CS4202 Computer Architecture - Process Scheduling for Heterogeneous Systems

140011146

November 20, 2017

1 Introduction

CPU scheduling is vital for the multitasking environment of modern operating systems. Processes waiting for I/O can be switched out to prevent wasted CPU cycles, but any process can be pre-empted for better user interactivity, efficiency and fairness [8]. As a result, it is important to understand the effects of scheduling on performance.

First, a round-robin scheduling approach for CPUs is looked into, discussing how the approach works and where the approach performs well or poorly.

Next, a genetic algorithm was implemented to explore the search space of possible schedules with the GEM5 simulator on two benchmarks. The schedules produced by the genetic algorithm are then compared with both the default approach of the simulator and random schedules to gain an understanding of the scheduling optimisation space.

2 Task 1 - Round-robin scheduling

The round-robin CPU scheduling approach is similar to a simple First-Come First-Served scheduling approach, but all processes bursts are limited by a time quantum. This ensures fairness, giving all processes a chance to run, but comes at costs such as high waiting time and high turnaround time. There is much research going into optimising a round-robin scheduler, usually by means of calculating an optimal time quantum dynamically [4] [5] [6] or introducing some form of priority [7].

In the simple round-robin approach, a circular queue of processes that are ready to be executed is maintained by the scheduler. New processes join at the end of the queue. The scheduler chooses and removes the first process in the queue to run [8]. Any running process will be pre-empted if they run longer than the time quantum. Processes that make block or finish before the time quantum are

also moved off the CPU [9]. A very long time quantum will effectively be the same as a FCFS approach, resulting in poor response and waiting time if there are processes with long bursts. On the other hand, a very short time quantum results in a large computational overhead of many context switches between the processes, wasting time and CPU cycles on context switching [1] rather than useful computation. Therefore, the length of the time quantum heavily impacts the performance of the round-robin approach and should be proportional to the time needed for context switches. For example, consider the following processes in the ready queue.

Process	**Burst time**
P_1	17
P_2	3
P_3	4

A time quantum of 3 would result in the following schedule and wait times:

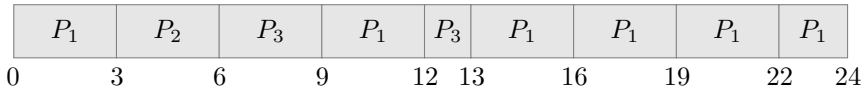

Figure 1: Schedule with a time quantum of 3

Wait times:

- P_1: 7 time units
- P_2: 3 time units
- P_3: 9 time units

Average wait time = $19/3 = 6.33$ time units.
Context switches = 8

However, a time quantum of 10 would result in a much longer average wait time.

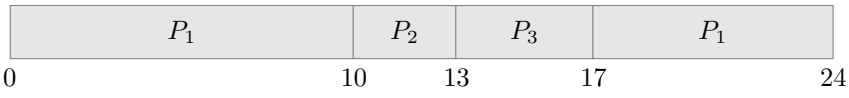

Figure 2: Schedule with a time quantum of 10

Wait times:

- P_1: 7 time units
- P_2: 10 time units

- P_3: 13 time units

Average wait time = $30/3 = 10$ time units.
Context switches = 3

It can be seen from the example how the choice of time quantum affects the waiting time, response time and number of context switches required. If the context switches took 1 time unit and a time quantum of 3 time units was used, the CPU would waste 25% of the time on switching between the processes. To make this worse, the small time quantum leads to an increased number of switches and a lot of computation will be wasted. To make good use of the processor, the time quantum should be a larger proportion compared to the time taken for context switching. Of course it also cannot be too big compared to the average burst time of processes or it will simply be a FCFS scheduler with little fairness. It is important to strike this balance of efficient CPU usage and fairness among ready processes. The long average waiting time is also an issue. Given n processes in the ready queue with a time quantum of t, each process only gets t time to run every nt time units and must wait at most $(n-1) \times t$ before each turn.

Round-robin is strong in cases where all processes bursts have similar lengths, or fairness is of greater importance to the system. For example in load balancing for parallel computation or time-sharing systems. However, problems with choosing an optimal time quantum and lack of ability to express priority, where some processes need a greater share of the processor than others means few systems would use a naïve round-robin algorithm.

There are many other more complex algorithms that are based on round-robin scheduling, keeping the concept of fairness but introducing other mechanics to achieve better efficiency. For example, schedulers which use multiple priority queues may have their high priority queues as round-robin with a small time quantum and lower priority queues with higher time quantum since those would be processes that need less interactivity [8]. If some form of feedback is added, then processes that exceed the high priority queue's quantum can be moved lower. This ensures fairness under round-robin while including priority to better match processes to time quanta. There are also many proposed algorithms that calculate the time quantum dynamically, which, depending on the suitability of the metric, helps relieves issues with un-optimal time quanta. The reason many of these algorithms continue to be based on round-robin despite its weaknesses is due how well it works as a fair way to run many different processes, which is important for modern processors, requiring high user interactivity and high number of concurrent processes being run.

3 Task 2 - Search for high performance schedules

3.1 Methodology

A genetic algorithm was chosen as a means to explore the search space of binary schedules. Because of the large optimisation space of schedules, a genetic algorithm with mutation adds elements of randomness to help explore more of the search space compared to a hill climbing algorithm which can get easily stuck in local maxima. Furthermore, hill climbing with random restart will only try to explore another unrelated area of the search space for a maximum without taking into account the previous local maxima. With a genetic algorithm, the next generation is created based on the previous generation, so good schedules can be maintained and modified through each generation. The fitness function used to evaluate a schedule is the total time taken of the execution of that schedule from the output of the simulation. The less time taken, the fitter that individual. As a schedule is a binary string, this maps perfectly to a genetic chromosome, where each gene is a single bit.

The genetic algorithm that was implemented follows three basic steps [2]:

1. **Selection** - Select parents to use for reproduction of the next generation. The fitter the chromosome, the more likely it will be chosen for reproduction. **Elitism** is also implemented, keeping exact clones of the most fit individuals so the best schedules are not lost.

2. **Crossover** - Mix the genes of the parents to create two new child schedules.

3. **Mutation** - Genes (bits) of all child schedules have a chance to be mutated to increase diversity and prevent the algorithm from being stuck in a local maximum.

3.1.1 Selection

A tournament selection method was used to select parents for reproduction. This method randomly selects a few chromosomes from the population, and chooses the most fit individual as the "winner" of the tournament. Tournament selection was used for its simplicity and flexibility. The tournament size can be decreased to allow weaker individuals more chance to be selected for more diversity or increased to give them less chance for faster convergence.

The same individuals are allowed to be selected multiple times for reproduction for better convergence. It is also done so the randomness in the crossover and mutation stages do not lead to a complete loss of strong genes as strong parents have more chance to be selected multiple times. Additionally, a small random number of elites are selected to be cloned in each generation as part of **Elitism**

to keep the fittest individuals. This is done to ensure the best schedules survive every generation as the effect of crossover and mutation on creating consistently better schedules is not known due to the size and complexity of the search space. The number of elites kept is randomised so the same elites are not kept every generation if the algorithm is at a local maximum.

3.1.2 Crossover

The simulator is deterministic, that is to say the same schedule will always take the same time to run. Additionally, if two schedules share the same initial n bits, they will run identically until they differ. This means that where the bits change in the binary stream matters. Given a binary schedule, if the first few bits are modified, the rest of the schedule may be interpreted completely differently even if no other bits are modified. If only the last few bits are modified, then the schedule is only different at those last few bits. The crossover step for genetic algorithms is quite suited for this behaviour. The children created by the two parents will each contain the first n bits from each parent and the rest from the other parent. If both parents were good schedules, this explores changing and mixing the second half of the schedule to potentially find an even better schedule without losing the good performance from the first half of the schedule.

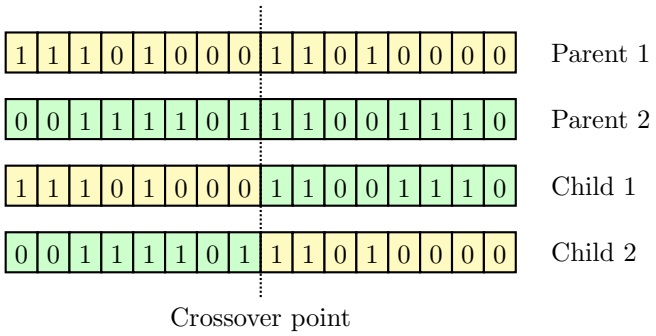

Figure 3: Example reproduction of children

There is only one single crossover point in the implementation as more complicated crossover techniques would take less advantage of the deterministic behaviour of the simulator and schedule. The crossover point is chosen randomly for more diversity as the same individual can be chosen as a parent multiple times.

3.1.3 Mutation

The mutation aspect of the algorithm will add diversity to the population, potentially changing a child schedule completely. This leads to a loss of a potentially good schedule, especially if the mutation occurred early and the parents were fit, but allows searching through more of the optimisation space

to try and find even better schedules.

Every gene of every child that is not an elite has a chance to be mutated. As such the mutation probability is rather low to prevent the algorithm devolving into a random search.

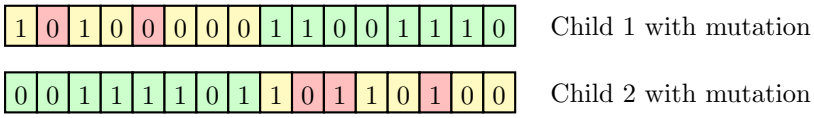

Figure 4: Children from figure 3 with mutation

It is important to note that although the combination of selection, crossover and mutation can lead to a systematic way of searching for better schedules while taking into account previous strong genes and keeping the best individuals, the complexity of the search space and lack of understanding on how changes to the bits directly affect the time taken make the performance of genetic algorithm unpredictable. It could well be that randomly creating schedules is an easier and faster method to find the fastest schedule.

3.2 Experimentation

The two benchmarks `sched-blackscholes_gen` and `sched-bodytrack_gen` are used in the experimentation. Initially, a naïve experiment was run using arbitrary parameters to see how well the genetic algorithm performed after many generations. Afterwards, the experiment was improved by adjusting the parameters to improve diversity. The results of both experiments are later compared with the default weighted RNG scheduler and a completely random approach.

To run the experiment, a python script was developed to connected to a number of lab machines equal to the size of the population. The input and output files to the simulator are saved on the local `/tmp` storage of each machine to prevent the files being overwritten when using the networked storage. Each machine ran the simulation on a given chromosome and when all the simulations are finished, the results are collected and the population evolved. Since many machines were used in each generation, there was a chance that some machines shut down while the simulation was still running. For these cases, instead of re-running the chromosome, it was given a fitness of 0. This is done due to time constraints so the experiments do not take more time as it is not uncommon for one or two machines to shut down for each generation. Although this is not ideal, as those chromosomes could have contained a really good schedule, it should not have greatly affected the algorithm as the whole population should be tending towards better schedules. This interestingly also mirrors natural selection in the wild, where even the best individuals may be killed due to unknown, random circumstances.

3.2.1 Experiment 1

In the first naïve experiment, the following parameters were chosen:

- **Population size**: 25
- **Chromosome length**: 100000
- **Mutation probability per gene**: $\frac{1}{(l/10)}$ where l is the length of the chromosome
- **Crossover point**: Random
- **Selection**: Tournament with 3 competitors

The population size was chosen as a size that was not too small or too large to start with. Fine tuning the population size for the specific task of finding a fast schedule will require more work, but starting with a medium sized population will give an indication as to whether this is too small or too large. A population that is too small will be stuck in a local maximum and unable to find better solutions, while a population that is too large would lead to a slow rate of convergence [3].

The length of the chromosome was chosen as a few times larger than the length of the schedule produced by the default approach. This is to ensure that the simulator does not run to the end of the schedule and fail. The issue with this is that if all the experiments only used up half of the generated schedule, any mutation or crossover with the unused half of the chromosome will have little to no effect. However, if the length of the chromosome is too short, the simulations will fail more often, making it more difficult to continue to evolve the population. Therefore the more conservative approach was taken to prevent chromosomes dying due to short length.

3.2.2 Experiment 2

From the first experiment, it was noticed that the algorithm starts to stagnate quickly. After a few generations, either because of a lack of mutation or population size, a large portion of each generation end up with the same schedule and fitness with little diversity. To fix this issue, the population and mutation probability is increased to increase diversity. The larger population and higher mutation rate explores more of the search space, but leads to slower convergence. To be able to keep strong individuals with the increased population and mutation, the number of individuals used in the tournament selection is also increased.

Below are the parameters of the second experiment:

- **Population size**: 40

- **Chromosome length**: 100000
- **Mutation probability per gene**: $\frac{1}{(l/100)}$
- **Crossover point**: Random
- **Selection**: Tournament with 5 competitors

3.3 Results

3.3.1 Benchmark properties

The two benchmarks chosen for this experiment are `blackscholes` and `bodytrack`. First, a quick overview on the performance of the default and random approach on the benchmarks. This will show how well these approaches work as well as if there are any distinct differences between the two benchmarks. Both the random and default approaches were run for 250 simulations.

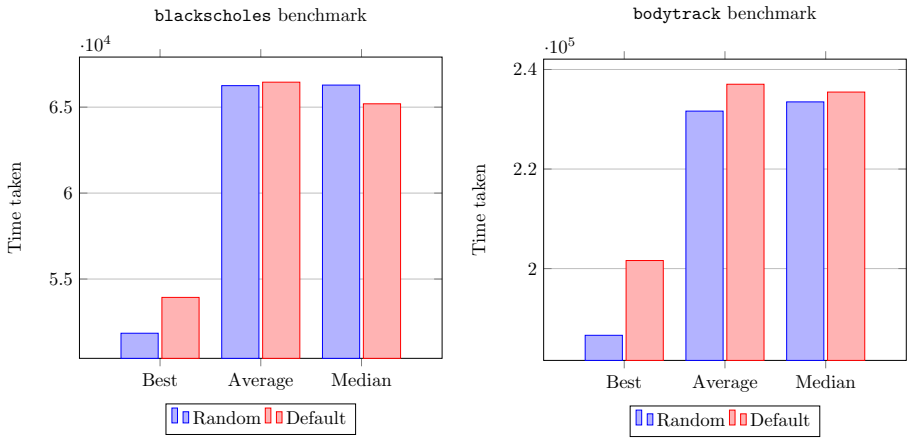

Figure 5: Best and average times taken by the random and default approach.

There are a few interesting points to note about the random and default approaches. First, the random approach is able to find a faster schedule than the default. This is interesting as one would expect the default approach to be able to come up with good and fast schedules. This shows that better schedules can be found through search that improve upon those made by the default scheduler. Following this, it can be seen that the average times for random and default are quite similar, though random more clearly outperforms for `bodytrack`. The median was also taken as an extra check to make sure the average was not skewed by anomalies, such as when the time taken is set to 0 for simulations which failed. This shows how the two approaches are similar in the kind of schedules being produced. An algorithm that can improve on a random approach should also improve against the default approach.

Another statistic of note is the variance, or standard deviation of the random and default approach. Here, the standard deviation is divided by the mean to get the variability relative to the mean and to be able to compare the two benchmarks.

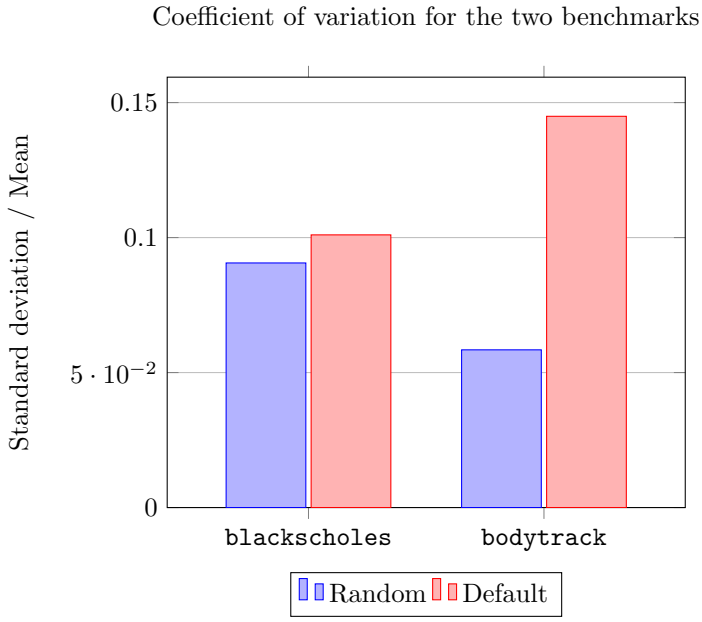

Figure 6: Coefficient of variation by the random and default approach, showing sensitivity of the benchmarks to random changes in schedules.

Again, it can be seen that the variation in schedules for the default approach is not necessarily better than that of random schedules. This may be due to some mistake or assumption by the scheduler which led to slower schedules than expected. A certain approach by the scheduler may be very effective for other benchmarks, but less useful for these benchmarks. Additionally, in the `bodytrack` benchmark, the low variation for random yet high variation for the default suggests the benchmark may be less sensitive to random changes in the schedule. This means a smaller search space, as changes in the schedule have a smaller effect on the time taken.

3.3.2 Genetic algorithm results

Now the results for both genetic algorithm experiments described earlier will be analysed and compared. In theory, the systematic approach of the algorithm should result in better schedules, or at least a better method to guarantee getting good schedules. Something to keep in mind is that since the second experiment had a larger population, it is able to run more simulations per generation, which may skew results to its favour.

First, the difference between the two experiments are compared, to see if the changes made to the parameters of the first experiment had an effect. There are two anomalies in the data for the first experiment on `blackscholes`, where the time taken significantly decreased, but subsequent generations did not pick up or re-use this good schedule. This may be a fault of the machines, where the best schedule was lost on the next generation. Still, a general trend can be seen for both benchmarks.

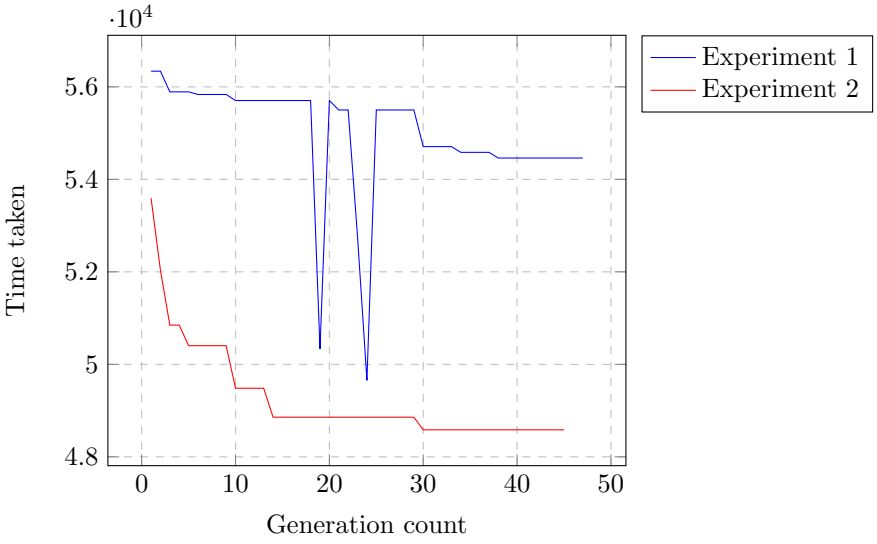

Figure 7: Best times taken by the genetic algorithm experiments on the `blackscholes` benchmark

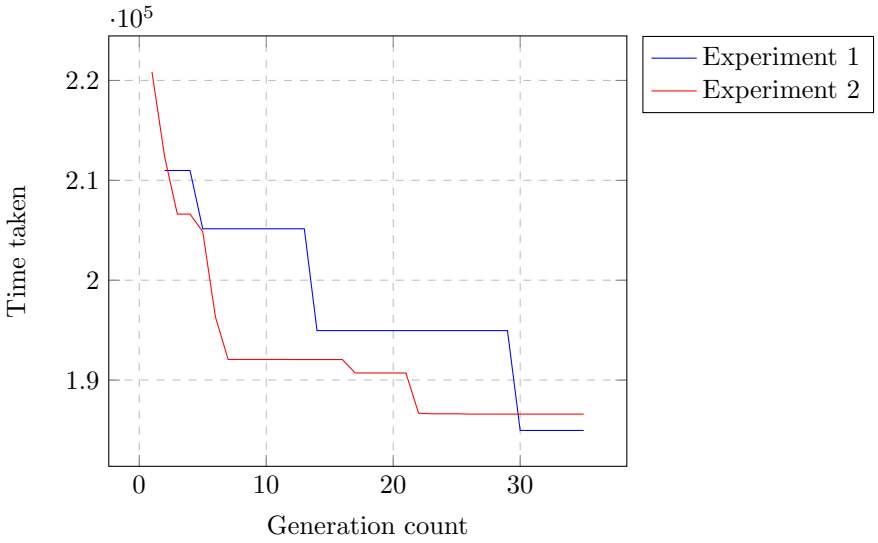

Figure 8: Best times taken by the genetic algorithm experiments on the `bodytrack` benchmark.

For `blackscholes`, the best schedules of the first experiment are clearly not as fast as those from the second experiment. The issue of diversity may have prevented the first experiment from getting out of a local maximum. By chance perhaps the first experiment could have found a better schedule, but it could not overcome being stuck. By contrast, the second set of parameters which increased the diversity of the populations seemed to be able to find faster schedules, but it is unclear whether this is due to the increased diversity, or if the mutation happened to be lucky. Both experiments had portions of generations where a better solution could not be found for a while, so it is not immediately obvious that increasing diversity is of great help to finding a better schedule.

The graphs for the `bodytrack` experiment support the claim that the parameters and diversity have little effect, as both experiments were able to continue finding better schedules. However, it could be the case that the first experiment got lucky. To be able to show the effect of the genetic algorithm parameters on the ability and consistency to find good schedules, more trials of the experiments would have had to be run to show these cases are not simply one-offs. If the diversity does have an affect, the expected results of more trials on experiment one would be finding some trials with good schedules and others where the local maximum is stuck somewhere far from a good time. The second experiment should then consistently end up with a good schedule every trial. Running more trials can also give further insight into how the properties of the benchmarks could have an affect, as perhaps both experiments perform similarly on one benchmark but differently on another. For example since `bodytrack` seems less sensitive to random changes, increased diversity from experiment 2 may have

less effect and therefore perform similarly to experiment 1.

The two experiments can also be compared by how their average time taken changes for each generation. This will show how the entire population of the genetic algorithm tends towards good schedules with each subsequent population.

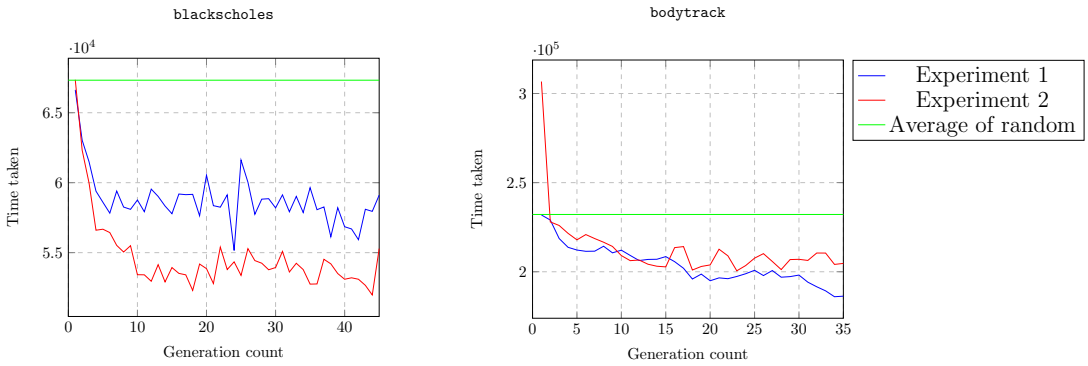

Figure 9: Mean time taken in each generation compared to the average time taken from all random runs.

Figure 9 shows the two experiments getting better schedules on average for each generation. The line of average time taken by random schedules serves as a comparison. It is a straight line to represent the average random schedule. This suggests that the logic of the genetic algorithms can apply to this search problem. For each subsequent generation, the performance of the previous generations has affected the schedules created, resulting in getting better average schedules each generation compared to a random approach. This shows the genetic algorithms will guarantee getting better schedules after each generation as the entire population tends towards better schedules over time. The search space therefore is being traversed in a methodical way as described earlier in section 3.1. `blackscholes` has a larger search space compared to `bodytrack` which may explain the trend being not very clear. The mixture of selection, crossover and mutation seem to work in being able to keep strong schedules while mutating new children to try and improve the schedules. For further work, the mutation rate of each gene can be looked into. It would be expected that mutating more of the later genes affects searching through the current local maximum while mutating early genes would find new schedules. The overall effect should also be weaker for `bodytrack` as it is less responsive to changes in the schedule.

Although this shows the genetic algorithm working for searching through the schedule space and will eventually lead to good schedules, it is not clear if this is a fast method of searching compared to the purely random approach. What is important is how fast the best schedule is for each generation, not the

average time taken for the entire population. Compared to a purely random approach, the genetic algorithm may take more simulations just to guarantee a good schedule that the random approach can find by chance.

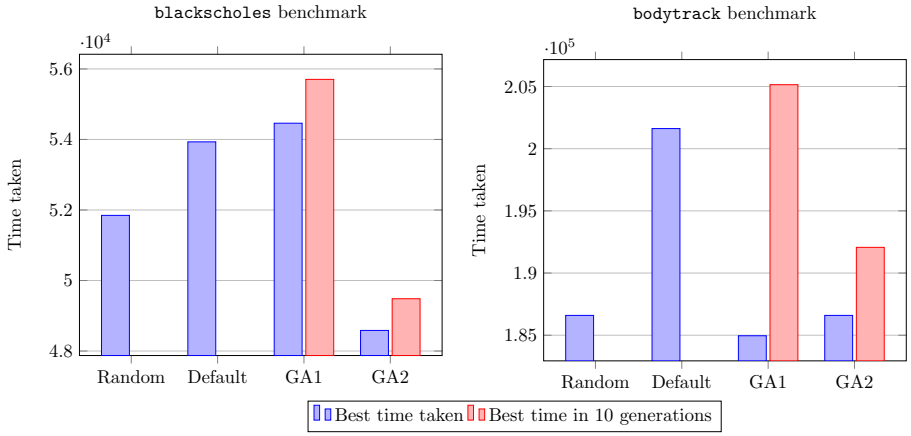

Figure 10: Best times taken by all approaches.

The best time taken for every approach is compared in figure 10. The best time taken for the first genetic algorithm experiment does not take into account the outliers that could be seen in figure 7 to be able to show the general trend that the experiment was exhibiting.

When comparing all the approaches, it is not clear if the genetic algorithms perform better at finding good schedules compared to the random approach. The red bars represent the best schedule found in only 10 generations, which is important for a proper comparison, as the genetic algorithms were allowed to run for more simulations compared to the default and random schedules. We see the genetic algorithms sometimes find better schedules, but not all the time and especially not with a comparable number of simulations. Given more time to run more experiments, the parameters for the genetic algorithm could be improved and analysed to find an optimal version where it converges to good schedules faster while maintaining diversity. The size of the search space also seems to affect the effectiveness of the genetic algorithm. For example in `blackscholes`, the second experiment is able to find a very good schedule compared to random and experiment 1 whereas random outperforms both genetic algorithm experiments for `bodytrack`. It was suggested earlier that `bodytrack` has a smaller search space due to how the variation of random schedules is small. This perhaps shows the genetic algorithm performing better for a larger search space, where it can methodically find peaks. In a smaller search space, the random is likely to be able to find a good schedule faster and before the genetic algorithms can converge to some maximum. This can be shown more clearly if a larger variety of benchmarks and more trials for every benchmark were run.

The difference between the genetic algorithm and random approach represents a trade-off in searching for fast schedules as well as how big of a space needs to be searched. If it is required under some constraint to find good schedule, or the optimisation space is quite large, it may be better to use an optimised genetic algorithm to guarantee a good schedule. However, if there are less constraints and a smaller search space, running random schedules can have a good chance to find something better in less simulations, but it is not guaranteed.

3.4 Conclusion

In conclusion, the genetic algorithm implemented was able to find better schedules as the number of generations increase and is able to improve upon the default scheduler. It works on crossing over good parent schedules with mutation to both avoid local maximum and use the parent's genes to try and get something better. However, when compared to randomly searching for schedules, the genetic algorithm was not very efficient with the number of simulations needed.

A random approach is able to find a comparable schedule in the same number of simulations as the genetic algorithm. This also depends on the size of the search space as a random search on a large space will be less effective. The advantage of the genetic algorithm is that the whole population will tend to a better schedule, so it is guaranteed to find something better than the default and it is able to more systematically search through larger spaces. A good schedule is not guaranteed for the random approach, but the smaller the search space, the more likely a good random schedule can be found.

References

[1] *Context Switch Definition*. May 2006. URL: `http://www.linfo.org/context_switch.html`.

[2] *Introduction to Genetic Algorithms*. 1996. URL: `https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html`.

[3] V.K. Koumousis and C.P. Katsaras. "A saw-tooth genetic algorithm combining the effects of variable population size and reinitialization to enhance performance". In: *IEEE Transactions on Evolutionary Computation* Volume 10, Issue 1 (Feb. 2006).

[4] Rami Matarneh. "Self-Adjustment Time Quantum in Round Robin Algorithm Depending on Burst Time of the Now Running Processes". In: 6 (Oct. 2009).

[5] Manish Kumar Mishra and Dr. Faizur Rashid. "An Improved Round Robin CPU Scheduling Algorithm with Varying Time Quantum". In: *International Journal of Computer Science and Engineering* Vol 4, No. 4 (2014). URL: `http://airccse.org/journal/ijcsea/papers/4414ijcsea01.pdf`.

[6] Abbas Noon, Ali Kalakech, and Seifedine Kadry. "A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average". In: *International Journal of Computer Science and Engineering* Vol 8, Issue 3, No. 1 (2011). URL: https://arxiv.org/pdf/1111.5348.pdf.

[7] Ishwari Singh Rajput and Deepa Gupta. "A Priority based Round Robin CPU Scheduling Algorithm for Real Time Systems". In: *International Journal of Innovations in Engineering and Technology* Vol 1, Issue 3 (2012). URL: http://ijiet.com/wp-content/uploads/2012/11/1.pdf.

[8] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts Essentials*. 2nd. Wiley Publishing, 2013. ISBN: 9781118804926.

[9] Ajit Singh, Priyanka Goyal, and Sahil Batra. "An Optimized Round Robin Scheduling Algorithm for CPU Scheduling". In: *International Journal of Computer Science and Engineering* Vol. 02, No. 07, (2010). URL: http://www.enggjournals.com/ijcse/doc/IJCSE10-02-07-83.pdf.