SmartOS: Towards Automated Learning and User-Adaptive Resource Allocation in Operating Systems

Sepideh Goodarzy
sepideh.goodarzy@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

Maziyar Nazari
maziyar.nazari@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

Richard Han
richard.han@mq.edu.au
Macquarie University
Sydney, NSW, Australia

Eric Keller
eric.keller@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

Eric Rozner
eric.rozner@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

ABSTRACT

Today's operating systems typically apply a one-size-fits-all approach to resource management, such as applying a scheduler that treats all processes of equal importance. The goal of this paper is to explore a learning-based approach to resource management in modern operating systems in which the OS automatically learns what tasks the user deems to be most important at that time and seamlessly adjusts allocation of CPU, memory, I/O, and network bandwidth to prioritize user preferences on demand. We demonstrate an implementation of such a learning-based OS in Linux and present evaluation results showing that a reinforcement learning-based approach can rapidly learn and adjust system resources to meet user demands.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; • **Computing methodologies** → *Reinforcement learning*; • **Human-centered computing** → Human computer interaction (HCI).

KEYWORDS

Operating systems, Reinforcement Learning, Human Computer interaction

APSys '21, August 24–25, 2021, Hong Kong, China

ACM ISBN 978-1-4503-8698-2/21/08.
https://doi.org/10.1145/3476886.3477519

ACM Reference Format:
Sepideh Goodarzy, Maziyar Nazari, Richard Han, Eric Keller, and Eric Rozner. 2021. SmartOS: Towards Automated Learning and User-Adaptive Resource Allocation in Operating Systems. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3476886.3477519

1 INTRODUCTION

Today's user-facing operating systems (OS)s, such as laptop, mobile, and desktop OSs, are typically designed with a one-size-fits-all approach to resource management. For example, the Linux Completely Fair Scheduler (CFS) effectively divides up the CPU equally among all processes, assigning them essentially the same static priority [1, 13]. In contrast, users interacting with an OS often move from task to task and application to application, wanting sufficient resources devoted to their current task, such as editing a document, chatting via zoom, or listening to music. In addition, users often have many applications open simultaneously, some from previous tasks that may be resumed in the near future, including multiple tabs in a browser persistently refreshing their content, resulting in a landscape of many applications continuing to consume system resources. In this context, the current approach of static prioritization often fails to devote sufficient resources to the applications that the user cares most about at that moment, resulting in degraded performance. For example, many users experience a slow down on their computer when there are numerous open applications which collectively act as CPU hogs or memory hogs, in some cases due to runaway processes, and interfere with the tasks that the user deems most important. Other processes occupy enough network bandwidth to interfere with real-time audio/video, including software updates and cloud synchronization. Anyone who has tried to conduct a Zoom

call while there is another local network-hogging application understands this difficulty. Ideally, a next-generation OS would benefit the user and alleviate these bottlenecks by being able to learn what applications the user currently considers to be most important and adaptively prioritizing allocation of resources to those applications.

Limited performance in today's operating systems has become enough of a concern that machine learning (ML) has begun to be applied to improve application execution. Perhaps the most closely related to this work is the recent Acclaim system that seeks to improve user experience in the Android OS by predicting what memory pages are going to be used next by employing machine learning [12]. Acclaim assumes that the most important applications are the foreground application and audio/video apps, and statically prioritizes page reclamation for these applications. As we show later, static prioritization is not ideal in a variety of circumstances. In addition, machine learning has been applied to improve Linux process scheduling [4, 19], I/O scheduling [11, 14], and network cloud systems [5], but each only considers one resource dimension, rather than jointly allocating CPU, memory, networking and I/O, and also do not learn user preferences for resource allocation.

Some previous work has combined machine learning with control theory to preserve the quality of service while minimizing energy consumption [16]. The issue with this method is that conservative configuration is used until enough data is gathered to do offline learning. Once training is finished, the resource configuration is changed in their control system. As a result, this method can not be used in an interactive environment with the user due to the fact that despite the user giving the system feedback, the system will not change the configuration of resources in a computer until it has enough data that it can do offline training. Other works in the literature used control theory try to find the sweet spot of accuracy and performance trade-off space in dynamic environments [9, 10]. Still, their method is not suitable for our problem space as the goal in our problem space is subjective (user preference) and changes more frequently than objective goals such as application performance and energy consumption which are the focus of those works. This is due to the fact that control theory is based on modeling the environment, while reinforcement learning methods in machine learning are also applicable to environments where there is a lack of knowledge to be modeled perfectly (model-free reinforcement learning).

Building upon this theme, this paper explores the role of machine learning in the design and implementation of next-generation operating systems, harnessing the exploding interest in artificial intelligence and machine learning to improve the user experience through automated learning and resource allocation in the OS. We investigate how the OS may be structured to accommodate learning-based management of joint resource allocation for memory, CPU, I/O, and network bandwidth in response to user behavior. We consider which machine learning algorithms may most effectively integrate and learn from user behavior. We also seek to understand whether there is any net benefit in performance to applying machine learning in OS design and if so to quantify the benefits.

The challenges we face are significant. The context that governs what the user values as most important at any given time is complex and difficult to learn. For example, a user may currently be engaged in editing a document, and would prefer to have CPU and memory resources prioritized towards editing. Adding complexity, the user may also be streaming a music video from a cloud provider at the same time and wish to listen to music while editing. This different modality should also receive high priority in terms of CPU, memory, and notably network bandwidth. Static prioritization policies become difficult to craft as we consider such increasing complexity. Heightening the complexity, each application may consist of multiple dependent communicating processes who would as a group need to receive elevated allocation.

This paper proposes SmartOS, a learning-based operating system that takes the user interactions with the OS into account to perform automated resource management. SmartOS leverages reinforcement learning to continuously gather feedback from the environment and make changes in the resource parameters exposed by the Linux operating system's kernel to improve the user experience. The contributions of this paper are as follows:

- We describe an architecture that integrates machine learning into the OS as a user space module controlling allocation of memory, CPU, network, and I/O.
- We utilize reinforcement learning (RL) to solve the difficult challenge of learning the user's context and applying the appropriate resource allocations.
- We demonstrate that our RL algorithm is able to rapidly converge to the desired allocation of system resources and that the overhead is modest.

In the following, we first describe our design goals and overall architecture in Section 2, then explain the system implementation in Section 3, followed by the evaluation results in Section 4, discussion and future work, and finally conclusions.

2 OVERVIEW

Figure 1 depicts an overview of SmartOS. The main learning component of the system is the Cortex, which is the "brain" of the system and is responsible for monitoring the application status, user context, and the user's interaction with the computer. In response to this state information, the Cortex

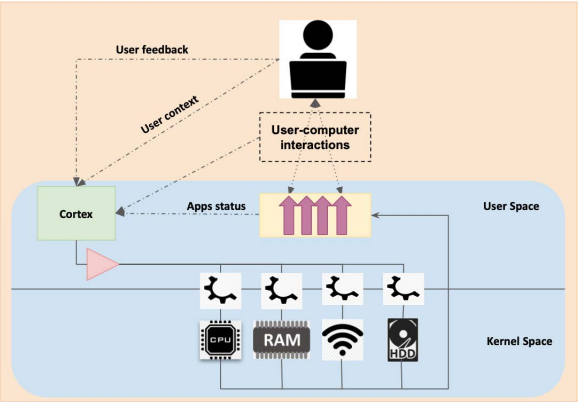

Figure 1: Learning-based SmartOS architecture.

determines which resource allocation policy is the best in satisfying the user's expectations. The Cortex applies the determined resource allocation policy through parameters exposed by the kernel. The kernel then allocates CPU, memory, network bandwidth, and disk I/O among applications according to the specified parameters. The new resource distribution changes the quality of the user experience. To indicate the resource policy has successfully achieved its objectives, the user provides feedback based on their experience to the Cortex.

The overall architecture and design of the SmartOS system reminds us of Reinforcement Learning (RL). RL consists of an agent that measures the current state of the environment, acts on the environment, and watches the environment's next state. Also, it receives a reward based on how that action has affected the environment to reach its objective. In general, the RL agent's objective is to maximize the total gained reward. As a result, the instant gain of action is not as valuable as the entire gain of a set of actions over time. The nature of RL makes it suitable for decision-making problems.

Due to the similarity between the SmartOS system and RL design, we decided to utilize an RL algorithm in the Cortex. Another reason for this decision is that the OS constantly decides on sharing of resources among processes and scheduling processes. This is highly suitable to RL, which is also constantly adapting its decisions based on the available state information. Other machine learning techniques such as supervised learning and unsupervised learning are not very suitable for our problem as they cannot handle the dynamic nature of our problem. However, the correct identification of the environment's current state, set of possible actions, and reward makes this approach challenging [16].

In pursuing this Cortex model for integrating user-adaptive learning into an OS, we chose to locate the Cortex component in user space. This affords us the ability to easily manage, test, debug, and update the software and learning algorithms. Another approach is to place the learning component within kernel space. This can improve performance. For example, KMLib provides a fast library framework for ML applications [3] in kernel space. However, using such a library is difficult because of the challenge of managing and debugging code in kernel space. It is also a trend in the industry to move applications to user space to prevent tight coupling and slow release cycles [15]. We show in section 4 that the performance of our RL-based system in user space is quite responsive given the human time scales for adaptation.

Actions: In order to distribute the computer resources among different applications, one can use the parameters in the kernel or change the kernel code. We have chosen to use the predefined parameters to control resource allocation to be compatible with different Linux versions (see Section 3). This choice will enable us to write our system in user space and will make it easily pluggable.

Among different resources in a computer, the CPU, memory, network, and disk I/O are the main focus of our system as their allocations are critical to application performance.

Rewards: The best reward selection is user feedback because SmartOS's final goal is to find the best resource allocation policy based on user preferences. Therefore the user is the only person who can show how effective the Cortex was in reaching its objective. This feedback can be implicit and non-intrusive based on passively monitored interactions such as keyboard strokes or mouse clicks and movements. For example, when the user is frustrated with their computer, they may move the mouse quickly or type more quickly. The feedback also can be explicit via a specialized application for providing feedback.

3 PROTOTYPE

The Cortex of SmartOS was implemented on Linux, Ubuntu 20.04 in user space. As a first step to test the feasibility of employing RL to automatically tune parameters, we constructed simplified discrete-valued models for the environmental states, actions, and rewards, reasoning that if we can show benefits in such a system, then we can later address the complexity of continuous-valued states. We limited the environment state to the applications' resource usage profiles, whether the applications are foreground or background, and whether the application is a video/audio application. If an application is in the foreground, the foreground value in the environment's current state vector will be one and zero otherwise. As noted in the Acclaim work, whether a process is in the foreground or background state is a key indicator of what the user views as being important at that moment, though as we shall see statically prioritizing the foreground is not the whole story. Similarly, the value is one

if an application is video/audio or zero otherwise. This allows us to examine how the modality of the application must be considered in learning the best resource allocation. As for the CPU, memory, network, and disk I/O values in the state vector, if the application is intensive in the consumption of any of these resources, the corresponding vector index for that resource type will be one otherwise zero. By choosing four independent resource dimensions, this allows us to examine cases where high priority applications may demand intensive resources in certain dimensions while being interfered with by other applications in a variety of complex ways.

We also limited the range of the possible actions. A value of one in the action vector for any types of resource means high priority in that corresponding resource and otherwise normal priority.

As for the reward, to test the space of a wide variety of user behavior, we created a script that generates user feedback synthetically. This gives us more freedom to explore many different types of user behavior, both from a user resource perspective as well as a temporal perspective. The script only gives +1 as a reward for the best action, and 0 otherwise. The best action is the resource allocation policy that results in highest performance in the applications that are important to the user in a given scenario.

Resource allocation: We then leveraged various system tools in Linux to implement changes in relative allocation of system resources to each application, effectively giving us the ability to change the prioritization of different processes independently in four separate dimensions: CPU, memory, I/O, and network bandwidth. The RL algorithm would manipulate these system "knobs" and then inspect whether it had converged towards the best allocation of resources. Different resource parameters are controlled as follows:

- **CPU:** We used the *nice* value of -20 for high priority and 0 for normal priority to control CPU allocation.
- **Memory:** To control the memory, we used the parameters *oom adjacent score* and *cgroup memory swappiness*. To set a high priority for memory for an application, we used −1000 and 0, for *oom adjacent score* and *cgroup memory swappiness*, respectively, and for normal priority, we used 0 and 60.
- **Network:** We utilized *cgroup netprio ifpriomap* to distribute the network bandwidth among applications. For high priority and regular priority applications, we used 10 and 0 respectively as priority values.
- **Disk I/O:** To manage the Disk I/O, we leveraged *ionice*. We used a real-time class with priority 0 for high priority applications and left the priority of the standard application as default (idle class with the priority of 4).

Reinforcement Learning algorithm: For implementation of the automated learning, we employed the Monte Carlo Reinforcement Learning algorithm, which is shown in Section 4.5 to have the best convergence rate compared to other methods in Table 1. For testing some of the algorithms such as DQN and A2C, we used [8], and for the other algorithms, we used Python 3.6 [2] and the Numpy [6] package.

4 EVALUATION

We sought to compare the automated learning approach of SmartOS, which applied RL to learn the proper allocation of CPU, memory, I/O, and network bandwidth, with a variety of static prioritization schemes. The following are the different static prioritization heuristics that were compared against:

Fg only: This heuristic sets the CPU, memory, network, and disk I/O parameter of the Foreground application to high priority.

Fg + video/audio: This heuristic inspired by [12], adds to the previous heuristic by also giving high priority in all four resource dimensions to a video/audio application resulting in competition in some dimensions for resources with the foreground application. The idea was to test the case where a user may be editing a document while listening to music or playing a video.

Fg + dependent: It gives high priority to the foreground application and all other applications that foreground performance depends on. It identifies the mentioned applications by a predefined directed acyclic graph. We specify the DAG based on our common observation of what applications each application usually depends on. We aim to test the case where an application on a computer may consist of a set of dependent processes that communicate via network message passing, as is the case for complex applications like browsers and video/audio.

Multi-dimensions: This heuristic uses a predefined map that stores the essential resources per application's performance based on our common observation of the applications. Then, it will prioritize the foreground application in all resources necessary to its performance. After that, if there are any remaining resources that are not assigned to the foreground application, it will assign them to the important applications to the user. These important applications are also stored in a hash map defined by asking the user in the beginning. The intent is to examine the case where the foreground application may have variance in its resource needs while competing with applications that may have resource needs in different dimensions.

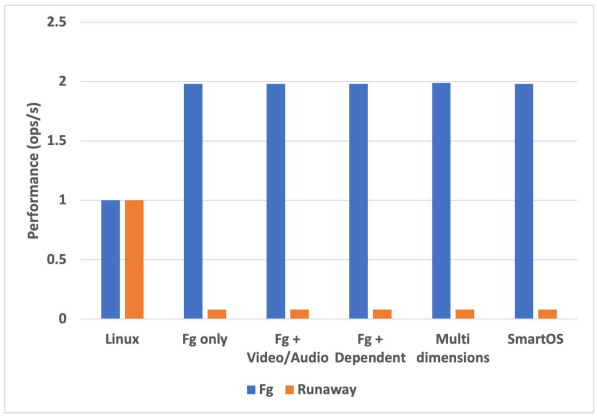

Figure 2: Foreground app contends with a resource-intensive background app.

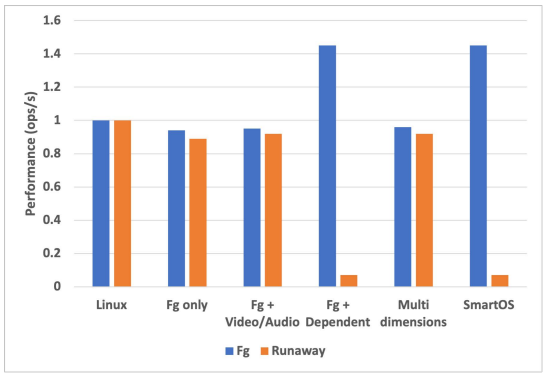

Figure 3: Foreground underperforming due to a dependent application.

4.1 Foreground contends with resource-intensive background

In this experiment, we set up a scenario to observe how well our SmartOS' automated RL utilizes resources compared to the aforementioned static prioritization schemes as well as Linux' base CFS scheduling algorithm. In this scenario, a foreground application that should be given highest priority for all resources to maximize user experience is forced to compete with a resource-intensive background application that is not as important to the user. Both the foreground and runaway background applications are implemented as stress applications that make intensive equal use of the CPU, memory, and disk I/O, consuming the same resources on the same core in an Ubuntu 20.04 virtual machine with 8 GB of memory, one processor, and 50 GB VDI disk drive.

In Figure 2, the blue and orange bars are showing the performance of the foreground application and background application correspondingly proportional to base Linux over 60 seconds of execution. As we can see, base CFS-based Linux gives equal priority to both the foreground and background, so that the foreground application is not able to make as much progress as in other policies. In contrast, for each of the heuristic static prioritization policies, the foreground is able to run at twice the ops/sec rate of the base Linux case, while the background application is appropriately given scant resources. Similarly, SmartOS' user-adaptive RL strategy is able to learn the correct policy and converge to the same actions, prioritizing the foreground application.

4.2 Foreground underperforms due to a dependent application

In this experiment, a stress app runs as a foreground app that consumes CPU, memory, and disk I/O and is communicating through blocking pipes with another process that is also a stress app and is intensive in consuming the same resources as the foreground application. We are also running a third runaway application which is additionally intensive in CPU, memory, and disk I/O usage. All of these are running on the same core. Hence, these three applications are competing on attaining CPU, memory, and disk I/O. We can instantly see in Figure 3 how the Fg only heuristic does not achieve good performance as it only changes the priority of the foreground app to high priority and leaves the background app unchanged. Thus, the foreground app, which is messaging through pipes with the background app, will remain in the blocking stage for a significant portion of its execution time, making the CPU available to the runaway application, resulting in an undesired resource allocation, which leads to user frustration. The same also happens with the Fg+Video/Audio and Multi-dimension heuristics. The only successful heuristics in this experiment are the Fg+dependent static policy and SmartOS, which are able to converge to the best resource allocation decision.

4.3 Multiple important applications with needs in separate dimensions

Sometimes the performance of other applications besides the foreground application is also crucial for enhancing the user experience. An example for this scenario would be when a user is working inside a document editing application, but is also monitoring a stock widget on their laptop screen, or listening to music. The stock widget or music is not a foreground application, but its performance is important to the user. Suppose the other applications critical to the user are consuming different kinds of resources from the foreground application. In that case, the priority for all resources should not be given to the foreground application. In order to test such a scenario, we used a VM with four cores, 3 GB of RAM, and 50 GB of VDI hard disk with installed

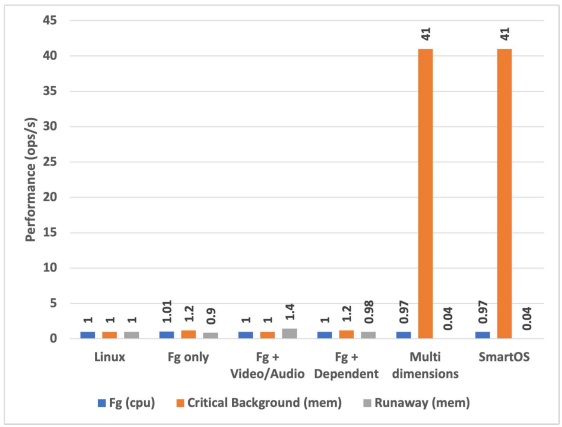

Figure 4: Multiple important applications with resource needs in separate dimensions.

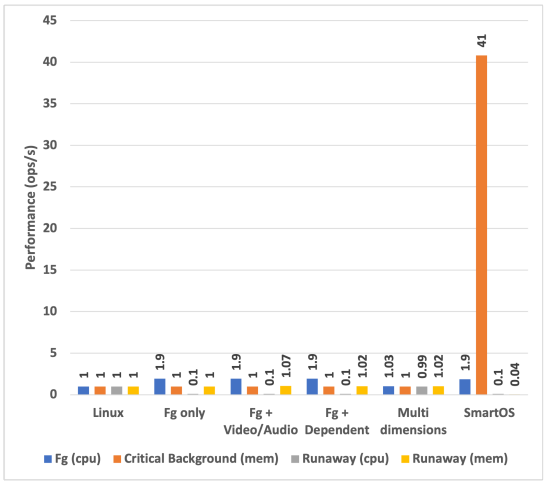

Figure 5: Variation of dimensions.

Ubuntu 20.04. We ran a stress app that is CPU intensive as a foreground application on core one and two, and memory-intensive applications on cores 2 and 3, respectively. One of the memory-intensive applications plays an important factor in the user experience, and the other one is a runaway application. As all applications are running on different cores, they are not competing for more computation. As a result, giving more priority to the foreground application will not affect its performance. Thus the Fg only, Fg+ Video/Audio, and Fg+Dependent static prioritization policies will not improve the user experience. However, the multi-dimensional heuristic is able to achieve a better user experience as it only gives priority in CPU to the foreground application and gives more priority in memory to the critical background application. The two memory-intensive applications compete over the memory since they can not be fit simultaneously in the memory and should be swapped out to disk space. As a result, giving more memory priority to the critical application can enrich the user experience because it prevents swapping out of the critical application. As we can see in Figure 4, we can also observe the same result. SmartOS is able to learn and also reach the same performance as the multi-dimensional static prioritization approach.

4.4 Variation of dimensions

We next constructed an experiment with random assignments of the importance of various applications and their resource needs. A random generator first randomly chose CPU as the resource needed for a stress application running as a foreground application on core one. It also randomly chose CPU for the second stress application competing with the foreground running on the same core. After that, the random generator chose memory as the resource needed for applications three and four. These two memory-intensive applications are running on core two and core three competing over memory. One of these two applications' performance is critical to the user experience quality, and another one is a runaway application. Figure 5 shows that the static multi-dimensional heuristic is unable to allocate resources efficiently, because it assigns memory as the resource needed for the foreground application and the second application and CPU as the resource required for the third and the fourth applications, which differs from what these applications currently need as defined by random assignment. The other three heuristics, Fg only, Fg+Video/Audio, and Fg+dependent, successfully improve the foreground application, but they don't change the critical application's performance. Only SmartOS is able to successfully enhance both the foreground and the critical application's performance.

Table 1: Different Reinforce Learning methods convergence in a dynamic setting.

RL Algorithm	Feedbacks#	Episodes#
DQN [18]	52000	13000
QLearning [22]	28400	7100
Sarsa [20]	3680	920
Double Qlearning [7]	2400	600
A2C [17]	1600	400
Monte Carlo [21]	400	100

4.5 SmartOS dynamicity and convergence

We next examine how quickly SmartOS can adapt and converge to appropriate resource allocations based on user feedback. We designed an experiment that combined all the previous experiments. In other words, we execute a script that runs all the applications in Section 4.1 for 60 seconds. Meanwhile, SmartOS applies a resource allocation policy and asks the script for its feedback. After 60 seconds, the script closes all the running applications in Section 4.1 and starts all the applications in Sec 4.2 for 60 seconds, etc. This same procedure is applied then to the applications in Section 4.3 and Section 4.4. Each repetition of Section 4.1 to Section 4.4 is called an *Episode*. After an episode completes, we repeat the process all over again with a new episode. Table 1 shows the number of feedbacks required for each reinforcement learning algorithm to find the best collection of policies. We found Monte Carlo to converge most rapidly compared with other reinforcement learning algorithms.

Figure 6: Convergence of Monte Carlo in a dynamic setting (each episode consists of 4 feedbacks).

Figure 6 provides a detailed temporal perspective of how SmartOS can achieve the best set of policies and a maximum reward of 4 (one for each distinguished experiment) after receiving 400 feedbacks using the Monte Carlo method in 100 episodes. We see that there is rapid convergence early in the learning process. Note what is pictured is a smoothed average over 10 episodes. Though episode 100 appears not to achieve the maximum award, namely full convergence of 4, the unsmoothed value attained a value of 4 so that Monte Carlo reached full convergence by episode 100. While this experiment combined many iterations of different application scenarios, we note that for just a single application scenario of the foreground application only considered in section 4.1, convergence was achieved in just 8 steps.

We used the Monte Carlo implementation of RL in the SmartOS Cortex as a basis for performance measurements. SmartOS adapted to each user feedback in 0.218 ms of total execution time, of which 0.21 ms of that time consisted of purely CPU execution, and the rest contains context switch time. These results seem to be reasonably responsive to user feedback without excessively burdening Linux, given that time scales for human adaptation are on the order of seconds. Also, the required memory to run the Cortex application was 23.1 MB.

5 DISCUSSION & FUTURE WORK

We desire to focus on the following future work areas:

Real-world SmartOS - SmartOS needs to be tested under realistic use cases, i.e., working with several typical applications. We plan to conduct a user study to show how it will impact the user experience interacting with the OS and applications. Besides, we are planning to collect implicit feedback like pressing keyboard keys abnormally from real users in the human study to experiment with how well SmartOS can adapt according to user behavior. We plan to conduct IRB-approved human user studies with SmartOS, continuous-valued vector and state evaluation, and incorporate more complex user context.

SmartOS Performance- SmartOS should be able to perform equally or better than native Linux, for example, CFS for scheduling. Thus, using the collected feedback, SmartOS needs to realize when to gracefully degrade to the native CFS scheduler in case of failures resulting from RL algorithm decisions.

Cross-platform SmartOS- It should be noted that SmartOS can perform cross-platform. Its cortex can be placed somewhere in the cloud or edge cloud and talk to its RL agents to adjust necessary parameters. So, this enables SmartOS to gather data from someone's mobile, desktop, etc., at the same time and then make decisions. Cloud-based flexibility could also enable aggregated learning across users.

6 CONCLUSIONS

We presented SmartOS, a learning-based operating system that implements reinforcement learning to automatically adjust allocation of memory, CPU, I/O, and network bandwidth according to learned user preferences. SmartOS is implemented in Linux user space, and our test results show SmartOS is able to automatically adapt to increasingly complex allocation scenarios unlike static prioritization policies. We also showed a Monte Carlo RL algorithm achieved the fastest convergence in terms of its learning rate, and that its overhead was on the order of tenths of milliseconds.

7 ACKNOWLEDGEMENT

This research was supported in part by VMware and the NSF as part of SDI-CSCS award number 1700527, and by the NSF as part of CAREER award number 1652698.

REFERENCES

[1] [n.d.]. *Completely Fair Scheduler*. Retrieved July 14, 2021 from https://man7.org/linux/man-pages/man7/sched.7.html

[2] [n.d.]. *Python 3.6.0*. Retrieved May 26, 2021 from https://www.python.org/downloads/release/python-360/

[3] Ibrahim Umit Akgun, Ali Selman Aydin, and Erez Zadok. 2020. KMLIB: Towards Machine Learning for Operating Systems. In *Proceedings of the On-Device Intelligence Workshop, co-located with the MLSys Conference*. 1–6.

[4] Siddharth Dias, Sidharth Naik, Sreepraneeth K, Sumedha Raman, and Namratha M. 2017. A Machine Learning Approach for Improving Process Scheduling: A Survey. *International Journal of Computer Trends and Technology (IJCTT)* 43, 1 (2017), 1–4. https://doi.org/10.14445/22312803/IJCTT-V43P101

[5] Sepideh Goodarzy, Maziyar Nazari, Richard Han, Eric Keller, and Eric Rozner. 2020. Resource Management in Cloud Computing Using Machine Learning: A Survey. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 811–816. https://doi.org/10.1109/ICMLA51294.2020.00132

[6] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[7] Hado Hasselt. 2010. Double Q-learning. In *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta (Eds.), Vol. 23. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf

[8] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. 2018. Stable Baselines. https://github.com/hill-a/stable-baselines.

[9] Henry Hoffmann. 2015. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 198–214.

[10] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic knobs for responsive power-aware computing. *ACM SIGARCH computer architecture news* 39, 1 (2011), 199–212.

[11] Julian Kunkel, Michaela Zimmer, and Eugen Betke. 2015. Predicting Performance of Non-contiguous I/O with Machine Learning. In *High Performance Computing*, Julian M. Kunkel and Thomas Ludwig (Eds.). Springer International Publishing, Cham, 257–273.

[12] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 897–910. https://www.usenix.org/conference/atc20/presentation/liang-yu

[13] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326

[14] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M. Wild. 2018. Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems. In *High Performance Computing*, Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis (Eds.). Springer International Publishing, Cham, 184–204.

[15] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 399–413.

[16] Nikita Mishra, Connor Imes, John D Lafferty, and Henry Hoffmann. 2018. Caloree: Learning control for predictable latency and low energy. *ACM SIGPLAN Notices* 53, 2 (2018), 184–198.

[17] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1928–1937.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[19] Atul Negi and P. Kishore Kumar. 2005. Applying Machine Learning Techniques to Improve Linux Process Scheduling. In *TENCON 2005 - 2005 IEEE Region 10 Conference*. 1–6. https://doi.org/10.1109/TENCON.2005.300837

[20] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK.

[21] David Silver. 2015. Lectures on Reinforcement Learning. URL: https://www.davidsilver.uk/teaching/.

[22] CJC Watkins. 1989. H. Learning from Delayed Rewards, Ph. D. Thesis, Cambridge University, 1989. (1989).