CPU governance and scheduling using machine learning implemented on a Linux kernel

Jonathan Gao

Cornell University
October 28, 2020

Project Introduction

As an exploratory dive into the Linux kernel, I will be modifying the Linux CPU scheduler and governor using learned decisions from a machine learning model. This project will be implemented specifically on a Raspberry Pi 4 Model B, a small single board computer running a quad-core ARM SoC. Thus, this modification will be specifically done to the Raspberry Pi OS linux distro.

This project was inspired by the constant tug-of-war between performance and power consumption. Schedulers and CPU frequency scaling have largely remained separate. Schedulers "maximize throughput, minimize wait time, minimize latency, and maximize fairness" [1]. CPU governors instead "scale the CPU frequency up or down in order to save power" [2]. These two operations are innately tied together, as frequency scaling is dependent on CPU load, and CPU load is determined by the scheduler. So what if these two operations become intertwined?

We can consider several cases where this may be advantageous.

1. To optimize for power, the CPU governor should only scale frequency up when we significantly load our CPU. Processes that are dominated by idle time, e.g. by I/O operations, can squander away high power CPU time. Optimally, we should increase CPU frequency only when our CPU is expected to be fully loaded - something we can accomplish through scheduling.

2. Given how a majority of current CPU frequency scaling works (by aggressive shifts up and gradual shifts down), we should consider how much power we waste by constantly scaling up and down repeatedly. If we could estimate our CPU load times, could we prevent unnecessary scaling? e.g. intermittent repeated short processes just above the scaling threshold can lead to a CPU constantly scaling up to max frequency despite minimal load.

3. CPU frequency can affect process wait time, throughput, and latency - factors that contribute to scheduling decisions. How do we decide on an optimal frequency then? On current governors, we see that CPU load directly determine frequency scaling. But, what if our CPU does not necessarily need max frequency, even at full load? e.g. a long latency-insensitive process or a process that simply runs in the background without terminating.

While we can prove an optimal algorithm that can use all of our inputs to decide CPU frequency and CPU scheduling, I will not be pursuing that route. Rather, as an experiment I will try to use a machine learning algorithm to do the balancing. I will try to pose this problem of scheduling and frequency scaling as a classification problem, where given certain inputs we will determine the next process that should be scheduled and the frequency at which we will run our CPU.

Semester goals

For this current semester, I will be primarily focused on research. As I have no experience working with the Linux source code, initial steps include unraveling the Linux kernel and learning to make modifications appropriately.

A few goals are outlined for this semester:

1. Compile a Linux kernel and boot

   Using the Lab 4 guide from ECE 5725, I downloaded the Raspberry Pi OS 5.4.72 source code from the github repo and patched it using the PREEMPT RT 5.4.70 patch for real-time kernels. Then, I used a cross-compile toolchain to build the kernel on my Linux machine [3].

2. Understand the architecture of the Linux kernel, more specifically how the cpufreq and sched modules of the kernel are implemented

3. Derive methods to inject a machine learning algorithm somewhere into the mix. More research needs to be done on how this can be exactly implemented. Currently thinking of training the model offline with data, but that will then require obtaining a dataset somehow.

4. As a continuation of above, I will need to look into learning algorithms. I am currently thinking of a standard SVM classifier, but that requires test data under a supervised learning model. Unsupervised learning may be a better approach, but I am not as familiar in this domain. Training might be much easier on a test framework that simulates the kernel as it will also be easier to train over epochs.

5. Again as above, a test framework will be optimal. Not only will it help with model training, but also with testing and data collection.

Final deliverables

The deliverables I plan to have by the end of this project are:

- A modified Linux kernel with a ML-driven scheduler and CPU governor.
- or, a patch to the Linux kernel that makes these changes.
- An analysis on performance, pointing out optimal workloads and sub-optimal workloads with comparisons to existing methods.

Testing

Testing for this project can take a multitude of paths.

On the Raspberry Pi itself

The Raspberry Pi allows configuration of multiple kernel images through the boot partition. Using this allows me to easily switch between kernels with both kernel images on the SD card. Using a bash script, I can create a workload to benchmark and test my algorithm.

Using a virtual test framework

As aforementioned, I will also look into creating a test framework that can emulate the kernel. This will be optimal, as then I should be able to do testing on my computer instead of the Raspberry Pi. This will also constrain the variability to only within my algorithm.

Data points

When testing, I will be focused on the following data points

- Power consumption (averaged)
- Wait time, response time, and throughput of processes
- Histograms of time spent in each frequency quanta
- Time spent in the kernel vs time spend in user processes

Tentative design

This section will discuss the current implementation details I have, prior to any deep dives.

Machine learning approaches

For the machine learning algorithm, I want to optimize for speed in calculations. This naturally warrants simpler models such as support vector machines (SVM) as opposed to complex convolutional networks.

1. An SVM classifier to predict the level a process belongs to in a multilevel feedback queue scheduler, rather than the continous promotion and demotion. This may be similar to guessing how much CPU quanta each process requires, which could be another technique. This might be a good approach as most operating systems today use multilevel feedback queues as their scheduler. The mentioned design may be similar to the one discussed in this paper [4]. This method may require a separate component for CPU frequency, where another SVM classifier can predict the frequency level based on statistics like queue capacity in each level.
   This approach may also prevent starvation in a multilevel feedback queue, or be able to avoid the overhead of feedback queues by better predictions for a regular multilevel queue as noted here [5]. Furthermore, the model can be used to determine whether or not to pre-empt running processes.

2. The above design can be extended using a generative adversarial network, or GAN for short, to avoid the need of a supervised test dataset. However, much more research must be done to determine its feasibility. My guess is that I will end up constrained by hardware limitations unless I can develop a test framework.

3. A genetic algorithm could potentially be used for unsupervised learning as well, but there is no guarantee on its performance and I also do not have much experience in it. Could be worth a look though.

4. Lastly - into the realm of unsupervised learning models, a denoising autoencoder network may also work. They are good at dimensionality reduction, which might be helpful in determining how each input should be weighed in making a decision in scheduling.

5. Another approach may be to use machine learning algorithms to decide between several different scheduling algorithms. There are many algorithms in place that each have their advantages. Instead, we may be able to exploit those advantages simultaneously.

My best bet is the first approach mentioned. Multilevel feedback queues are plentiful and abundant, where an SVM would be essentially an added boost of performance (hopefully). However, with more research the other methods could prove fruitful.

Currently, I believe that incorporating CPU frequency scaling into scheduling decisions may return the largest profits in power and performance. Perhaps some classifier can be used to predict workload types, on which we can scale our frequency accordingly?

All of these techniques will likely require copious amounts of data, which may be a challenge as discussed below.

Test apparatus

Using a Raspberry Pi makes it much easier to test kernels as the kernel is configurable from the boot partition. The boot partition allows selection of kernel images, of which multiple can be installed at a time. This would allow me to compare workloads quickly without the trouble of manually configuring the OS boot loader each time.

Using the Raspberry Pi also makes it really easy to test power consumption, where there is no battery onboard to buffer power consumption and I can use a battery to emulate power constraints.

It also allows cross-compilation of the the kernel using my laptop or even ecelinux, which will definitely speed up development.

Additional thoughts

Multicore scheduling vs single core scheduling

Multicore scheduling might be much more complicated than the single core scheduling that I have learned. This will definitely require a deeper dive as this may end up increasing the complexity of this project severely. Scheduling across multiple cores will have varying complexity depending on architecture. For example, newer Intel architectures can support NUMA computing, which leads to concerns such as cache affinity as scheduling one process onto different cores has different costs due to varying cache performance [6]. However, in this project I will focus on a multicore processor model without these concerns. The Raspberry Pi 4 uses the ARM Cortex A72 core design in the BCM2711 chip, which offers a quad core processor.

Performance impact of scheduling using a ML algorithm

As most scheduling algorithms are implemented to be as fast as possible, the overhead of running an machine learning algorithm might be orders of magnitude larger. Thus, this approach may not be worth pursuing for practical purposes, but rather for an exploratory dive into the possibility.

Feasibility of Linux Kernel modification

Having no experience in modifying the Linux kernel, I may end up finding that the Linux kernel is too complex to fully understand within the time frame proposed for this project. While I am limiting the scope to just the scheduler and governor, it may still end up being too complex. Thus, I will reduce the scope of this project as necessary and update the project goals accordingly.

An example of such a change may be to only implement a ML-driven governor, as a governor should be simpler.

Data

Machine learning is a beast constantly thirsting for more data. Coming with this data may be a challenge, especially for supervised learning as there is not really a correct level for a multilevel queue. Thus, either I have to generate these through approximation, or develop another metric that I can optimize for.

Expected Results

Expectations are not high for any noticable improvement in scheduling performance or power consumption. As mentioned previously, the overheads of a machine learning algorithm may end up overwhelming any performance gains. Rather, this project is an exploration into the potential of machine learning algorithms in scheduling and the potential of co-dependent scheduling-governance algorithms.

References

[1] "Scheduling (computing)," *Wikipedia*, Oct. 2020.

[2] "CPU frequency scaling - ArchWiki." https://wiki.archlinux.org/index.php/CPU_frequency_scaling.

[3] "Kernel building - Raspberry Pi Documentation." https://www.raspberrypi.org/documentation/linux/k

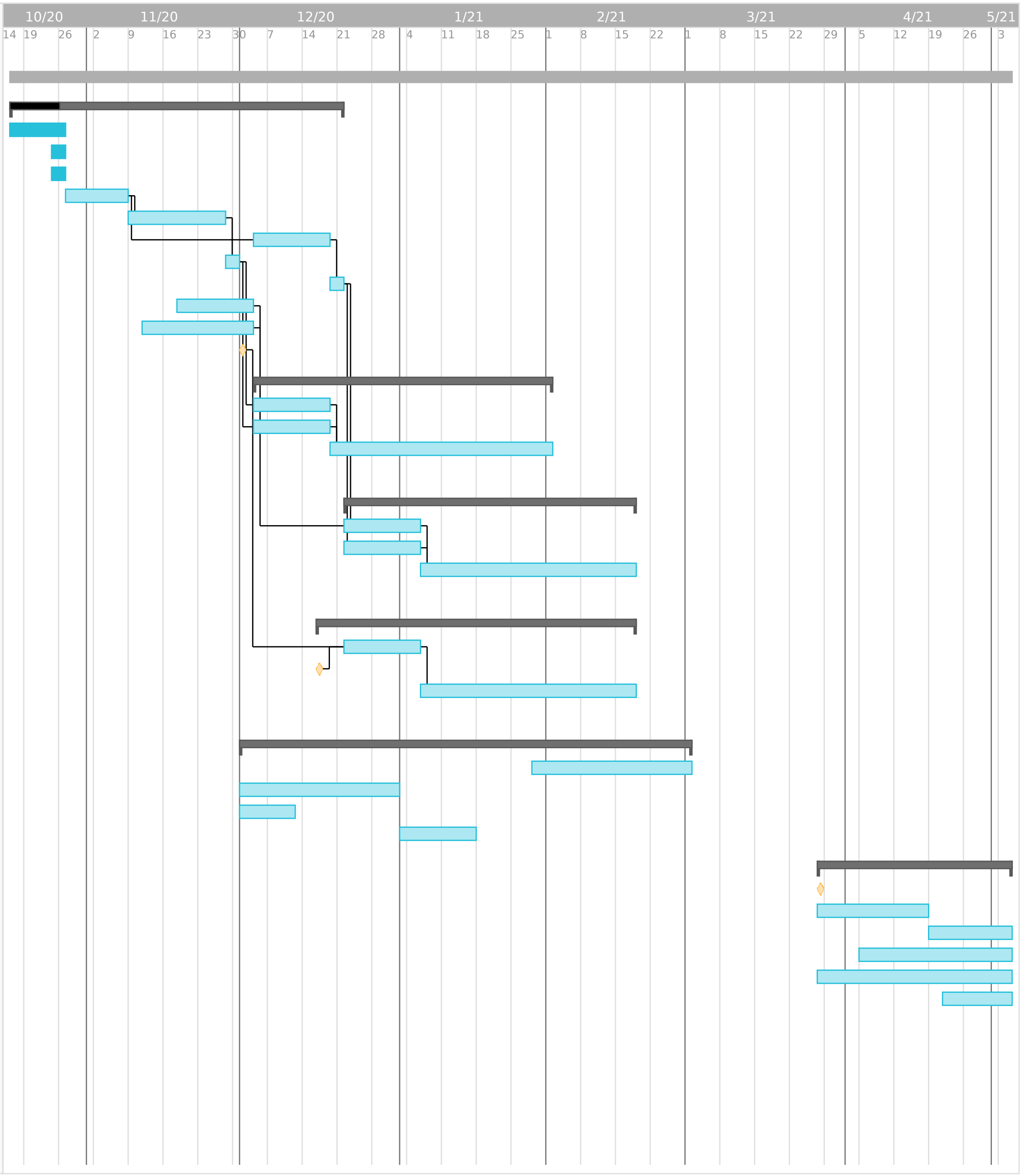
[4] S. Satyanarayana, P. Sravan Kumar, and G. Sridevi, "Improved Process Scheduling in Real-Time Operating Systems Using Support Vector Machines," in *Proceedings of 2nd International Conference on Micro-Electronics, Electromagnetics and Telecommunications* (S. C. Satapathy, V. Bhateja, P. S. R. Chowdary, V. S. Chakravarthy, and J. Anguera, eds.), vol. 434, pp. 603–611, Singapore: Springer Singapore, 2018.

[5] "Difference between Multilevel Queue (MLQ) and Multi Level Feedback Queue (MLFQ) CPU scheduling algorithms." https://www.geeksforgeeks.org/difference-between-multilevel-queue-mlq-and-multi-level-feedback-queue-mlfq-cpu-scheduling-algorithms/, July 2020.

[6] "Cpu-sched-multi.pdf." http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-multi.pdf.

MEng Linux Kernel Modifications

	start	end
Research	**10/15/20**	**12/21/20**
Compile Raspberry Pi OS kernel	10/15	10/26
Make a change to the kernel	10/23	10/26
Cross compile Raspberry Pi OS kernel on Linux	10/23	10/26
Understand how the kernel is built	10/27	11/06
Understand how the Linux Kernel scheduler is i...	11/09	11/26
Understand how the Linux Kernel governor is im...	12/03	12/17
Draw out diagram of how scheduler functions	11/27	11/30
Draw out diagram of how governor functions	12/18	12/21
Research other potential scheduler implementat...	11/18	12/02
Research existing machine learning libraries for ...	11/11	12/02
Decide on language for machine learning model	12/01	12/01
Implementation of Scheduler	**12/03/20**	**02/01/21**
Design overview of scheduler implementation	12/03	12/17
Modularize scheduler to incorporate external in...	12/03	12/17
Scheduler Implementation Prototype 1	12/18	02/01
Future tasks dependent on design	-	-
Implementation of Governor	**12/22/20**	**02/17/21**
Design overview of governor implementation	12/22	01/05
Modularize governor to incorporate external inp...	12/22	01/05
Governor Implementation Prototype 1	01/06	02/17
Future tasks dependent on design	-	-
Implementation of Machine Learning Model	**12/16/20**	**02/17/21**
Design overview of machine learning model	12/22	01/05
Decide on unsupervised or supervised learning ...	12/16	12/16
Machine Learning Model Implementation Protot...	01/06	02/17
Future tasks dependent on design	-	-
Data Collection	**12/01/20**	**03/01/21**
Power consumption measurement prototype	01/28	03/01
Figure out how to measure wait time, response t...	12/01	12/31
Research kernel benchmarking programs	12/01	12/10
Deep dive into potential test framework that wr...	01/01	01/15
Report	**03/26/21**	**05/04/21**
Begin drafting report	03/26	03/26
Design overview	03/26	04/16
Discussion	04/19	05/04
Results	04/05	05/04
Methodology	03/26	05/04
Introduction	04/21	05/04


10/20
11/20
12/20
1/21
2/21
3/21
4/21
5/21
14 19 26 2 9 16 23 30 7 14 21 28 4 11 18 25 1 8 15 22 1 8 15 22 29 5 12 19 26 3