Page Scheduling using Temporal Difference Learning

1st Hayden Coffey
University of Tennessee, Knoxville
Knoxville, US
hcoffey1@vols.utk.edu

***Abstract*—This paper examines the previous work of the Kleio page scheduling manager and attempts a similar simulated implementation using temporal difference learning.**

***Index Terms*—Machine Learning, Reinforcement Learning, Page Schedulers, Temporal Difference**

I. Introduction

Heterogeneous memory systems, computer systems with more than one type of memory device, are becoming more common with the arrival of NVMe technologies such as Optane. These systems present new challenges to developers if they are to effectively take advantage of these devices. For instance, comparing DRAM to NVMe, DRAM is generally less energy efficient and more expensive per byte of storage, but has faster access times than NVMe. It would then benefit a program's performance to have frequently accessed pages backed by the faster memory type, whereas colder data could reside in the slower, but larger, device.

Page schedulers are one form of tool that can be used to efficiently organize pages across varying memory devices in order to maximize program performance. Below are two types of page schedulers relevant to this project:

- History Scheduler: Tracks page accesses across a scheduling epoch and assumes that pages will maintain their values while deciding page placement i.e. if a page was hot last epoch, assume it will be hot this epoch and place in fast memory.
- Oracle Scheduler: Ideal page scheduler, knows future access counts for pages and can choose optimal page placement.

This project aimed to simulate an additional form of scheduler that utilizes reinforcement learning to choose page placement. This idea is derived from the Kleio paper by Doudali et al. which explored using machine intelligence in page schedulers, but used RNNs instead of reinforcement learning methods [2]. As a result, approaching this problem with reinforcement learning is the novel aspect of the project.

II. Previous Work

A. Kleio

The primary source for this work is the Kleio paper. As stated before, Doudali et al. explored machine intelligence in page schedulers in their publication, but did not use reinforcement learning due to the exponential growth of the problem space. As will be explained later, this project was able to get the problem space to a more manageable size, but did not achieve the same level of improvement to performance as Kleio[2].

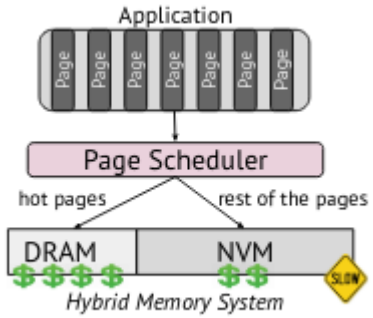

Fig. 1. Illustration of page scheduler. Graphic taken from Doudali et al.[2]

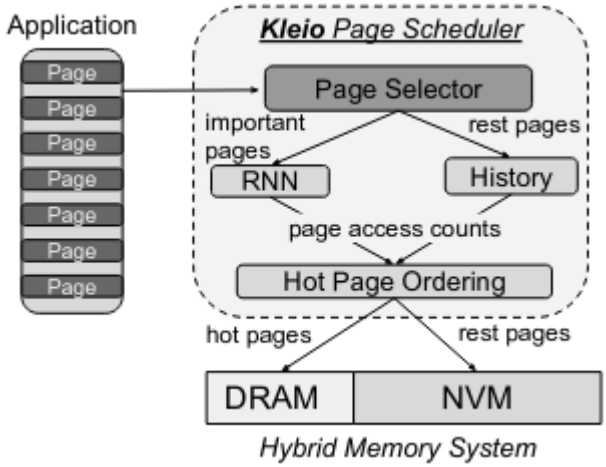

Fig. 2. Kleio's model design. Taken from Doudali et al. [2]

This paper followed a similar methodology to that of the Kleio paper as illustrated in Figure 2. Kleio's version reads in the applications pages, chooses the pages that impact performance the most to be used with distinct RNNs, and uses the history scheduler on the remaining pages. The authors found that only a small portion of pages need to be decided using

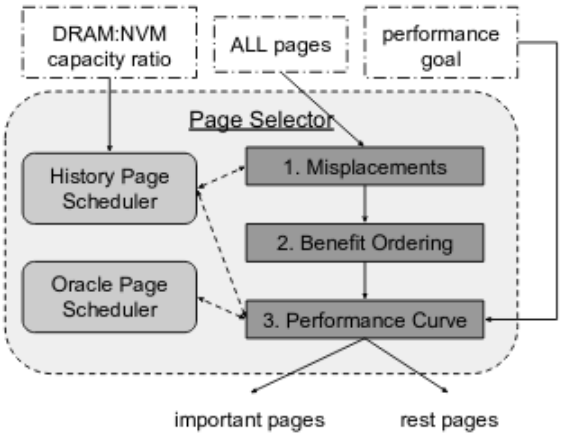

Fig. 3. Kleio's page selector system. Taken from Doudali et al. [2]

machine intelligence in order to gain performance benefits. The process for selecting pages in Kleio's implementation is illustrated in Figure 3. The misplacements for each page are tracked, with a misplacement being when the oracle and history page schedulers differ in decisions. These are then multiplied with the total accesses to that page to calculate a benefit metric for using machine intelligence for that page.

$$\text{benefit} = \text{misplacements} * \text{total accesses} \quad (1)$$

The pages are then sorted by their benefit values and those pages with the highest metrics are chosen for machine intelligence based selection.

B. State Space Reduction

In an attempt to further reduce the problem space growth, state space reduction techniques were examined. The actual RL model will be described later, but it appears that the problem does not reduce easily. One paper on MAXQ Value Function Decomposition by Dietterich was examined, but did not easily apply to the problem space as there is not a discernible hierarchy to actions within problem space of the page scheduler [1]. Another paper by Sadamoto et al. used dimensionality reduction techniques to project the state space onto a lower dimensional space, though this was not explored further due to time constraints. Additionally, this work was performed within the context of large networking systems and may not be as effective on the more simple page scheduling problem[3].

III. Domain Problem

For operating systems like Linux, memory is divided into units called pages for program use. Additionally, virtual memory address ranges are used in order to give each application the appearance of its own memory address space. These virtual pages still require physical backing of memory on the hardware and are assigned physical page frames. Memory accesses are then translated from their virtual address and page to the physical address and page frame assigned via the memory management unit of the operating system.

The page scheduler is responsible for examining memory access patterns and determining which virtual pages should be given which page frames, as shown in Figure 1. The goal for this project was to use reinforcement learning in the page scheduler to evaluate these access patterns and determine whether a page should be placed in fast or slow memory.

This project loosely follows the methodology used by Kleio and shown in Figures 2 and 3. A page selector was written that uses the benefit metric described in equation 1. This is done by performing an offline profiling run using the history page scheduler and calculating the benefit values for each virtual page and saving the results to a file to be used later by the RL based scheduler. The RL scheduler then chooses the pages that will provide the most benefit if properly trained and creates a separate RL "agent" for each chosen page that has its own Q-Value matrix and state. The agent chooses whether to place the page it is assigned to in either fast or slow memory and receives a reward for its decision on the next scheduling epoch.

A. Framing as MDP

1) State Space: The parameters defining the state for the problem are as follows:

- Page hit count from past scheduling epoch (and optionally scheduling epoch before that).
- Current memory device (slow or fast).

In order to reduce the state space, the hit counts were aggregated into denominations of 100 up to a max value of 10000, for a total of 100 possible values. With two memory device types, the state space ranges from $100 * 2 = 200$ (depth of 1 for hit count history) to $100 * 100 * 2 = 20000$ (depth of 2 for hit count history).

2) Action Space: There are two possible actions for the scheduler to take with a given page. It can either place the page in high or low tier memory, high being faster than low.

With these two actions, the Q-Value matrix expands to 400 to 40000 values for state action pairs. It should be noted that splitting the agent into multiple agents and by only choosing which device to place the page in, and not its ordering in the device, the state action space should only scale linearly with the number of selected pages to train on. This is worth mentioning because otherwise, as the Kleio paper pointed out when rejecting RL, the action space will grow at an exponential rate of 2^n for n pages. This method does, however, assume that there is enough space on the chosen memory device to fit the page.

3) Reward Model: With the goal being to improve performance by placing hot data on the fast memory device, the negative magnitude of the time spent accessing memory in the next scheduling epoch is used as the reward. Memory access to the high tier add 1 to the delay, while low tier memory accesses add 10. The agent will then aim to minimize this value by placing hot pages on the high tier.

IV. Reinforcement Learning Methods

One step temporal difference learning was chosen for use with this project. The reasoning behind this is that each episode of the problem encompasses an entire program execution. These episodes could run for quite some time, and the memory cost of saving an entire episode's worth of state/action/reward would become impractical as a lightweight system is desired for this problem. Instead, TD allows for the model to learn from its actions mid-episode while keeping its memory footprint minimal. Additionally, in a real application, the page scheduler should be able to adapt to the current workload as access patterns will differ between runs, which TD possesses the ability to do so.

V. Code Design

The code for this project is available in both Python3 and C. The file **pagesim.py** contains the Python3 implementation, while **pagesim.c** is the C version. The code can be divided into three main categories:

A. Page Simulator

Adapted from UT's Fall 2020 Computer Architecture course solution code. This is primarily implemented with the *proc_page_lookup* method and *Page_table* data structure which are used for simulating accesses to a page table and managing the pages using an LRU eviction scheme.

B. Page Schedulers

The *history_scheduler*, *oracle_scheduler* and *rl_scheduler* methods provide implementations of the history, oracle, and TD based page schedulers. The *schedule_epoch* method is called once every scheduling epoch and chooses which page scheduler to use based on the user's configuration. The *page_table* global array provides lookup information about virtual pages, while the *phys_pages* global array provides details about the physical pages.

Additionally, the *page_selector* method implemented the page selector used to choose the pages that would provide the most benefit if trained with machine intelligence.

C. TD Implementation

One step temporal difference learning was written using a similar approach to that in our text and used for our previous project. The *getAction* method returns the action chosen by an ϵ-greedy policy for the given state, and the *updateQValue* method performs a TD update to the Q-Value matrix given the state/action pair and the reward from the previous epoch.

The actual code for TD ended up being rather simple, however implementing it in the context of the problem presented new challenges as there is a substantial workload between agent decisions and the TD implementation had to fit into the context of the problem rather than building the problem around it as had been done in previous assignments.

VI. Results Analysis

While both the Python and C versions of the project are functional, the results presented here are derived from the C version of the project. The reasoning for this is due to the increased performance from using C which greatly reduces the time needed for training. Additionally, working in C is closer to the spirit of the problem, as you will likely not find kernel code written in Python. However, for standardization purposes, the project is also available in Python and these same tests may be ran on the Python implementation, albeit slower.

A. Experiment Setup

The tests were conducted on an Intel i5-8350U CPU with 16 GB of memory, though these tests were CPU bound and not memory bound. As described in the previous sections, a page selector was utilized to choose virtual pages that if managed properly, could yield the most benefit to the performance. Each chosen page was allocated a Q-Value matrix, and each had its state tracked separately. The Q-Value matrices were initialized to 0. Since the reward values are always negative, this encourages the agents to explore the action space at the beginning. Additionally, an ϵ-greedy policy with an ϵ value of 0.1 was used for the experiments when making decisions.

A memory access trace file was generated using Intel's PIN tool from the lulesh benchmark comprising 1 million accesses. Every 10000 accesses the scheduler would step in and adjust the positioning of the virtual pages present in memory. This trace was read in and parsed line by line repeatedly for 10000 epochs. There is concern for potential over fitting by not randomizing the input, though this particular problem relies on the model learning how to handle a specific page. Multiple runs of an application will have randomized virtual address ranges and page numbers will vary between runs, making it difficult to carry over previous learned experience.

B. Results

Figure 4 shows the results for training the scheduler using the page hit counts of the previous scheduling epoch relative to the history scheduler. For this experiment, the 4 pages with the largest benefit metrics were chosen to be used with the RL schedule, and the remainder were scheduled using the history scheduler. As can be seen, the agent rapidly converges to around the performance of the history scheduler. This makes sense as the state space is relatively small compared to the training set size and the information provided by the state is very similar to that used in the logic of a history scheduler. The agent learns to behave like the history page scheduler and achieves comparable results, albeit somewhat worse due to the occasional sub-optimal decision made by the ϵ-greedy policy. Adjusting the γ value for TD(0) did not appear to affect performance much in this case either.

Figure 5 shows the results for training the scheduler using the page hit counts of the previous 2 scheduling epochs. While this greatly increases the state space of the problem, the reasoning behind this approach was to provide the agent

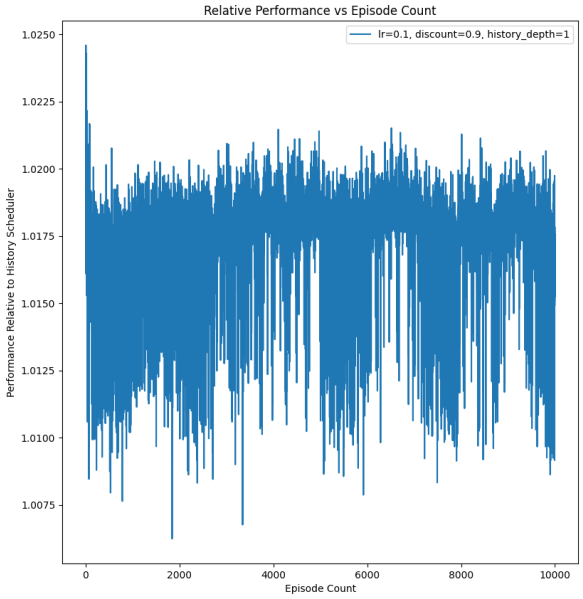

Fig. 4. Relative performance to history scheduler using 1 level of page hit history (200 states). Lower is better.

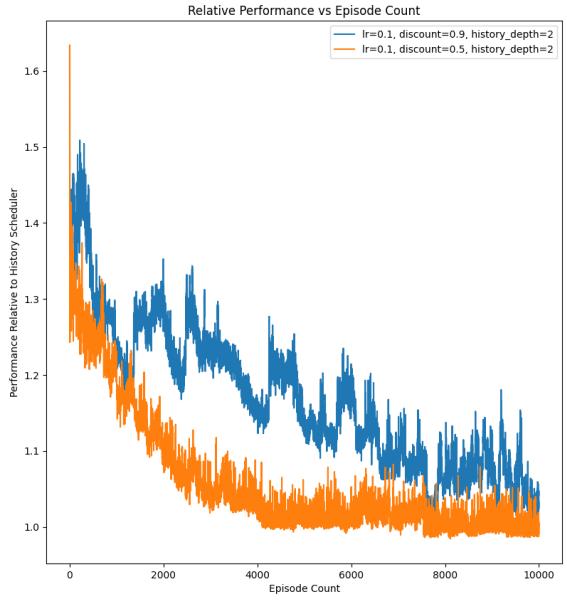

Fig. 5. Relative performance to history scheduler using 2 levels of page hit history (20000 states). Lower is better.

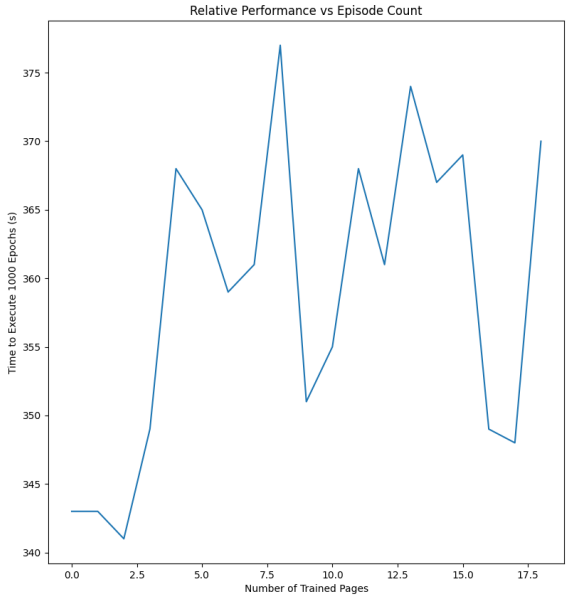

Fig. 6. Simulation run time in seconds to perform 1000 epochs as number of pages trained increases.

with an idea of the 'momentum' of the data accesses, similar to feeding the agent multiple frames when training on classic games like PONG. As can be seen, this increased state space resulted in a much slower rate of learning for the agents. Reducing the γ value for TD(0) did speed up the learning rate, which converged closer to the history scheduler which does not take into account future values. It should be noted that the the $\gamma = 0.5$ experiment did end with an impressive performance value that was 1% better than that of the history scheduler. These models could potentially continue to improve with time.

Figure 6 examines the computational complexity of the project and illustrates the time needed for the program to execute 1000 epochs as the number of pages being selected increases. While the execution time does generally increase as the number of pages increase, it is not the unreasonable amount of growth that may be expected from a naive RL implementation. One explanation for why we do not see larger execution times is that only the chosen pages who are present in physical memory during a scheduling call are trained, meaning, if there are 24 pages that have been selected for RL training, but only 3 are present in memory when the scheduler is called, only those 3 will be trained.

Both of the experiments covered in figures 4 and 5 converged to around the same level of performance as the history scheduler. On the other hand, Doudali et al. were able to achieve performance better than the history scheduler presenting the opportunity to examine what may be preventing the agent from exceeding the bounds of the history scheduler.

One possible reason for the difference in performance is the finer granularity offered by the RNNs used in Kleio. Kleio's RNNs would generate predictions for hit counts for the next scheduling epoch and would be used to assign the page to a memory device. For this project, however, hit count predictions were not generated, but instead pages were placed into fast or slow memory based off of their past hit counts aggregated into denominations of 100. Meaning, a page that had 1 hit was treated the same as a page that had 99. These classifications ignored the values of the other pages, whereas Kleio's implementation sorted the historical values and generated values together in order to choose the page ordering.

Another possible improvement would be to further expand the state space by giving the model access to a longer history of page accesses. However, this runs into the problem of a rapidly expanding state space. A potential solution then would be to increase the denomination sizes of previous hit counts. For instance, the most recent scheduling epoch's hit counts may be divided into groupings of 100, but the epoch before that could be divided into groupings of 1000.

VII. Conclusion

In conclusion, this project implemented a RL based page scheduling algorithm using TD(0) that performed around the level of a history page scheduler, exceeding it by 1% in one case. While the results did demonstrate learning, the models essentially learned to behave like the history scheduler and did not achieve the same level of performance as Kleio[2].

However, I do believe that RL can still be an option to tackle the problem of page scheduling, but compared to other tools available it is not the easiest approach, and I can understand why Doudali et al. avoided using it in Kleio. Though with careful management, RL could be used. For instance, the problem space was adjusted for this project so that it does not grow at the exponential rate that was expected in the Kleio paper and figure 6 shows that using RL is not completely out of the question so long as the state space is managed intelligently.

Further experimentation can be done by adjusting the state space parameters and RL algorithm. Deep reinforcement learning may be a viable strategy given the proven effectiveness of deep learning for this problem.

Lastly, these results further support the idea that there is no golden algorithm to machine learning, and that the different methods available in the field excel in different cases. It is up to the researcher to choose effectively.

VIII. Appendix

A. Role Delegation

Hayden Coffey: Everything

References

[1] Thomas Dietterich. *Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition*. 1999. DOI: https://arxiv.org/abs/cs/9905014.

[2] Thaleia Doudali et al. "Kleio: a Hybrid Memory Page Scheduler with Machine Intelligence". In: *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'19), Phoenix, AZ* (2019). DOI: https : / / www . blagodurov . net / files / hpdc002 - doudaliA.pdf.

[3] Tomonori Sadamoto, Aranya Chakrabortty, and Junichi Imura. *Fast Online Reinforcement Learning Control using State-Space Dimensionality Reduction*. 2020. DOI: https://arxiv.org/abs/1912.06514. arXiv: 1912.06514 [eess.SY].