Job Scheduler with Improved Turnaround Time

Name: Jaime Sun  
SID: 45662398

Introduction:

This project is to design and implement a new job scheduling algorithm for a server simulation that can schedule jobs using an algorithm that we have created. The job scheduler algorithm is to take into account three different performance metrics which are average turnaround time (ATT), server rental cost (SRC) and average resource utilisation (ARU) and maximise one or more of these performance metrics when compared to three different baseline algorithms First-Fit (FF), Best-Fit (BF) and Worst-Fit (WF) as well as the All To Largest (ATL) algorithm that was developed in the first assignment. The job scheduler will be successful if we can have our algorithm perform better in our chosen metric compared to all of the three baseline algorithms and the ATL job scheduler.

Problem Definition:

The ds-server simulation has a number of performance metrics that can be used to evaluate how well a job scheduler has performed with regards to ATT, SRC and ARU. These three performance metrics can also be conflicting in their goals where trying to improve one metric can lead to the worsening performance of the other(s).

I have chosen to seek to improve the ARU of my job scheduling algorithm and have added on a smaller secondary goal where I try to not impact the other two performance metrics in too negative a manner. No more than a 20% performance loss in the other two performance metrics.

Resource utilisation can be described simply as how well all the resources (servers) have been utilised over the course of the simulation. If a server at any time during the simulation returns to an idle state with no jobs allocated, then the resource utilisation will start to fall below 100% for that server. For a single server it is calculated as follows:

***Server Resource Utilisation = (total time idle/total simulation time) \* 100***

The average resource utilisation is just the calculated average utilisation of all the servers.

***Average Resource Utilisation = Sum of all servers Resource Utilisation / Number of Servers***

I have chosen to try to maximise this particular performance metric as I believe that by properly utilising the available resources in a distributed system you end up with a system that is not wasting its resources as starting new servers costs money. A server that is sitting idle is still using power staying in this idle state, it should instead be used to perform jobs which it can start to work on immediately due to already being switched on as opposed to a server to switches back into an inactive or unavailable state due to being idle for too long.

Algorithm Description:

The algorithm I have designed has taken some elements from the FF and BF algorithms and made subtle changes to certain parts of their algorithms. It has also improved upon the known weakness in their current ds-client implementations where they were not scheduling to a server that still had cores available to run a job, but instead were scheduling to the smallest available server to run that job.

My algorithm is as follows:

1.Use the first server that can run the job in regard to the core needs of the job and  
2.Its server state is either “idle” or “active” or “booting” in that priority order

If no such server exists, then

1.Use the first server that can run the job with some boot-up time required

If no such server exists, then

1.Use the first server that is capable of running the job and  
2.Has a fitness test\* result equal to 0

If no such job exists

1.Use the first server that is capable of running the job and  
2.Has the highest fitness test (most cores available to run the job)

\*Fitness test is calculated as: Job core count – Server core count  
e.g a server with 2 cores and a job needing 2 cores gives a score of 0, also a server with 4 cores and a job needing 2 cores gives a score of -2 (lower value is higher fitness test)

Example Scheduling Scenario ds-config01—wk9.xml:  
  
Job Scheduling event order with algorithm  
J0 -> tiny server  
J1 -> small server  
J2 -> medium server  
tiny server running J0  
small server running J1  
J3 -> medium server  
medium server running J2  
medium server running J3  
J4 -> medium server  
medium server completed J2  
medium server running J4  
tiny server completed J0  
small server completed J1  
medium server completed J4

Discussion on example scenario:  
We have seen this particular configuration file run in our workshops with all three of the baseline algorithms used to schedule the jobs. A key point of difference is how the algorithm handles the scheduling of job 3 compared to the baseline algorithms, where the baseline algorithms in regard to FF and BF would schedule the job to the small server where the RUO schedules the job to medium server instead. This is due to the RUO using GETS Capable to create four array lists based on server states. By separating the servers into these different server states, we can find the server that still has space left to run the job to be scheduled even though the system considers it to be handling another job. Another important note is that the RUO scheduler when failing to find a server that has cores left to run a job will then schedule the job to either a server that exactly matches the jobs core needs or has the most cores free to run the job, which creates a more distributed approach to scheduling the jobs as it ensures not just the smallest servers will always run the jobs but also the larger servers will see more jobs to run and therefore help to keep the resource utilisation high.

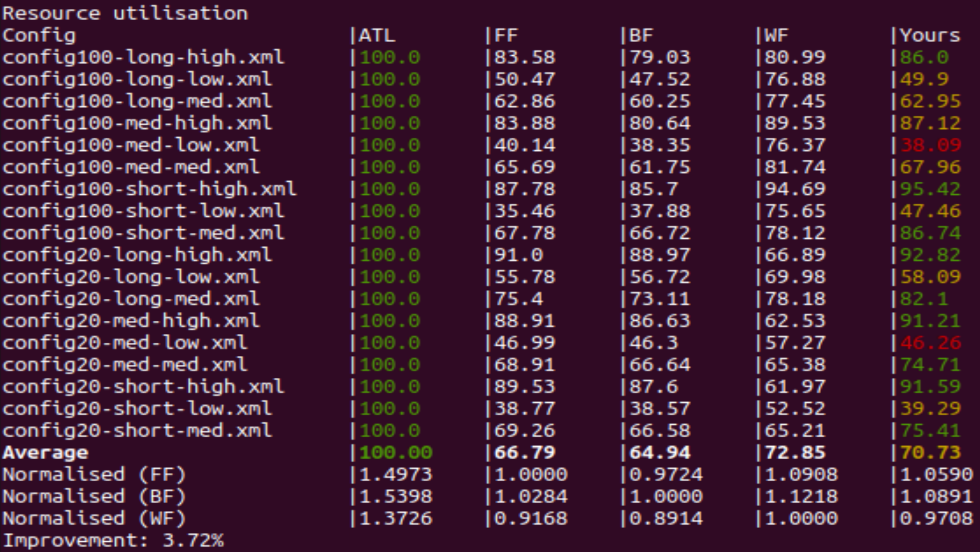
Implementation details:

The Resource Utilisation Optimizer (RUO) scheduler is contained across three classes, with the main class being the RUOptimizer class which contains the main driver code for the job scheduler. The other two classes are the Servers class and the Job class which are objects relating to servers and jobs respectively and are used in the main RUO class. Servers are stored in ArrayLists<Servers> and the current job to be scheduled is stored as a single instance as there is only one job to schedule at any one time. Servers are further subdivided into different ArrayLists depending on their system states and these lists are named accordingly, idleList, activeList, bootingList and inactiveList. The RUO scheduler will retrieve the servers from the ds-sim by reading in the information as a string delimited by the newline character. This string is converted into a string array which then is converted into a new instance of a server to be stored in the server ArrayList before being sorted into its appropriate state ArrayList. The main algorithm logic is contained with the method findBestServer() in the RUO class. This method contains other methods to send and receive messages from the ds-server, the main two messages this algorithm is based around are the GETS Avail and the GETS Capable messages. The control flow of the method is separated by an if statement that checks whether there are any servers to be returned from the GETS Avail message which we check if the number sent after the DATA portion of the reply is a number greater than 0. If there are servers available to run the job then the first part of the algorithm will run, finding, and scheduling the job to run on a server deemed to be best. If there are no servers from the Available message then the method will use the GETS Capable message instead and will find a server that either matches the jobs core requirements exactly or a server that has the most core capacity available at the time to run the job.

Evaluation:

The simulation was set up and run using the released test script “test\_results” and the 18 configuration files that this script needs to run properly contained within the “other” folder in the ds-configs folder. The testing environment was a folder containing the ds-sim, the RUO java files and the config folder “other”. The RUO scheduler is designed to read and send newlines in messages sent to and from the ds-server and so must be run in the newline mode “-n”.  
The test script was run using the following command prompt:  
./test\_results “java RUOptimizer” -o ru -n -c other

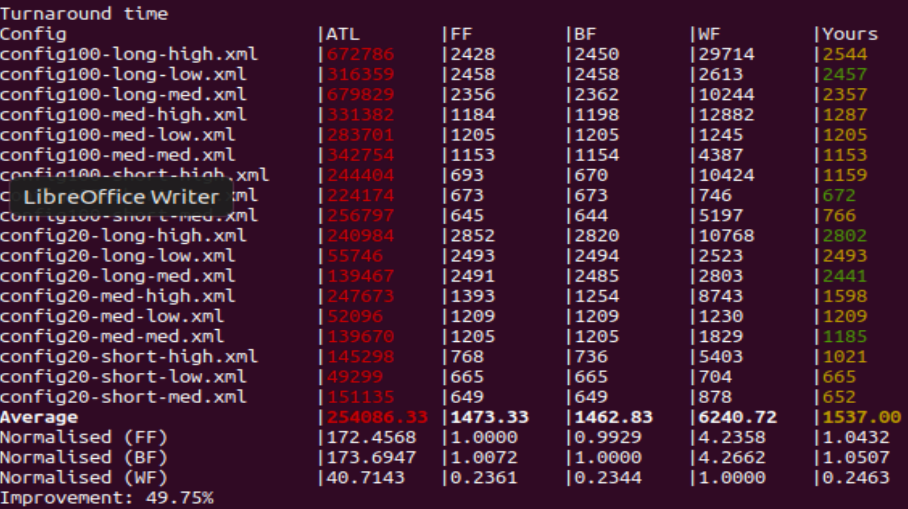
Testing Results:

  
**Figure 1**: Testing results for the Resource Utilisation Metric compared to FF, BF, WF and ATL with the RUO scheduler.

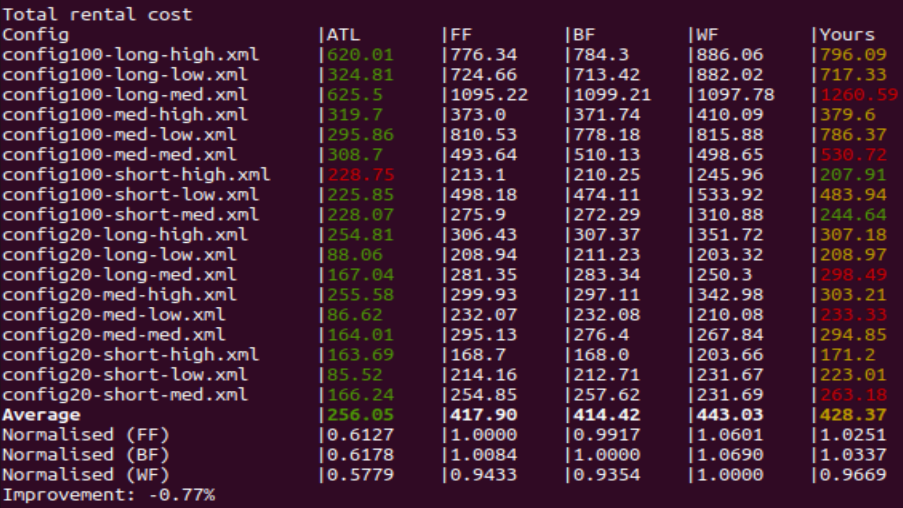
The results for the main objective of optimising the resource utilisation metric achieved largely successful results with a total of 9/18 configs achieving a green grade, as the RUO scheduler outperformed the other baseline algorithms in this metric. 7/18 configs were considered on par with the other baseline algorithms achieving a yellow grade meaning they were in the middle of the pack, not achieving the worst or the best performance using the particular config file. Finally there were 2/18 config files where the RUO was the worst performer achieving a red grade for the metric.  
  
Looking closer at the granularity of these results and counting the number of times an algorithm won a placing for each config file in terms of 1st through 4th (excluding ATL) we have the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | No. 1st places | No. 2nd places | No. 3rd places | No. 4th places |
| RUO | 9 | 6 | 1 | 2 |
| FF | 0 | 9 | 8 | 1 |
| BF | 0 | 0 | 7 | 11 |
| WF | 9 | 3 | 1 | 5 |

comparing this table with the Average utilisation shown in figure 1 helps put into context the performance of the RUO algorithm with the other baseline algorithms. In terms of the average the RUO achieved the 2nd highest utilisation average, only losing out to the WF algorithm. Both of these algorithms achieved the same number of 1st place performances across the 18 config files and the average shows us that when the WF algorithm performed the best, it did so on a greater margin than the RUO algorithm. However, looking at the distribution of the performances not just for first place but also for other placings overall we start to see a picture that shows that the RUO algorithm is more consistent in achieving a solid performance in this metric. Comparing again with the WF algorithm, the RUO only came 4th twice over the 18 test configs while the WF came 4th five times, similarly, the RUO achieved more 2nd places with 6 to WF’s three 2nd places. Comparing to the FF and BF algorithms the RUO outperformed both of these consistently with neither of these two achieving a first placing.

  
**Figure 2**: Testing results for the Turnaround time Metric compared to FF, BF, WF and ATL with the RUO scheduler.

Looking quickly at the results in figure 2 for Turnaround time we can a fairly good performance achieved by the RUO algorithm compared to the baseline. The RUO achieved five instances where it had the best turnaround time, but also another five instances where it achieved a joint 1st place result for turnaround time. Also of note was the RUO never performed the worst for any configuration file. These results hint at the similarities that are employed in the RUO algorithms logic where the first part of the algorithm closely matches the design of the FF and BF algorithms.

  
**Figure 3**: Testing results for the Total Rental Cost Metric compared to FF, BF, WF and ATL with the RUO scheduler.

The RUO algorithm performed the worst in this metric, achieving 1st place only on 2 occasions and last place on 5 occasions. This was a pretty competitive metric with the three baseline algorithms all having fairly close averages to each other but also having the ATL algorithm far outperform all other algorithms.  
  
Finally looking at the Improvement score in all three metrics the RUO achieved improvements in resource utilisation and turnaround time (3.72% and 49.75%) while achieving a less than 1% performance dip in rental cost.

Conclusion:

The RUO’s mission statement was to improve the resource utilisation metric compared to the three baseline algorithms with a secondary mission statement to be competitive with the other two performance metrics of Turnaround Time and Rental Cost. In the evaluation we came to the realisation that this algorithm achieved the goals that it set out to achieve as it had a higher performance consistency compared to WF which was the 2nd best performer in this performance metric. In terms of the secondary goal, it also achieved this by not having any metric record a negative improvement of more than 1%, showing that it can deliver on the primary performance metric but also offer a good trade off on the other two performance metrics, particularly for the turnaround time.

The RUO definitely has room to improve in terms of its primary goal of maximising resource utilisation however, and I feel this could be achieved through sacrifices made to the turnaround time. The RUO wants to utilise all servers in its execution and by doing this can maintain a good turnaround time while staying the better choice in terms of resource utilisation, but by trying to use all servers to maintain a good turnaround time it becomes harder to excel in resource utilisation as more servers require a very busy server environment to maintain high utilisation scores as the servers need to remain saturated with jobs at all times. For future iterations on this algorithm I would suggest changing the algorithm to seek to minimise the number of servers it uses, only utilising a server when a new job cannot be completed on any other server, and maintaining a list of these servers in use so that it can rotate the scheduling to each server. This would negatively impact turnaround time but it can stand to lose some of its current performance in this particular metric as it has a current improvement percentage of 49.75% and as the secondary goal was set to no loss greater than 20% then it could have further been optimised in this area.

References:  
  
Git hub link:  
https://github.com/JSFun9888/job-scheduler-ass2