workers that is smartly adjusted by the Auto-Scaler, an entity that manages them based on CPU utilization. The load distribution was possible via the implementation of instrumentation techniques of the solver using the BIT tools.

1 Introduction

The goal of this project is to design and implement a cloud-based app that serves radar scanning requests. As this service can be computationally intensive, our work was focused on developing a Load Balancer and Auto-Scaler that aimed to correctly match demand whilst economizing on cloud operation costs.

More specifically, the function of the Load Balancer is to evenly distribute requests among the available machines. To do so, it estimates request complexity based on previous ones. The Auto-Scaler is responsible for maintaining an adequate number of machines, based on CPU usage.

In order to evaluate the complexity of a request we instrumented the scanning itself, which was provided in the form of bytecode. By using results from different queries, it was possible to anticipate the computational complexity of a given request. This instrumentation was performed using BIT tools [1]. We verified that the number of instructions executed per query gave a good estimate on the complexity of each task and is accurately predicted based on the parameters of each query.

The application was developed to be run in AWS EC2 instances of type available for the free-tier service (t2.micro). In case of a real deployment, paid machines could be considered to reduce overhead on startup, improve performance of solving tasks. This would imply a economic analysis complementary that is out of the scope of the project.

2 Architecture

2.1 Structure

Figure 1 depicts how the application is organized. It contains several packages and can work as a Master or a Worker. The package pt.ulisboa.tecnico.cnv.solver contains the instrumented solver and is responsible for solving the queries. Then we have the master and workers packages and other complementary packages used with a set of utilities needed for the execution of complementary functions (i.e. packages utils and BIT).

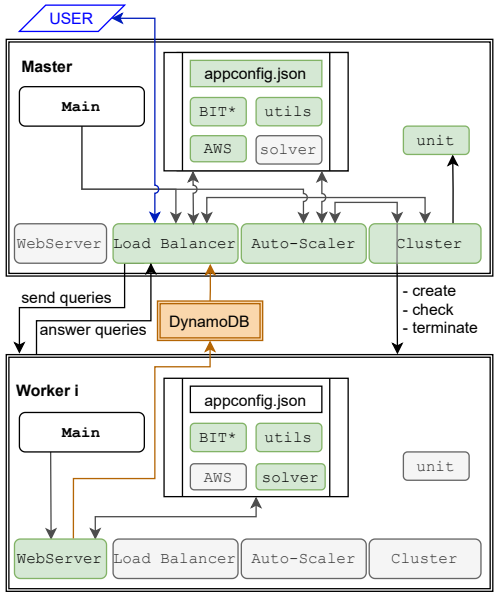


Figure 1. Structure of the application.

The Master is the main class of the application: it runs both the Load Balancer and Auto-Scaler on different threads of the same instance. The Load Balancer receives requests, distributes them, receives the results and redirects the response back to the user. The Auto-Scaler monitors the performance of each machine and is responsible for starting new instances in case of overloading, terminating instances in case of under-loading and rebooting instances if they become unresponsive. The master also contains a singleton class, called Cluster, which acts as the interface to manage units. These are objects of the class Unit which store information and methods that can act on their corresponding AWS instances. Only the Cluster can interact with this class. By doing so we abstract cloud-provider specific methods which

[1] diogo.g.moura@tecnico.ulisboa.pt
[2] diogoemeneses@tecnico.ulisboa.pt
[3] r.pedrasmartins@gmail.com

helps prevent lock-in.

Another important package developed, independent of the master previously explained, is the worker. It consists of the original WebServer class that we customized, the datasets, the ServerArgumentParser and also the customized BIT tool that is used to instrument the code, implemented by the class StatisticsTool.

Moreover the structure of the application also contains the package provided with the solver that was instrumented with the BIT tool, the package with the BIT tool and bit samples, complementary org classes and the package utils. The utils contains the classes that are used both by the master and workers. These class are the following: AppConfig loads the parameters of the application from a `appconfig.json` file, presented in Appendix A. To name a few: it defines the AMI Id which the cluster uses to launch units, Elastic IP allocation ID the load balancer gets associated with and the overload/underload thresholds for both units and cluster. This allows for their easy adjustment while avoiding the overhead of compilation. The class DeserializedParameters verifies the compliance of received queries and turns them into a more readable format, aiding in programming. The class DynamoConfig contains all the functions that communicate with the Dynamo database. The class UnitState and ClusterState enumerates all the possible states of the unit and cluster.

2.2 Deployment and Workflow

We prepared an AWS machine image (AMI) specially designed to host this application. The application is saved in a git repository and already compiled to avoid any associated startup overhead. Every time a new instance starts, the latest `appconfig` is always pulled before the program is launched.

Upon start, EntryPoint is called. After loading the appconfig, it tries to ping the IP of the Load Balancer therein defined. If this ping times-out, the new instance assumes the role of Master, otherwise it will become a Worker. The workers will be ready to solve queries sent by the Master as soon as the Auto-Scaler confirms they are operational through a similiar ping.

The application is assumed to be always up and ready to process any requests. That is ensured by the availability of one at least one master and 2 workers. Additionally, the master keeps track of the state of the available workers not only by the answer from the requests but also via a health-check with minimal load when the request remains unanswered. The workers are not able to close any communication channel with the master which means that if the health-check fails there might be a problem with the worker and it can be rebooted by the Auto-Scaler, if the problem persists.

The WebServer provides three endpoints, namely /scan, /test and /log. The /scan endpoint solves requests, the /test endpoint provides a health-check for the machine and the /log shows operation logs stored a text file to which the standard output was redirected to.

During run-time it is possible to deliberately shutdown the Master and in that case, it terminates all the workers that are running and the application is closed, via a shut-down hook.

3 Instrumentation

The instrumentation was developed using the available functionalities of BIT. Our custom tool obtained metrics representative of the complexity of a given request, while minimising overhead. The instrumentation results were first saved locally to perform the following analysis and after our strategy was delineated, the storage system was then deployed to DynamoDB.

The preliminary analysis of the solver consisted in it's static evaluation. This analysis reported that the solver package is composed of 76 methods, 1177 basic blocs and 3346 instructions. This indicates an approximate value of 44 instructions per method. Nevertheless, this static instrumentation does not take into consideration the real number of instructions or methods that are executed upon a query.

To understand the real complexity one query takes to be executed, we considered the dynamic instrumentation that counts the total number of methods, basic blocks and instructions that are executed. For this analysis, we selected only the classes that distinguish the scan strategy, namely all the classes with the filename ending in *ScanSolverStrategy.class*. The instrumented code was tested for 500 different queries with a uniform distribution of parameters chosen randomly. Figure B.1 in B represents the pairwise comparison of the instrumentation results for the total number of queries tested and the colors indicate the different strategy that was tested.

The number above the main diagonal indicates the correlation between the variables. From this result, we immediately conclude that analysing complexity based on the number of basic blocks or the number of instructions is the same as their correlation is 1.00. Moreover, the correlation between the number of methods and the number of instructions is relatively high, 0.80 and from the in-plane representation of the data points we observe that the strategy influences this linear dependence.

For the purpose of our application, we decided to analyse the complexity of each query only based on the number of methods executed. Further on, whenever load and complexity are referred, we're alluding to this metric. This methodology reduces the

quantity of instrumented instructions introduced in the code which allows for fast scan requests without compromising estimations of complexity due to its correlation with the number of instructions being very high (0.8). We considered this as a good balance between performance and overhead, since the number of methods is 13 times lower than the number of basic blocks for the Grid Scan strategy and around 100 times lower for the other strategies (see Figure B.1 in Appendix B).

Figure B.2 presented in Appendix B shows all the parameters of each query, except the map type, with the additional calculation of the search area and number of methods. The lower row of plots indicates the relationship between the variables with the number of methods. From the analysis of these data we foresee a linear dependence on the number of methods with the search area. This is explored in Figure 2. For grid, progressive and greedy range scan the linear estimators of the number of methods (nm) based on the search area (A) are presented in Equations 1, 2 and 3, respectively. The R^2 for the linear regression presented ranges from 0.38 for the progressive scan to 0.70 for the grid scan. As these values are relatively lacking, this methodology to estimate complexity is only employed when there are no similar recorded queries.

$$nm_{grid} = max\{0, 391.96 \cdot A - 6068305.35\} \quad (1)$$

$$nm_{prog} = 4.55 \cdot A + 409230.34 \quad (2)$$

$$nm_{greedyrng} = 3.14 \cdot A + 195721.04 \quad (3)$$

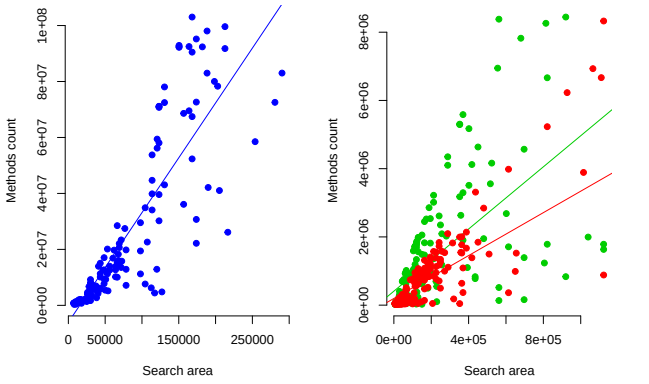


Figure 2. Number of methods per search area for the different scan strategies: grid in blue, progressive in green and greedy range in red, with representation of the linear regression. Separate plots were considered because the order of magnitude of the number of methods is different in both cases.

4 Storage

The storage system used by our application is another service provided by AWS, the NoSQL database DynamoDB.

This DB is used to store all previous requests and their instrumented complexity:

For each query, the timestamp is used as the primary key, with a sorted key based on the strategy. All the parameters of each query are saved (mapType, width, height, search coordinates and starting point) along with the number of methods executed. An index was also created on the mapType key.

The interaction with the database is done pragmatically using the AWS SDK in the class utils.DynamoConfig.

5 Load Balancer

The Load Balancer is the class that forwards the requests of clients to the multi-threaded workers. The following sequence of operations outlines the workflow of the Load Balancer:

1. Receives a query from $client_i$
2. Identify and validate the arguments of each query
3. Estimate the complexity
4. Assign a new worker based on their previous load
5. Send request to the worker
6. Wait for the result
7. Send the result to the client
8. Free worker load

All of these events are logged so that the developer can easily identify any issues that arise and what may have caused them. This log file is accessible through the EC2 instance via a http connection to the link IP:port/log.

Each step of the process will now be explained in detail.

5.1 Methodology

Receiving a request and reading the parameters

The requests to the application can be sent via http post to the IP address of the Load Balancer. When the request is received a thread is allocated for that request and the parameters are deserialized by the class *DeserializedParameters*. If any exception is thrown during this phase, we can safely assume there is an issue with the client's queries, so an image is sent to warn them of it.

Estimate the complexity

After processing the parameters, the complexity of the task has to be estimated. Recall that this complexity/load is defined to be the number of methods executed by the solver to answer the request.

For this procedure, we define two distinct strategies. The first approach requeries similar requests to be available in the DynamoDB. The second uses the regression obtained during the pre-analysis of the instrumentation of the solver.

In more detail, the first estimation strategy, implemented in the method *scanLoad*, retrieves from the database all requests with the same strategy and map type. Then we store the search area (A), starting coordinates (XS, YS) and complexity in a priority queue, ordered by request similarity, defined further ahead. If during this process a request with the same area and starting points is found, we empty the queue and use the load of that request. Otherwise, a similarity measure is given by Equation 4 for each previously stored query, i. Index q refers to the query being estimated. W and H represent the width and height of the map, respectively.

$$S_i = 1 - \frac{1}{5} \cdot \Big(3 \cdot \frac{|A_i - A_q|}{W \cdot H} \\ \frac{|XS_i - XS_q|}{W} + \frac{|YS_i - YS_q|}{H}\Big) \tag{4}$$

After evaluating the similarity for all the data points available in the database the 3 most similar points are selected and the complexity of the task in terms of methods (nm) is estimated by Equation 5

$$nm = \left(\sum_{i=1}^{3} S_i \cdot nm_i\right) \Big/ \sum_{i=1}^{3} S_i \tag{5}$$

When $\frac{1}{3}\sum_{i=1}^{3} S_i < 0.6$ we have empirically determined that this estimation is usually not as accurate as the second strategy. This second strategy is implemented in function *estimateComplexityHeuristic*. It predicts future load based on the linear regressions presented in Equations 1-3.

Assigning a new worker

The strategy to distribute the load aims at maximizing usage of allocated machines without compromising performance. A dynamic load balancing strategy was implemented, which uses the current information of the workers, in this case current load.

The strategy to distribute the tasks among the several workers consists of the following heuristic Algorithm 1. To select a Worker for a given task, the available workers are sorted in a priority list by descending order of work load. If no worker is available then the thread will block waiting for one. Availability is defined to be $CPU < 90.0\%$ and responsive. During this waiting time it updates a cluster variable `waitingLoad`, so that the Auto-Scaler is properly informed. Then we try to fit the received load in a worker, if it fails, we wait for one to become less loaded.

Algorithm 1: Load distribution

```
while online do
    receive a task ;
    load:= max{estimated load, 90% of
     maxAcceptableLoad};
    cluster.waitingLoad += load;
    while true do
        workers_list:=
         cluster.getAvailableWorkers();
        for worker in workers_list do
            if worker.load + load <
             maxAcceptableLoad then
                cluster.waitingLoad -= load;
                return reserve worker;
            end
        end
        No worker reserved;
        Sleep;
    end
end
```

By ordering them in decreasing load, we use our allocated machines to the maximum and allows the Auto Scaler to more easily free machines, if it deems so. By being conservative with the `maxAcceptableLoad` we are ensuring that no worker will become over loaded.

Route the request, wait for response, send the answer to the user

After a request has been assigned to a worker it is routed via http post to the unit's public IP, as provided by AWS.

The Load Balancer indefinitely waits for the response of each query, as some requests are very large.

After receiving the answer from the worker it is forwarded to the client and the thread is released.

Exception occurs

If there is any exception during these last two sections, a variable `attemptNumber` is increased and a health-check is performed on the failing unit. If it also fails the health-check, it will be marked as unresponsive and if $attemptNumber < 15$, and the process restarts from *Assigning a new worker*. On the other hand, if the maximum number of attempts has been achieved, the client shall receive a gif explaining that there was a crash.

6 Auto-Scaler

The Auto-Scaler is the entity responsible for managing the number of instances running. It has the re-

sponsibility of creating new instances when the workload is high for a long period and terminating them otherwise. For the Auto-Scaler to operate it is in constant communication with the AWS using the SDK for Java and the available functions to analyse the condition of the instances.

This Class is executed on the Master in a different thread from the Load Balancer. It keeps the number of instances running appropriate to the current average cluster load and CPU utilization. For this implementation and regarding that at the moment there is no information regarding specific periods, e.g., day time or week, where it is expected to have more traffic, the algorithm implemented for the Auto-Scaler only takes into account the current usage of the system, by the analysis of the CPU utilization and the number and complexity of requests that arrive. Before going into details of the implemented algorithm it is important to understand the life-cycle of an unit.

6.1 Unit life-cycle

As the Auto-Scaler is running in the master, every EC2 instance it creates will become a worker that solve tasks assigned by the Load Balancer. Figure 3 depicts a diagram with the life cycle of one of these instances that can be classified in 7 different states: *created*; *startup*; *running*; *overloaded*; *unloading*; *unresponsive* and *rebooting*. In the diagram the states are represented int he rounded boxes with the arrows representing the possible transitions between states.

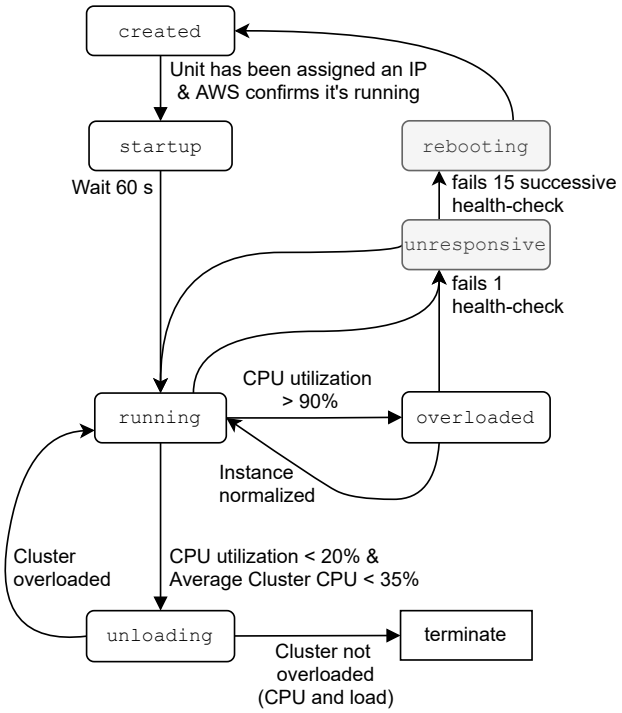


Figure 3. Unit life-cycle diagram.

The Auto-Scaler is responsible for managing all workers of the application. To do that it periodically (every 10 s) acts on the state of the cluster and its units. This period corresponds to the `mainLoopPeriod` defined in the appconfig A.

If the Auto-Scaler determines that it needs additiona units, it requests AWS to create a new instance and the unit is registered with the state *created*. As AWS takes some time to provision the instance, every loop of the Auto-Scaler, it checks if the IP address was provisioned and when it is, it will be pinged until it responds successfully. When it does, it enters state *startup*. This state allows the instance to receive load before the Auto Scaler deems it under loaded. If some problem occurs with the instance during these phases, it is automatically rebooted.

After 60.0 s, that corresponds to the startup time, the state is changed for *running* and the unit is healthy and available to receive requests. *Startup* and *running* are the only two states that can receive requests.

If the workload of a certain instance exceeds the threshold of a CPU utilization larger than 90.0 % the instance is classified as *overloaded*, forbidding the Load Balancer to route traffic to it until it becomes normalized, which happens when it's CPU load decreases. If the CPU utilization of one unit is kept under 20.0 % and the average CPU of all workers is bellow 70.0 %, the instance enters state *unloading* so that it can wind down for termination, which happens when it has served all its requests and if no more units where requested during that time.

The *unresponsive* state occurs when an instance fails a health-check. If it keeps failing them for more than 150.0 s, the instance is rebooted, briefly changing it's state to *rebooting* before it enters state *created*. After being rebooted the information of the instances needed to send requests, namely the IP remains unchanged.

6.2 Cluster

The class Cluster was already addressed when the structure of the application was explained in Chapter 2. As indicated before, the Auto-Scaler is responsible for analyzing the global state of the system so it can decide when to launch or terminate instances. To make this verification easier, we classify the cluster state in 3 different states: *underloaded*, *overloaded* and *normal*. This state is evaluated every 10 s, taking into consideration all the instances in the system.

6.3 Adjustment strategy

In this section we describe the strategy applied do adjust the number of units. One of the parameters considered in the analysis is the CPU utilization percentage that is provisioned by the CloudWatch av-

eraged over a 60 s period. To improve the reliability of the system and its response time to incoming requests, the application considers not only the CPU Utilization but also the load factor of the requests in the system.

While the CPU Utilization measures the current performance of the Units averaged on the last 60 s, the load factor quantifies the total amount of estimated load. This load takes into account the requests that are already being solved and the ones waiting for a worker. The total load is evaluated based on the estimated complexity of each request.

The number of instances added or removed from the system at a given time is evaluated by the Auto-Scaler based on the current metrics to obtain a CPU utilization of 50%. For precaution measures the maximum number of instances allowed in this program is 10.

The available instances in the system shall increase either when the CPU Utilization of the Cluster is higher than 70.0 % or the average load per instance is 90% higher than $8 \cdot 10^7$, whichever occurs first. The maximum load per instance was tuned with the final version of the application based on the number of methods that would make one instance at maximum CPU Utilization.

When the workload decreases on the system, due to reduction of the number of requests or its complexity, the Auto-Scaler is responsible for terminating units. The unloading strategy is slightly different than the strategy to increase units. This considers only the cluster state to be *underloaded* (< 35.0 %), which will only occur if the total load in the system is small for the the number of units available. In this case it will decrease units that are set to *running* for the CPU to be 50 %. These units assigned to be terminated will first finish all the requests they are processing. In this period, they are kept inactive, meaning in the state *unloading* they do not receive more requests. The units to create or delete are obtained from solving this equation by solving for `unitDifference`:

$$\frac{\text{totalCpu}}{\text{currSize} \pm \text{unitDifference}} = 50\% \qquad (6)$$

The plus is used when the cluster is overloaded and minus when it is underloaded.

7 Results and tests

We tested our application using JMeter [2] to check its capacity to answer requests and also increase/decrease the number of units in AWS accordingly with the intensity of requests.

From the analysis of requests tested, the Application showed a good distribution of requests and low latency for small queries. Additionally, the cluster size was correctly scaled with the number of requests. These results can be found in Figure C.3 in Apendix C.

8 Conclusion

This report documents the analysis and implementation of an application that resembles a Radar Scanner. The solver is provided as an execution class that was instrumented using the BIT tools available in JAVA. This methodology allowed to define a metric on the complexity of each request that consisted of the number of executed methods. The deployment of the cloud application considered a distributed system of solvers that are recruited and terminated according to demand. The techniques implemented showed a good distribution of the load among several workers and good scaling policies.

Further developments could be made on the load distribution, namely a comparison on the predicted versus real number of methods executed to refine the estimator and for large requests, a intermediate check-up could be implemented to analyse the progress of each request.

Moreover regarding our heuristic, the number of queries solved is not sufficient to obtain a precise measure of the complexity of each request. If more data is measured, we suggest the implementation of a machine learning algorithm to predicts the complexity of each request.

Acknowledgements

The authors thank the teaching staff Professors Luis Veiga, Rui Cruz and Rodrigo Bruno for the outstanding support during office hours and deadline extension.

References

[1] Ben Zorn and Han Bok Lee. Bit: A tool for instrumenting java bytecodes. Advanced Computing Systems Association, December 1997.

[2] Apache jmeter. `https://jmeter.apache.org/`. Online; accessed: 2021-05-30.

A Configurations: `appconfig.json`


```
{
  "autoscaler": {
    "mainLoopPeriod": 10,
    "clusterOverLoadThreshold": 70.0,
    "clusterUnderLoadThreshold": 35.0,
    "unitOverLoadThreshold": 90.0,
    "unitUnderLoadThreshold": 20.0,
    "unitFailedPingsThreshold": 15,
    "unitStartupCountThreshold": 6
  },
  "loadbalancer": {
    "minSimilarity": 0.6,
    "maxAttempts": 15,
    "elasticIp": "34.194.90.215",
    "port": "8000"
  },
  "cluster": {
    "maxUnitLoad": 8e7,
    "credentialsFile": "credentials_XXXX",
    "minSize": 2,
    "maxSize": 10,
    "securityGroup": "CNV-ssh+http",
    "elasticAllocationId": "eipalloc-0829cf29b4593ef48",
    "imageId": "ami-07e405262f768ac18"
  }
}
```

B Instrumentation complements

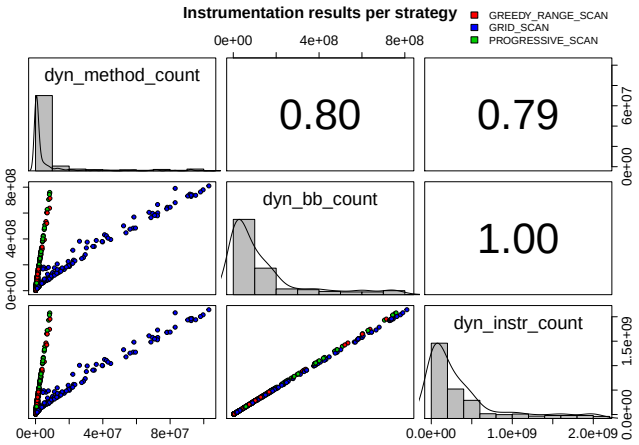


Figure B.1. Results from instrumentation (number of methods, number of basic blocks, number of instructions) grouped by scan strategy for the simulated 500 queries. The values above the histogram indicate the correlation between the variables.

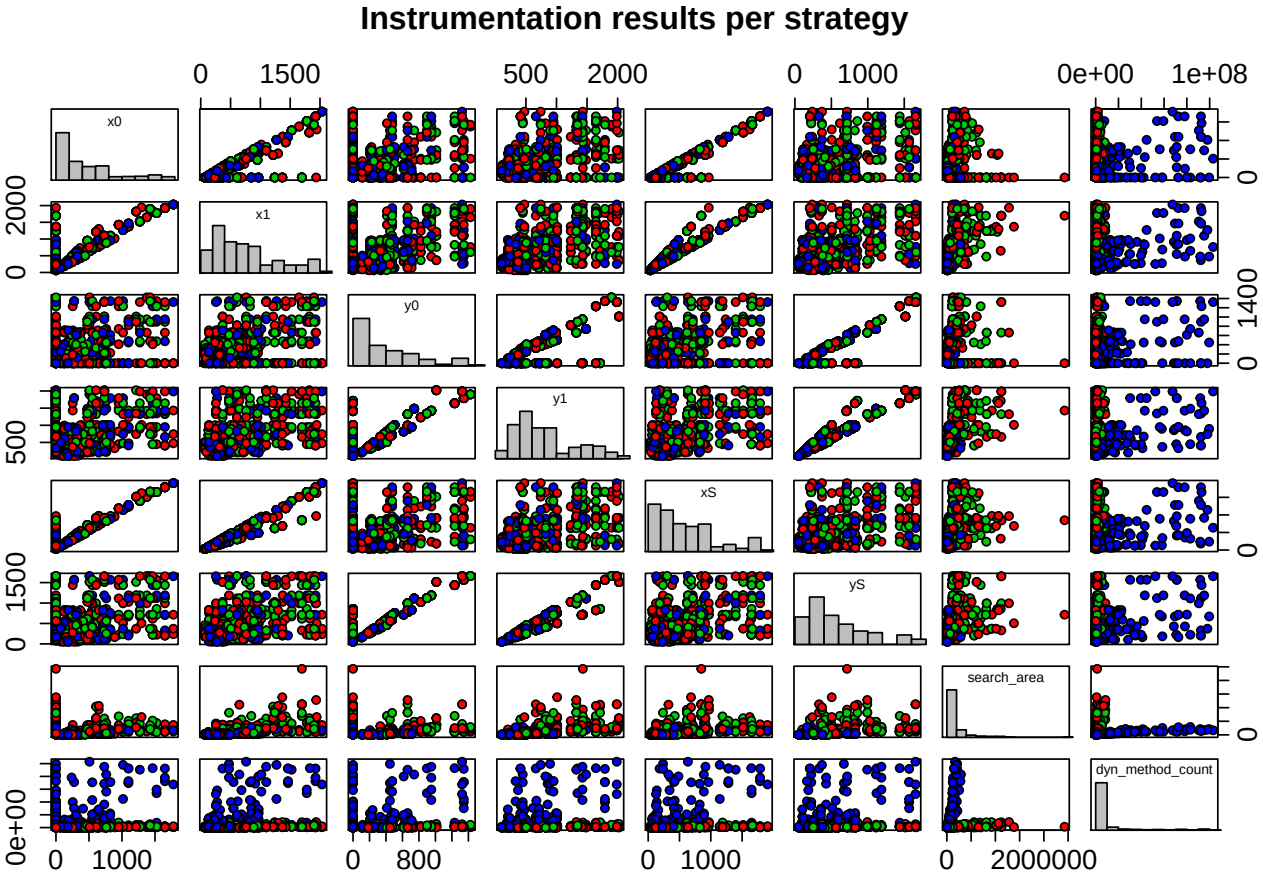
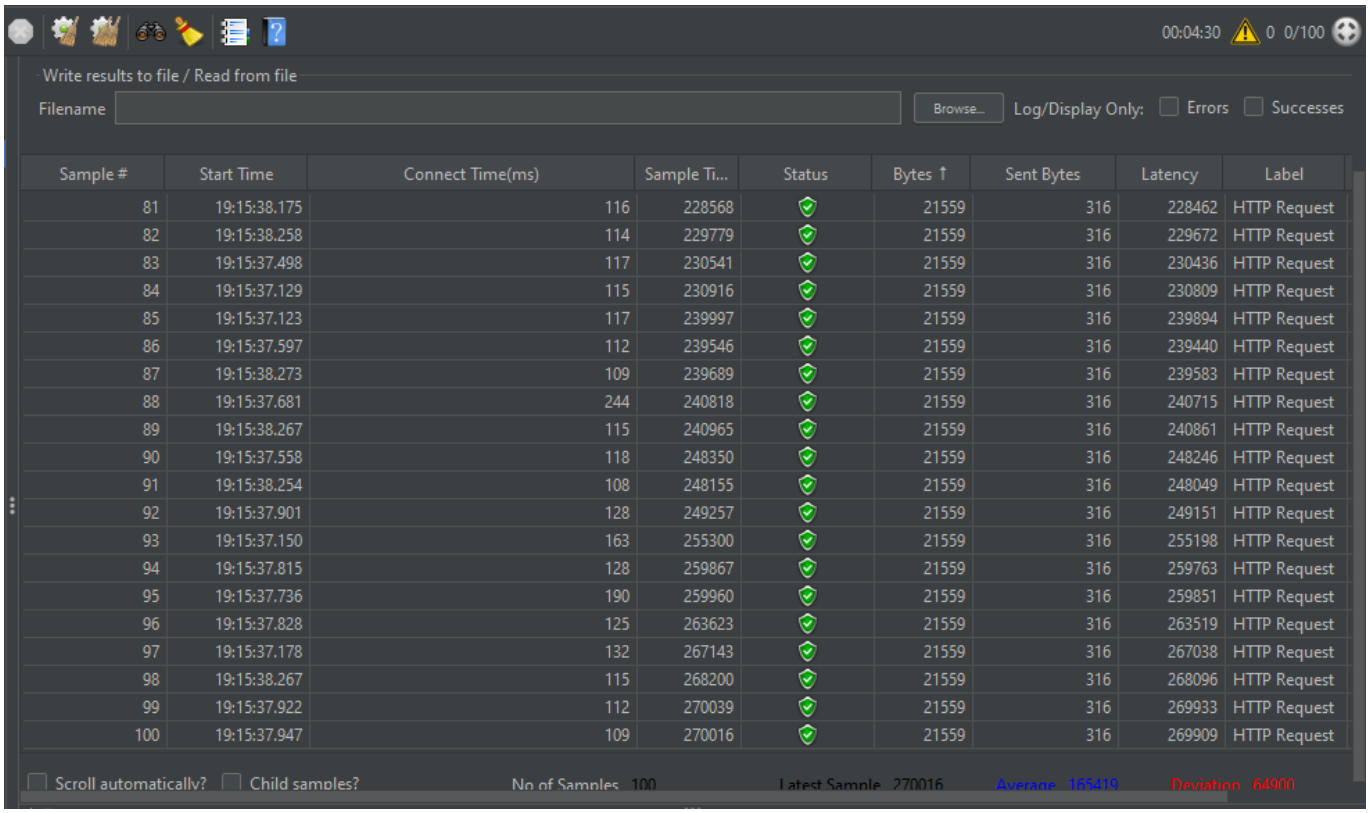


Figure B.2. Metrics by scan strategy

Table B.1. Average ratios of the results from the instrumentation of 500 queries divided by scan strategy

	basic blocks/methods	instructions/basic blocks	instructions/methods
Grid	13.6	2.6	35.9
Progressive	102.3	2.8	286.0
Greedy range	98.5	2.8	277.5

C Results from tests with JMeter



Sample #	Start Time	Connect Time(ms)	Sample Ti...	Status	Bytes	Sent Bytes	Latency	Label
81	19:15:38.175	116	228568		21559	316	228462	HTTP Request
82	19:15:38.258	114	229779		21559	316	229672	HTTP Request
83	19:15:37.498	117	230541		21559	316	230436	HTTP Request
84	19:15:37.129	115	230916		21559	316	230809	HTTP Request
85	19:15:37.123	117	239997		21559	316	239894	HTTP Request
86	19:15:37.597	112	239546		21559	316	239440	HTTP Request
87	19:15:38.273	109	239689		21559	316	239583	HTTP Request
88	19:15:37.681	244	240818		21559	316	240715	HTTP Request
89	19:15:38.267	115	240965		21559	316	240861	HTTP Request
90	19:15:37.558	118	248350		21559	316	248246	HTTP Request
91	19:15:38.254	108	248155		21559	316	248049	HTTP Request
92	19:15:37.901	128	249257		21559	316	249151	HTTP Request
93	19:15:37.150	163	255300		21559	316	255198	HTTP Request
94	19:15:37.815	128	259867		21559	316	259763	HTTP Request
95	19:15:37.736	190	259960		21559	316	259851	HTTP Request
96	19:15:37.828	125	263623		21559	316	263519	HTTP Request
97	19:15:37.178	132	267143		21559	316	267038	HTTP Request
98	19:15:38.267	115	268200		21559	316	268096	HTTP Request
99	19:15:37.922	112	270039		21559	316	269933	HTTP Request
100	19:15:37.947	109	270016		21559	316	269909	HTTP Request

Figure C.3. Results from testing with JMeter.