IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Integration of Machine Learning into Operating Systems: A Survey

Mayuresh Kulkarni
Department of Computer Engineering
Bharati Vidyapeeth College of Engineering
Navi Mumbai, India

Torana Kamble
Assistant Professor, Department of Computer Engineering
Bharati Vidyapeeth College of Engineering
Navi Mumbai, India

***Abstract*—Struggle for resources has always been an integral part of life of a process. The task of management of resources such as RAM, storage, CPU, GPU, etc. is handled by the operating system of the computer. Every millisecond is crucial when user experience and system performance are at stake. Machine learning has made an entry into the world of computers and has successfully conquered majority of the domains. This research paper attempts to study the various machine learning techniques used to automate and simplify the job of process scheduling in operating systems. Various algorithms, their approach towards the system (at hardware and software levels), time-scale compression, etc. previously researched and discovered are discussed in this text. This paper can be reliably used as a comprehensive summary of 3 different research papers and their outcomes regarding the concept of process scheduling, CPU scheduling, system optimization with machine learning at different levels (like User level, kernel level and hardware level), etc. A discussion about a dedicated resource (inside a computer) specifically for handling machine learning tasks has been made which has a near unity ratio of success.**

Keywords—component; formatting; style; styling; insert (key words)

I. INTRODUCTION

Operating Systems (OS) are what defines the user experience, performance and multitasking capabilities of the computer. OSes form one of the most complex and intricate pieces of software, closely followed by the recently developing gaming mammoths. Operating systems have been in existence since the year 1956, made by General Motor's Research division for IBM-704 Mainframe computer. And since then, OSes have evolved to be completely different than what they were during '56. Current operating systems perform tasks like CPU scheduling, file management, memory management, provide security from viruses, provide the user interface, etc. Parallelly, the field of Machine Learning (ML) has also been evolving since past few years. Beginning with mathematically evolving formulae, to providing a real-life software assistant, ML has also grown to something which nobody could think of when the first OS was introduced. A combination of these 2 rapidly evolving domains of Computer Science will be extremely beneficial for both. Integrating Machine Learning into operating systems is one of the most difficult tasks as of today. The researchers or programmers must take into consideration hundreds of factors and their implications on the system in real life.

It is a widely known fact that tasks like Machine Learning, Deep Learning, NLP (Natural Language Processing), or the whole Artificial Intelligence domain as a whole, require more computing power than a normal task (like opening a file or reading a file). From operating system's point of view, performing machine learning is same as rendering a high-resolution image from disk to memory via GPU. This is because they both require the same amount of resources from the system. Hence, the OS tries to treat the ML task like a regular task, and this is where the problems begin. It is expected for any regular task to have a bust time, after which it has to end or be terminated forcefully. Machine learning tasks are bound to run for a much longer amount of time, or infinitely until system is turned off or it doesn't crash. Therefore, treating an ML task like a regular task is the first thing that OSes do wrong. Another issue with current generation of operating systems is that, at root level, they treat all languages as equals. Meaning, for a Linux based operating system, at kernel level, C is same as C++ is same as Python. To summarize, OS treating high level languages to low level languages is the second thing that OSes do wrong.

This survey paper aims to study, this combination of Machine Learning and Operating System, and comment on the credibility to do so. This paper also attempts to provide a practical solution to management of resources while the ML algorithms run in the background, by giving them dedicated hardware for it. This theoretically will not consume the resources provided for the user to consume because this dedicated hardware is not assessible to any other entity other than the ML algorithms and the operating system kernel. Further explanation and detailing on this topic are provided at the end of the text.

II. METHODOLOGY

The methodology that will be followed during this text is that each of the research papers that this text deals with, will be studied indivisually. Their offerings, their scope, their results, everything will be discussed at a survey level. Beginning with
[5]. This research paper tells where Machine Leaning can be used in Operating Systems, learning configurations, challenges that researchers may face while integrating ML into Operating System and potential solutions to those challenges and problems. It says that current operating systems cannot
'change' at runtime, making them incapable to dynamically adapt to applications' changing behavior and needs. The learning

configurations that the operating systems use are categorized into 2 types, Timing Related and Size Relatedconfigurations. The author states using Objective-C instead of C++ will be much more beneficial here because Objective-C is focused on runtime-decisions for dispatching & heavily depends on its runtime library to handle inheritance and polymorphism, while in C++ the focus usually lies on static, compile time, decisions.

III. Research Paper 1: 'Learned' Operating Systems

One of the most important facts provided in this paper is that it states the operating systems do not change at 'runtime'. Now, this statement might be contradicting to the very concept of machine learning, i.e. ability of programs to adapt to changing situations. But, through this text, it can be explained by giving the logical explanation behind it. Machine learning algorithms work on data provided to them. Which means the data, or the instance of generation of data is in the past. Secondly, based on 'n' such data sets, the algorithm modifies the system behavior and the modified behavior is reflected into the system from the *next* run. That means the modification done to the system in the current run will be shown in the *next* run. Therefore, operating systems do not 'change' at runtime. The statement remains valid.

The paper [5] discusses 3 different opportunities for machine learning to be integrated into operating systems. Namely, Learning Configurations, Learning Policies and Learning Mechanisms [5]. Some settings and tools are preset in OSes. They are configured to stay static. To modify them, one needs to manually edit them. Machine learning can automate this process.

Configurations can be sub-divided into 2 parts, Timing Related Configurations and Size Related Configurations. Setting timing related configs is very difficult. Because system has to make a decision between less-aggressive thread scheduling (low CPU consumption but high effective CPU utilization) and more-aggressive thread scheduling i.e. more pre-empting and context switching threads (high CPU consumption as it is always under maximum load but low effective usage as majority threads handled are for context switching and not for process execution). This duality can be reduced using Machine Learning. Setting size related configurations is finding a balance between improving stored file system performance and available memory for other applications. This is also very difficult trade-off as large buffer cache improves the performance of the system but on the other hand reduces amount of free memory. This paper provides a formidable solution to this. Using Objective-C instead of C++ will be much more beneficial here because Objective-C is focused on runtime-decisions for dispatching & heavily depends on its runtime library to handle inheritance and polymorphism, while in C++ the focus usually lies on static, compile time, decisions.

Policies are ways to choose which activities to perform. Similar to configurations, policies can be divided into 3 sub-categories, Space Allocation Policies, Scheduling Policies and Cache Management Policies. Machine Learning models will be able to combine the benefits of Best Fit & *ext* policies (these are the polices currently used in Linux based operating systems). It is necessary to analyze how much space users requested, what space the OS allocated, how efficiently spaces are used and how much fragmentation is done. Also, modelshould be dynamic because of the necessity to adapt to changing user-behaviors.

An OS needs to decide which (& how much) memory space to allot, whenever an application requests for memory. This task is executed based on general heuristics and aged algorithms. Techniques like Best-Fit policy, mmap system call in Linux, etc. can work for many workloads and usages, they are nowhere close to what Machine Learning can achieve for the same task. For ML to work, a model is needs to be created such that it will track which application uses how much memory, which sections of memory is occupies, what is the priority of resource allocation, etc. and form a general trend based on this data for every application. After 'n' trials, the model can predict the memory demand and accordingly keep those parts of memory blocks free for that application.

The scheduling policies of the current gen OSes are very outdated and not suitable for extended heavy-usage which changes at rate more than the algorithm can handle. This results in unexpected crashing of applications and failing to be productive. Therefore, like learning policies, the scheduling algorithms should also be given ML treatment to improve CPU scheduling. Presently, most OS virtual memory systems try to swap memory pages (least used) using approximate LRU detection algorithms. Instead of using traditional LRU techniques, OSes can use ML models which can dynamically locate and update the policies which deal with cache management.

Mechanisms are the implementations that enforce policies, and often depend to some extent on the hardware on which the operating system runs. For instance, a processes may be granted resources using the first come, first serve policy. This policy may be implemented using a queue of requests. Often the kernel provides mechanisms that are used to implement policies in servers.

Like configurations and policies, mechanisms can also be divided into 2 sub-categories, first is Virtual Memory Address to Physical Memory Address Location Mapping mechanism; and second is mapping from File Name & Offset to Disk Logical Block and Address mechanism. The paper states that these 2 mapping mechanisms are extremely crucial when dealing with performance of memory and storage systems. This is in-fact true because every bit of data that is transferred from main memory to secondary memory, and vice versa, for each and every task executed in the system, is done through these 2 mechanisms. Failure to perform at high speeds directly imply reduced system performance. It is possible to further improve the learning mechanisms' performance and CPU cache hit-rate by storing the parameters used by the model in contiguous memory space.

After reviewing the possible areas where machine learning can be implemented, it is necessary to audit the problems or difficulties that might be faced while doing so. The possible difficulties that might arise are Model Selection and Building, Training of the Model, Integration Difficulties and Security of the final operating system with the ML model integrated in the system.

Model selection is a very crucial decision in the long run. A correctly selected model might improve the system performance and increase efficiency; but an incorrect selection might reduce the performance and deteriorate efficiency. Next,

it is also essential to think about a situation of unavailability of a model. It is very much possible that there is no model which matches the requirements of the project in-hand. In such cases, the developers are expected to build their own model and use it for the same. Global models can be used/made if its job is general purpose and doesn't affect the system performance much. Example, pre-fetching of network modules, post-processing of graphics data, etc. GPMs (General Purpose Model) can be used in such situations. If the model is going to tasks which directly affect the system performance, it is highly recommended to make very accurate and job-specific models, even if that particular model cannot be used elsewhere. The reason being fine-grained models always achieve better accuracy and performance in the long run than GPMs (which generally tend to break after some time due to over-training and more modification than the architecture allows).

Training of the model has to be very operation specific. Model which will be dealing with CPU scheduling should not be trained in environment where memory management or file system management or any such domain adulteration will occur. Another important point to be kept in mind while training the model, is that, even if the model is trained using supervised or unsupervised learning methods; the way in which the model is tested is very much important. As a proven fact, reinforcement learning comes out to be the best among the 3 methods for testing models in real life scenarios. Secondly, it is possible to use theoretical facts to evaluate the model's performance in testing. For example, if the model indicates increasing the clock speed of the CPU will increase the process execution time, it is necessary to check how much increase in frequency is the model indicating. If the frequency that the model says is more than the frequency the CPU can generate, the model is definitely not performing well. On the other hand, if the model is suggesting moving some processes to other cores for parallel execution will be beneficial (if the processes are mutually exclusive), then it can be said that the model is working correctly.

On a superficial level, it is possible to divide all of the configurations and the policies into 2 categories. First category, in which these configurations and polices need to run only once in a while. Therefore, they have a liberty to take some time to increase their accuracy. Costly machine learning models which are very aggressive and take little time, but give excellent results, can be used here. Second category, where 'decisions' must be made very quickly, and their decisions affect the performance of the system for a small amount of time (or time for which the decision's effect lasts). Configurations and policies pertaining to storage systems and networking devices can be included here. Realtime thread scheduling and core management algorithms can be included in this category.

The paper also puts up a query regarding storage of these ML models. Accepted that good models can occupy hundreds of Megabytes of space; but nowhere it is mentioned that these models have to be stored in the RAM. These models which take up lots of space can be paged and can be accessed from there. Otherwise, if the system has powerful specifications, one or two important and frequently required models can be put onto RAM for increasing data-accessing speed.

IV. Research Paper 2: A Machine Learning Approach for Improving Process Scheduling – A Survey

This is another survey paper which provides an in-depth information about previous attempts to improvise CPU scheduling, or process scheduling, by using machine learning techniques. This paper attempts to put forth the process scheduling from the resources' point of view, in contrast to CPU point of view otherwise. One of the most important and critical bit of information present in the paper is that higher number of context switches do not indicate improved user experience (or even CPU performance for that matter). Instead, constant context switches may actually increase indivisual processes waiting time. Every context switch is associated with an additional overhead which consumes some of the CPU time for each instance. This results into loss of valuable processor time slices.

The authors did an extensive research and gathered some vital information that will be required for the machine learning algorithms to process. Data was gathered in terms of attributes. Which attributes relate to which process – that gives all the details of the process, were found out. The process id, also known as PID, does not give complete information about the process and it's process cycle when it is under execution inside CPU, was also found during this exercise. The processes were divided into 2 sub-categories 'interactive processes' and 'non interactive processes'. After applying different ML techniques like Trees, Lazy, Rules, etc. and verifying by using different search methods such as Genetic Search, Best First Search and Rank Search, best attributes were found out for tracking the process inside the CPU. "Input Size" and "Page Reclaims" came out to be the best attributes among the 24 of the selected ones. An accurate prediction rate of 91.4% - 99.7% was achieved by these 2 attributes. Another research paper [8], entitled 'Automatic Classification of Processes in Operating Systems', discusses a similar attempt to classify different processes into groups where each group has processes having similar behaviour. Machine learning techniques like Deep Learning and Deep Mining were used for classification.

After finding the best attributes to track their process cycle in CPU, the processes were divided into 3 groups – Batch, Daemon and Interactive. Model was allowed to form groups by unsupervised learning algorithms and was manually analysed for verification. Manual check showed processes could be further divided into 6 groups (rather than 3): A (interactive applications), D (daemons), F (desktop features), N (network), C (text commands) and K. (kernel threads) Processes which do not fit into any of the above listed groups were put into O (other) category.

The authors propose another scheduling method known as 'cognitive scheduling' which calculated the usefulness of a process by using a parameter 'utility value'. The method aimed at grouping the processes under different types of application types. Example, processes which request location, send the location to server, display relevant content, etc. will be grouped under some 'mapping' application; processes which calculate RGB count of every pixel, record them frame by frame into some excel sheet, perform some calculations based on that, etc. will be grouped under 'image processing' application; and many such cases. This grouping is done on the basis of the value returned by the 'utility value' parameter. Value ranges

will be preset and as and how the values are received, processes are grouped after checking the upper and lower limits of the particular range.

A similar approach was presented in the paper which could incorporate current machine learning techniques for improvising process scheduling. Variable time slices were used to allocate CPU time to processes, reducing the number of context-switches which further consume some time for their own. Single integer field, special_time_slice (STS) was used to identify optimal CPU cycles per process. A restriction condition was imposed of minimizing the effective turnaround time for the processes. With 50 processes used for testing the model, improvement of 1.4%-5.8% was observed, and it was predicted that the numbers will be higher with increase in number of processes to be computed.

V. Research Paper 3: A Dedicated Small Computer for Artificial Intelligence

This paper [1] tries to provide a decent solution to the problems mentioned in the beginning of the text. The authors put forward a small computer, named SLIM (Small Lisp based Machine), whose only job is to process ML algorithms and store the result. Further analysis and computation will be done on another computer. Therefore, there is theoretically no lag for any ML process as such. Only extra time that might be required is the waiting time for the previous process to finish execution.

Since this computer is dedicated only for Artificial Intelligence related tasks, the different languages that the computer will deal with are called AI languages. Some of them date back to 1980s where the term AI was starting to get evolved. Languages like Prolog or Lisp (Common Lisp, ETALisp, etc) are the most popular and widely used languages for AI programming in early days of artificial intelligence. SLIM was built to support these languages exclusively. The following were some of the features:

1. It supported all variants of Lisp and Prolog. Modules written in core C and GP C were also supported.

2. Open OS was built into SLIM. This gave pathway to open source project testing and applications to be checked for compatibility and performance recordings.

3. Language libraries were not stored into RAM completely when in use. But partial importing was applied and only those files which were in use where taken on RAM, rest was stored in ROM as usual.

These features were some of the most advanced techniques back then, difficult to implement and especially for newly discovered domains like AI.

There were 2 versions of SLIM released. The first one was single-user and single-language system. ETALisp was the language that was supported. Common Lisp, Prolog (IF/Prolog), KCL (Kyoto Common Lisp) and C language libraries were supported in the second version of SLIM. As far as hardware is concerned, SLIM had plasma display, an MC68020 processor with clock speed of 20MHz, onboard memory of 8MB and 2MB of ROM. Another subsidiary processor MC68000 with clock speed of 10Mhz was dedicated to handle only I/O operations. The MC68020 was dedicated only towards processing AI algorithms, because of which the SLIM became a single-user multiprocessor system. It was a 6-layer system with dual port (also known as dual channel) memory being the only link between the 3-layer groups. The main processor, MC68020, and the subsidiary processor, MC68000, had a common interval timer to keep these 2 processors in synchronization with the memory; obviously to avoid memory wastage. The I/O operations were flagged as *lightweight* processes and the data processing operations were flagged as *heavyweight* operations. Main processor, one of the channels of the dual port memory and complete ROM was dedicated for heavyweight operations, while the remaining resources were open to be used for lightweight operations.

Another unique feature of SLIM was that it supported MS-DOS operating system. MS-DOS was chosen because no other operating system had as many powerful file management systems as it. With additional memory and a more powerful subsidiary processor, support for MS-DOS gave SLIM an additional functionality to locally host a file server. Since floppy disks were used for ROM, virtualization was limited to few kilobytes of data.

Thus, introduction of a computer like SLIM which is dedicated solely towards processing of AI and ML algorithms paved pathway to ideas like running a remote server dedicated only towards AI, and this server would be lent to users so that there would be no need to upgrade their systems to run these algorithms; and many more.

References

[1] Hiromitsu Hirakawa, Hitoshi Ogawa, Masayuki Fujiwara, A Dedicated Small Computer for Artificial Intelligence, Ritsumeikan University, Department of Computer Science and Systems Engineering

[2] Siddharth Dias, Sidharth Naik, Sreepraneeth K, Sumedha Raman, Namratha M, A Machine Learning Approach for Improving Process Scheduling: A Survey, International Journal of Computer Trends and Technology (IJCTT) – Volume 43 Number 1 – January 2017

[3] Atul Negi, Kishore Kumar P, Applying Machine Learning Techniques to improve Linux Process Scheduling, Department of Computer and Information Sciences University of Hyderabad

[4] Naila Aslam, Nadeem Sarwar, Amna Batool, Designing a Model for improving CPU Scheduling by using Machine Learning, International Journal of Computer Science and Information Security (IJCSIS), Vol. 14, No. 10, October 2016

[5] Yiying Zhang, Yutong Huang, 'Learned' Operating Systems, Purdue University

[6] Terek Helmy, Sadam Al-Azani, Omar Bin-Obaidellah, A Machine Learning-Based Approach to Estimate the CPU-Burst Time for Processes in the Computational Grids, 2015 Third International Conference on Artificial Intelligence, Modelling and Simulation

[7] Charles Leech, University of Southampton, UK, Charan Kumar and Amit Acharya, IIT Hyderabad, India, Shen Yang, Geoff V. Merit and Bashir Al-Hashimi, University of Southampton, UK, Runtime Performance and Power Optimization of Parallel Disparity Estimation on Many-Core Platforms, ACM Trans. Embed. Comput. Syst. 17, 2, Article 41 (November 2017)

[8] Araujo, Priscila Vriesman, Carlos Alberto Maziero, and Júlio César Nievola. "Classificação Automática de Processosem Sistemas Operacionais. Diss. Dissertação de mestrado", 74 p. Pós-GraduaçãoemInformática, PontifíciaUniversidadeCatólica do Paraná, Curitiba, 2011.