Computação em Nuvem e Virtualização

Sudoku @ Cloud

Grupo 14

João Rafael Soares
87675

Miguel Barros
87691

Introduction

This report presents our solution to the CNV project. The goal is to design and develop an elastic cluster of Web Server capable of solving a set of Sudoku puzzles on demand. Each request can result in a task of varying complexity to process, as different solving approaches will take a different number of steps to find the missing elements in the sudoku puzzle. To have scalability, good performance and efficiency, the system attempts to optimize the selection of the cluster node for each incoming request and to optimize the number of active nodes in the cluster.

Architecture

The architecture is divided into the Load Balancer, Auto Scaler, Web Servers and the Metrics Storage System.

The Load Balancer distributes the incoming workload equally between all available Web Servers, while the Auto Scaler keeps tabs on the total workload of the system and decides if more instances of Sudoku Solvers are needed or if some can be taken down to save costs. Both of these components run on the same EC2 Instance.

First, all requests arrive at the Load Balancer. It will then forward each request to an instance in a way that maintains an even workload between all Web Servers. We will see how this decision is made in future sections.

Once the request has been completed by the instance, it returns the response to the Load Balancer, which will then return the response to the client.

The Web Servers are responsible for uploading the instrumentation taken for each request to the Metric Storage System.

The Metrics Storage System is a set of Tables on top of DynamoDB storing the cost (in terms of estimated number of instructions ran) of requests executed by the Web Server EC2 Instances.

Instrumentation

On a first approach, we instrumented the whole package solver for all metrics available in order to study the behaviour of each type of solving technique. We tested these solvers by running three puzzles of each size with each of the available solving techniques (except for the 25x25 puzzle size, which only has 2 puzzles available) and with an ample sample size of unsigned entries. We did this to understand how the solver performance is affected by different puzzle sizes and number of unassigned entries. (this is not the only thing that affects performance but it is the only one we can control). We saved these results in JSON files and generated graphics from these results. We calculated the correlation between Instruction Count and the other metrics taken and selected those with a high correlation. Using that, we chose the set of metrics with the least amount of overhead (measured in instrumentation function calls) to be the one instrumented during runtime, thus giving us a close approximation of the cost that puzzle will have on the system while giving us the least amount of overhead from instrumentation.

The metrics chosen were:

BFS - Method Count
CP - New Count
DLX - Method Count

To reduce the overhead even further we tried to instrument the Solver Classes, but could not get a sufficient correlation between any metric taken for the class *SudokuSolverDLX* and the Instruction Count for a DLX Sudoku Request (since it is by far the least consistent strategy in terms of performance w.r.t the request parameters).

The correlations obtained and metrics chosen can be seen in Annex A through C.

Web Servers

Each WebServer is responsible for attending requests, solving the Sudoku puzzles and instrumenting the request complexity while it does so. It also asynchronously uploads it's results to the MSS (as to not impact the latency perceived by the client), so that the Load Balancer can use these values to estimate the complexity of future requests. Since the server is multithreaded, care was taken to ensure that the metrics of each request don't interfere with each other. We do this by mapping each Instrumentation Object with the thread executing the request. Since threads may be reused by the Web Server, after each request is finished, the metrics associated with the current thread are reset.

Metrics Storage System

The collected metrics must be permanently stored in a way accessible to the Load Balancer that uses them to decide which Web Server to forward each request. Amazon DynamoDB was chosen for this purpose. Each entry in the database will have the parameters of the puzzle, the values of the metrics obtained and the cost associated with that puzzle (as can be seen in Annex E). A table is created for each strategy by the Load Balancer when the system starts.

Request Cost Estimation

In order to estimate the cost of a request, we calculated a linear regression for the obtained metrics in order to calculate how each aspect of a puzzle influences its cost. We arrived at the following formula by performing a linear regression of the variable n1(Width), n2(Height), un(Unassigned entries) of the request and the value of the metric obtained:

$$InstructionCost = N1 * N1\,Slope + N2 * N2\,Slope + UN * UN\,Slope + Metric * Metric\,Slope + Intercept$$

These values were all adapted to each of the three strategies, as each *Slope* and *Intercept* have different values for different strategies. We leave the exact values in Annex F.

In general, since DLX is the least consistent strategy w.r.t the request parameters, it is also the one whose estimation is the least correct. From the data we collected, requests with this strategy have their cost overestimated, however since they tend to not be the longest running requests in the system, it is something we can cope with. To mitigate this, a deeper inspection of what affects the performance of the algorithm would be needed, which would be out of the scope of this project.

Since requests can be long running, we must periodically update the cost of each request in each server to account for the already computed cost of requests. This is done by periodically subtracting the cost of each running request of each server taking into account the time passed, since it is expected for the cost to be a direct correlation between the cost in CPU and time of each request. We also take in consideration the number of requests running in the instance, assuming that the CPU is evenly split between all requests, it will slow down the answer time of that request.

We also insert a minimum limit on the cost of requests, in order to avoid cases where we would remove too much cost from a request and it would become negative. As such, we limit the loss to 99% of the original cost.

The cost to be removed from an instance is then given by the formula:

$$Max(cost - \frac{Cost\ Loss\ Slope}{Number\ of\ requests\ running} * TimeElapsed,\ minCost)$$

Being the *Cost Loss Slope* the estimated number of instructions executed per

millisecond (10000 - See annex D) and *TimeElapsed* the last time since the cost was updated.

Load Balancer

When the system starts, the Load Balancer creates the DynamoDB tables and creates 2 Web Server instances (this number can be configured). After the two instances are running, the Load Balancer can start receiving and forwarding requests. Finally it also starts the Auto Scaling Thread (more information in the next section).

It is up to the Load Balancer to detect failed instances and notify the Auto Scaler, as such periodic health checks are done on each instance (by way of a HTTP GET request to a dedicated endpoint on the web server and a response timeout). If an instance fails 5 consecutive health checks, it is considered dead by the Load Balancer and the Auto Scaler will eventually take down the instance.

Once an instance fails an health check, it is considered unhealthy and no more requests are sent to it until it passes a subsequent health check.

When a new instance is created, it starts as unhealthy until it passes the first health check. A grace period (40 seconds) is provided before health checks start being sent so that the server has time to start on that instance.

When the load balancer receives a request, it:

- Estimates its cost, as seen in the Request Cost Estimation section;
- Forwards the request to the instance with less work and awaits its response;
- Forwards the obtained response to the client.

To decide which instance has the least work, the Load Balancer keeps track of what requests each instance is running and estimates the cost of each one. It then chooses the one whose sum of request cost is the lowest.

Fault Masking

To perform fault masking, when a server crashes, the requests that were running on it are redistributed between the other instances. If no other instances are available, they wait in a queue for instances to become available (which will happen when the Auto Scaler creates more instances or when an instance responds to an health check).

Auto Scaling

Since the Load Balancer can only distribute work between the running instances, it is the job of the Auto Scaler to detect when the computing nodes are overloaded and launch new instances to handle the demanding workload. Conversely, it must also shut down instances when demand is small to spare resources. Launching new instances is simple: as every instance in AWS has the same computational power, we just detect when the load on the system, as a whole, is higher than the capacity of the current running instances and launch new instances until we can cope with the current running cost.

For this purpose, we create a thread in the Load Balancer that periodically checks the current state of the system and makes decisions based on the recent seen loads of the system.

We take in consideration various factors when deciding if a scale up or scale down is in due:

Number of instances running:

We wish to keep a minimum number of instances of 2 at all times. As such, when the Auto Scaler detects that there is less than the

minimum number of instances, it creates a new one until this minimum has been reached.

Average cost of the system:

We periodically measure the total load of the system and the average number of requests per instance (every 5 seconds). We calculate the average of the last 10 checks.

If both the average system load and average requests surpass a certain threshold (2E9 instructions and 3 requests respectively), we meet the condition to scale up and as such create another instance.

In the same way, we use these two averages to determine if a scale down is necessary, this is if the average cost and number of requests are below a certain threshold (8E8 instructions and 1 request respectively).

We take in consideration the average number of requests to avoid edge cases where either a very complex request with a high cost would lead to unnecessary scale ups or a lot of low cost requests would lead to unnecessary scale downs.

The thresholds were decided iteratively, using as a benchmark the workload in annex G.

Scale cooldown timer:

Whenever an instance is added or removed, a cooldown of 3 minutes is activated in order to avoid the Auto Scaler adding/removing instances in a short amount of time due to the system not yet adapting to the previous change. We used 3 minutes as a representative time of a system adapting to the new changes (the time for an instance to be operational and for existing requests to progress in a significant way).

Choosing which instance to shut down:

When a scale down is due, the Auto Scaler asks the Load Balancer which instance should be removed.

This decision is based on how much paid value we can get out of an instance before shutting it down, since we have already paid for the reserved time. As such, we estimate how long it will take to complete all current requests of an instance and calculate how close it is from the hourly payment. We choose the instance closest to this period, as it is the one that we will be able to extract the most out of.

This instance is then removed from the Load Balancer in order to not receive any more requests and once it has completed every pending request, it is terminated by the Auto Scaler.

Scaling up and scaling down are performed on separate threads in order to not collide with the scale monitoring task.

Annex A - Correlation Between Instruction Count and other Metrics

Metric	BFS	CP	DLX
A New Array Count	NaN	NaN	NaN
Basic Block Count	0.999	0.999	0.981
Branch Count	0.999	0.999	0.997
Field Load Count	0.983	0.997	0.762
Field Store Count	NaN	NaN	0.891
Load Count	0.999	0.999	0.940
Method Count	0.953	0.966	0.861
Multi New Array Count	NaN	NaN	NaN
New Array Count	NaN	NaN	0.763
New Count	0.715	0.999	0.769
Stack Depth	-0.072	0.611	0.910
Store Count	0.999	0.999	0.993

Annex B - Maximum number of instrumentation calls observed for each metric per strategy for a request

Metric	BFS	CP	DLX	Average
A New Array Count	2	2	3	~2.33
Basic Block Count	~2E9	~4E9	~2E9	~2.67E9
Branch Count	~6E8	~1.4E9	~7E8	~9E8
Field Load Count	~75000	~160000	~4.5E7	~1.51E7
Field Store Count	7	7	~80000	~26671
Instruction Count	~2E9	~4E9	~2E9	~2.67E9
Load Count	~6E8	~1.4E9	~1.2E9	~1.06E9
Method Count	~9000	~12000	~37000	~19333
Multi New Array Count	1	1	2	~1.33
New Array Count	0	0	~600	~200
New Count	38	~5000	~8000	~4346
Stack Depth	~18000	~24000	~74000	~38667
Store Count	~6E8	~1.3E9	~7E8	~8.67E8

Notes:

- Instruction Count and Basic Block Count always have the same number of calls. To count the number of instructions we place a call before any BasicBlock;
- Stack Depth always has two times the number of calls of Method Count since it involves calling a instrumentation function before a method is called and after it returns;
- Average was taken assuming a even distribution between strategies;

Annex C - Metrics Chosen for each Strategy

BFS	CP	DLX
Method Count	New Count	Method Count

Criteria: Metric with correlation of at least 0.85 w.r.t Instruction Count and with minimum average number of instrumentation calls.

Annex D - Time measured to complete requests

Strategy	Board	UN	Measured Cost	Request Time(ms)
BFS	16x16_03	128	478035945	48177
BFS	16x16_03	256	1587483337	140896
BFS	9x9_103	40	214704141	15459
BFS	9x9_101	80	314842652	23227
BFS	9x9_101	81	457933376	33670
BFS	9x9_101	1	25015522	302
CP	16x16_05	625	8955295620	746577
CP	9x9_03	40	166756590	13843
CP	9x9_03	81	349983375	29024
DLX	9x9_03	40	726535911	43710
DLX	9x9_03	81	712860032	45409

Notes:

- Each measure was taken with a single request running on the instance.
- We chose to remove the outliers (the DLX values) since they are the ones whose cost has the most error. We used the average to choose the Request Cost Loss Slope, but decided to cap it at 10000 (slightly below average) since we would rather overmeasure the system load instead of undermeasure.

Annex E - DynamoDB Schema

RequestQuery	UnassignedEntries	BoardIdentifier	BoardSizeN1	BoardSizeN2	RequestCost

Caption:

- RequestQuery - Request Query associated with each request:
- UnassignedEntries - *un* parameter of the query associated with each request
- BoardIdentifier - *i* parameter of the query associated with each request
- BoardSizeN1 - *n1* parameter of the query associated with each request
- BoardSizeN2 - *n2* parameter of the query associated with each request
- RequestCost - cost measured by the instance (in terms of number of instructions) upon completion of the request.

Annex F - Linear regression between the metric chosen for each strategy and measured instruction count of the requests

Strategy	N1 Slope	N2 Slope	UN Slope	Metric Slope	Intercept	Coefficient of Determination
BFS	-41030543.22	-41030543.22	-72864.72	617084.72	489035592.21	0.99
CP	-1026671.74	-1026671.74	184.75	2058642.81	-26810917.90	0.99
DLX	-182842646	-182842646	-651744.76	310610.87	2943278992.67	0.86

Annex G - Workload to observe scale up and scale down

Board	Strategy	UN
16x16_06	BFS	625
16x16_06	BFS	625
16x16_06	BFS	625
16x16_06	BFS	625
16x16_06	BFS	625
16x16_06	BFS	625

Notes:

- We recommend waiting for the scale period to finish before issuing the requests, or the system load will decrease below the Scale Down Threshold before it expires and the scale up will not be observed.
- Scale down will happen close to the requests completing.

Annex H - Auto Scaler values

Variable	Variable
Scale Cooldown	3 minutes
Number of recorded system load measurements	10 measurements
Time between system load measurements	5 seconds
Scale up request cost threshold	$2.5 * 10^9 \ cost$
Scale down request cost threshold	$8 * 10^8 \ cost$
Scale up average request threshold	3 requests
Scale down average request threshold	1 requests