Digital Innovations & Contemporary Research in SCIENCE, ENGINEERING & TECHNOLOGY

Vol. 7. No. 4, December, 2019

Article Citation Format
Habila, M., Agaji, I & Blamah, N.V. (2019)
An Intelligent Scheduling Framework in a Factored Operating System Using Deep Machine Learning.
Journal of Digital Innovations & Contemp Res. In Sc., Eng & Tech. Vol. 7, No. 3. Pp 1-14

Article Progress Time Stamps

Article Type: Research Article
Manuscript Received: 14th November, 2019
Review Type: Blind Final
Acceptance: 18th December, 2019
Article DOI: dx.doi.org/10.22624/AIMS/DIGITAL/V7N4P1

An Intelligent Scheduling Framework in a Factored Operating System Using Deep Machine Learning

[1]Habila Mikailu & Agaji Iorshase
[1]Department of Mathematics/Statistics/Computer Science
University of Agriculture
Makurdi, Benue State, Nigeria

[2]Blamah Nachamada V.
[2]Department of Computer Science
University of Jos
Jos, Plateau State, Nigeria

Correspondence Author: Agaji Iorshase
Email: ior.agaji@uam.edu.ng & sasemiks@gmail.com

ABSTRACT

Scheduling has become a complex problem with the advancement in hardware technology. This work designed and implemented an intelligent scheduling framework model in a heterogeneous many-core environment for a factored operating system. The framework comprises of a process model, a scheduling model, and a resource performance model. A Deep Machine Learning Algorithm was used to predict process sizes with respect to resource requirements as well as resource performance that evaluated and determined resource capacity, and efficiently distribute these resources to various operating system services. An intelligent scheduling framework that efficiently schedule processes to processors based on resource requirements and respective processor capacities in factored operating system was developed. The framework was simulated using Java 2 Enterprise Edition. Experimental results of the framework indicated a scalable, adaptable, better, faster and efficient scheduling of processes to an increased multiple resource cores with an enhanced system throughput.

Keywords: Factored operating system, processor burst, space sharing, multi-agent, intelligent scheduling, deep machine learning, uniprocessor, multicore system.

1. INTRODUCTION

Scheduling is an activity that handles the removal of running process from the CPU and the selection of another process for execution on the basis of a particular scheme. Schedulers are special system software that carries out the activity of process scheduling in various ways. Their primary job is to select jobs into the system and to decide which process is to execute (Kumari, 2015).

Scheduling problems in operating system comes with a lot of complexity and has attracted research attention over the years. As such, many scheduling methods have been presented, which were used to overcome many different problems that arise in scheduling. Scheduling problems have no known polynomial time algorithms, and are therefore categorized as NP-hard complexity problem. With respect to this reason, several individuals and group of researchers have applied several scheduling techniques, among which are the operation research, multi-agent, the artificial intelligence, expert systems, object-oriented and occasionally, a set of two or more of these techniques to develop scheduling facilities (Ezugwu, 2015). The multicore technology and cloud computing technology are the two categories of computational hardware developments that are capable of enhancing computing capability to its users. Thus, to adequately ascertain these computing capacity, a novel operating system with a new scheduling plan is worthy for these new hardware platforms (Nafaa, 2015).

Single microprocessors will soon contain hundreds to thousands of processor cores with current advancement in today's hardware technology. Thus, the manner in which an operating system manages thousands of processors will be fundamentally different from the way in it manages one or two processors. Consequently, the present-day operating system architecture according to Wentzlaff and Agarwal (2016) should be rethought in order to benefit from benefit from the multicore technology. It is predicted that the advancement of manufacturing technology, the progressing development agreeing to Moore's law, will afford computer chips with 100 to 1000 processor cores on a single piece of silicon chip within few years to come (Borkar, 2007). Various researches have shown that the conventional OS kernel does not scale well beyond eight cores, in spite of the fact that these routines have already been extensively optimized using fine-grained locks. It has also been found that locks were used to a considerable degree, which tends to create many opportunities for lock-related failures that are capable to bring down the system (Fedorovaet al., 2007).

Because the near future manycore processors will consist of thousands of heterogeneous cores, and the existing systems were not designed to be scalability and to handle the heterogeneity of manycore hardware, is what necessitated this research. The need for an intelligent scheduling architectural framework that does not support locks and shared memory abstraction is proposed using multi-agent method technology. In this research, an intelligent scheduling framework based on: scheduling FOS processes on space sharing approach on heterogeneous multi-core processors, to facilitate scalability and manage heterogeneity, through a process of determining the performance of individual processor cores, selects the most optimal processor core for an application process is proposed. This is achieved using multi-agent system technology where agents collaborate to discover available processor cores, and optimally schedule these resources to requesting processes.

2. LITERATURE REVIEW

FOS is a new operating system for cloud computing and manycore system with scalability as the main design constraint, where space sharing replaces traditional time sharing to increase scalability. Wentzlaffet al. (2011) described factored operating system as an operating system for thousands of processor cores, and that processes execute on their separate processor core from OS services. The designers of FOS made an assumption that processor cores will be so abundant in the future whereby devoting a core to a single process is rational without leading into limitations placed by core count. Servers gain typical operating system functionalities and run on committed system cores. This method is inspired by online services, and so they are organized in fleets of cores offering the same functionality to the system. According to Krzyzanowski (2015), the common approach to predict or estimate the size of the next processor burst is by using a time-decayed exponential average of previous processor core bursts for the process.

A formula for estimating the next processor burst period was presented as:

$$s_{n+1} = aT_n + (1 - a)s_n \qquad (1)$$

Where T_n is the measured time of the nth burst; s_n is the predicted size of the nth processor burst; and a is a weighing factor, $0 \leq a \leq 1$.

Nakajima and Pallipadi (2002) implemented a supplementary functionality for a scheduler called Micro-Architectural Scheduling Assist (MASA) on Linux (2.4.18) primarily as a user program, rather than in the scheduler itself. MASA used the information from the hardware monitoring counters to measure the load of process packages and processes. In the prototype of MASA they implemented, the result showed that it improved performance by 6% for the workloads from SPECCPU2000, especially for floating-point intensive case. However, the effectiveness of MASA running on various applications was not evaluated.

Zhuravlevet al. (2012) conducted a survey focused on the subset of solutions that exclusively made use of operating system thread-level scheduling to achieve its goals. Their survey shows that operating system scheduler has expanded well beyond its original role of time-multiplexing threads on a single core into a complex and effective resource manager. They viewed the ideal scheduler in future systems to be a: scheduler that balances system resources among application containers while satisfying applications' individual performance goals and at the same time considering global system constraints such as thermal envelope and power budget.

Schartl (2016) identified three design challenges of operating systems for multi-core architectures, they are: locks which do not scale, poor locality offered by the traditional approach of sharing processor cores between application and operating system, and no more cache coherent shared memory available to the operating system. The work discussed the impact of these challenges to scalability, and proposed locks avoidance to counter scalability threat. The work also proposed splitting up of application and operating system so that they do not need to share cores with each other anymore, rather, each core will be dedicated to every thread on the system. Furthermore, instead of cache coherent shared memory, the work proposed the use of message passing for communication. However, it is still not clear how to design an operating system with locks that proffers satisfactory structural scalability and load scalability as well, hence, it is a challenge in view of the technological advancement of multicourse.

Martorellet al. (1999) described dynamic Space Sharing (DSS) as a two–level scheduling policy, the high level and low level space share policies. In the work, high level space shares the machine resources among the applications running in the system, while the low level improves the memory performance of each program by enforcing the affinity of kernel threads to specific physical processors. DSS distributed processors as evenly as possible among applications taking into account the full workload of the system and the number of processors requested by each application. Each application received a number of processors which was proportional to its request and inversely proportional to the total workload of the system, expressed as the sum of processor requests of all jobs in the system.

Vajda (2017) introduced a novel process scheduling method for chip multi-processors with dozens but scalability spanning to thousands of cores. The method relied on few rules which included a space-sharing idea for scheduling processes on a manycore chip adapting performance of separate cores and the chip as a whole by a controlled and scheduled variation of the frequency at which distinct cores execute, reliance on application-generated requests for computing resources instead of thread assignment policies in the operating system.

The method assumed that each application has a certain amount of cores exclusively allocated to it and that only a single core can actively run an application code, as the sequential portion of the application is executing. However, issues that need to be looked into are the access latency and the speed to memory and buses. Modzelewski, et al. (2009) proposed space multiplexing replacing time multiplexing as a replacement of time multiplexing. In their method, operating system ran on distinct cores from applications, and spatially partitioned working sets. This was due to the emergence of the multi-core technology. In their design, spatial multiplexing was taken further as FOS is factored into function specific services, with each distributed into a fleet of cooperating servers. These servers collaborated to provide a service, and each server is bound to a particular processor core and communicates with other servers in the same service fleet via messaging.

Their design schedule entailed that whenever an application needs to access a service provided by the operating system, the application messages the closest core providing the desired service, found by querying the name server. Although this method may achieve minimal communication cost, however, it may not achieve optimal system throughput, due to the fact that the closest server to the application process may not match the processing capacity in respect to the requesting process.

Asmussen et al. (2016) integrated arbitrary cores as first and second class citizens into the system, leveraging on the idea that cores are abundantly available. They integrated cores and memories into a packet-switched network-on-chip (NoC) and equipped each core with a data transfer unit (DTU) as the common hardware component. The only means for the core to communicate with other cores or memories was through the DTU, offering message passing and memory access. Controlling the DTU allowed the control of core and therefore also the software running on the core. OS services like file systems and network stacks were provided based on a core-neutral communication protocol between DTUs.

They also introduced network-on-chip-level isolation, presented the design of microkernel-based OS, M3, and the common hardware interface, and evaluated the performance of the prototype in comparison to Linux. The result showed that without using accelerators, M3 outmatches Linux in some application-level benchmarks by more than a factor of five. However, this design decreases system utilization as processing element (PE) was idled for a certain time, waiting for an incoming message or the completion of a memory transfer.

Kvalnes et al. (2015) presented the omni-kernel architecture and its Vortex implementation whose goal was to ensure that all resource consumption is measured, that the resource consumption resulting from a scheduling decision is attributable to an activity, and that scheduling decisions are fine-grained. They used the method of factoring the operating system into a set of resources that exchanged messages to cooperate and provide higher-level abstractions to achieve their goal, with schedulers inter-positioned to control when messages are delivered to destination resources.

Results from their experiments showed all resource consumption was accurately measured and attributed to the correct activity, and schedulers were able to control resource allocation. Using a metric that concisely reflects the main difference between an omni-kernel and a conventionally structured operating system, the fraction of CPU consumption that can be attributed to anything but message processing, they determined omni-kernel scheduling overhead to be below 6 percent of CPU consumption or substantially less for the Apache, MySQL, and Hadoop applications.

Reuther et al. (2017) presented an elaborate feature analysis of fifteen supercomputers with big data schedulers. The theoretical scheduling latency model was the essential performance of scheduler's feature that was built and employed for the designing of experiments targeting the measurement of scheduling latency. A detailed benchmark of Slurm, Son of Grid Engine, Mesos, and Hadoop YARN were the four popular schedulers that were conducted. The theoretical model was compared with data schedulers and was shown that the schedulers' performance could be characterized by two key attributes: the marginal latency of the scheduler and the nonlinear exponent. For all these four schedulers, the utilization of the computing system was decreased to less than 10 percent for computations that lasted for only few seconds.

Wentzlaffet al. (2011) demonstrated an experiment in respect to the impact of proper spatial scheduling on performance, in multi-core processor systems. The experiment uses the read-only file system fleet with 'good' and 'bad' layouts, and was executed on a 16-core Intel Xeon E7340. This machine had a very small intra-socket communication cost applying user-space messaging, which revealed significant communication heterogeneity between intra-socket and inter-socket communication. The result of their experiment showed that good layout is uniformly better than the bad layout in performance.

Also, they maintained that future multi-cores will feature much greater heterogeneity, and commensurately higher end-to-end performance inequality from spatial scheduling.Currently research is focused on spatial scheduling in multi-core technology, but little on intelligent scheduling in factored operating system environment. This research work therefore, proposes an intelligent scheduling framework in a factored operating system that will make decisions based on performance metrics, on how to allocate processor cores in a system, thereby enabling fleet of servers (operating system services) to either grow or shrink to meet their allocation.

3. METHODOLOGY

The research methodology adopted in this work is the Multi-Agent Oriented Software Engineering Development Method. The aim is to develop a multi-agent scheduling system by using multi-agent technological concepts. The architecture of the proposed framework is as depicted in figure 1. In figure 1 the architecture is made up of a process sub-model, a scheduler sub-model, a performance sub-model and resource sub-model as a group of autonomous intelligent agents. Information about processes is retrieved by the process agent. The process agent in turn predicts the process' size using deep machine learning algorithm (ANN) with respect to the degree of resource requirements. From the result of this prediction, the process agent requests the most suitable scheduler for its process. The resource agent extracts resource core information, computes the capacities of resource cores and distributes these resources to the various operating services forming fleets of servers.

The selected scheduler agent in collaboration with other schedulers allocates process with high degree of resource requirement to a scheduler of high capacity and vice-versa, in an event of several processes with different degree of resource requirements are requesting for resource services (servers) at a time. The scheduler agent will always allocate available resource with higher capacity to the requesting processes. The repository agent controls the knowledge representation (database) of the performance sub- model, where information about the objects is stored for scheduling purposes. It also consistently updates the object information in tune with changing environment.

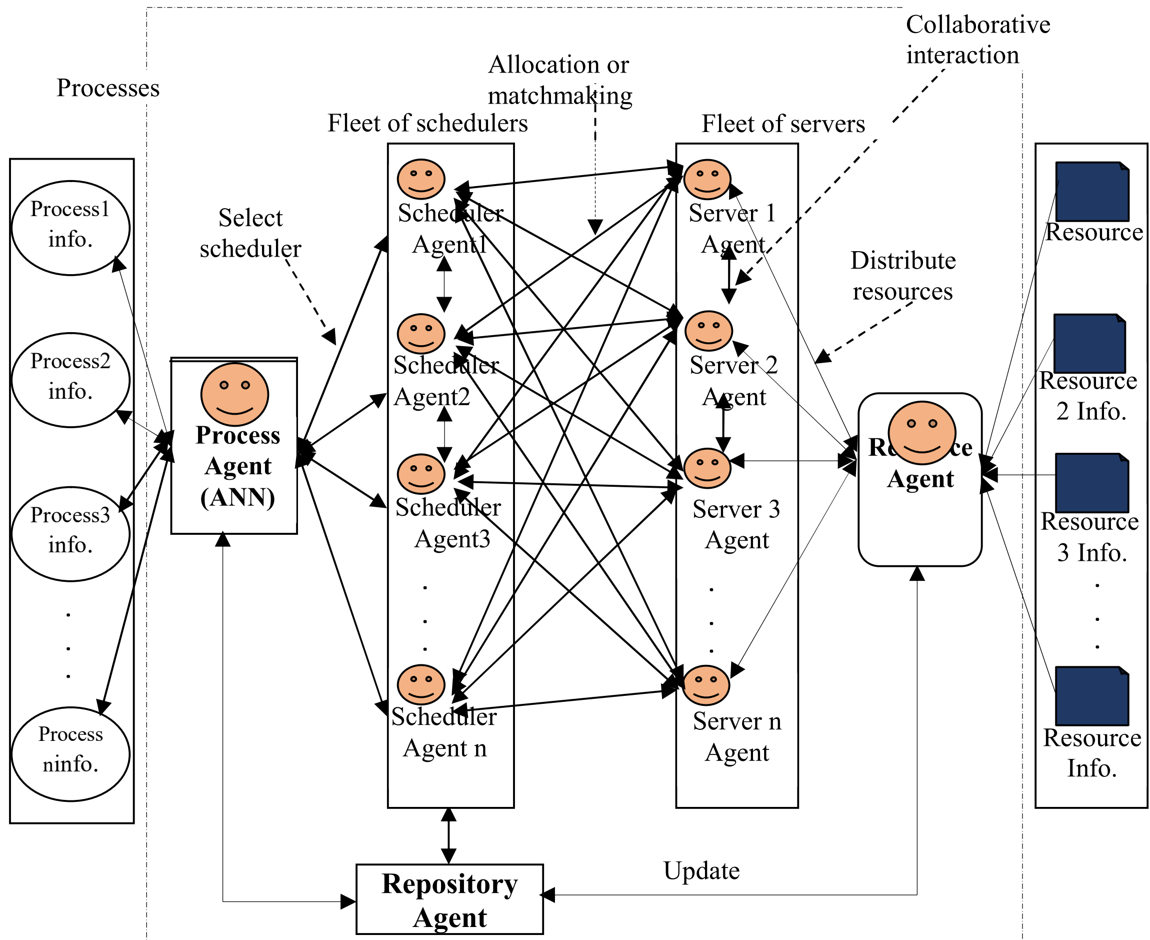

Fig. 1: The Architecture of the Proposed System

A deep machine learning using back propagation algorithm was used for process' size predictions. Figure 2 depicts the back propagation model of the proposed system. It shows an explicit model of a full scale network with $\{p_1, p_2\}$ as input parameters. Also, two hidden layers with three and two numbers of neurons respectively, as well as one output neuron are shown. Parameter P_1 is associated with weight W_{11}, W_{12} and W_{13}. Where W_{11}, W_{12} and W_{13} represent input parameter 1 weight 1, input parameter 1 weight 2 and input parameter 1 weight 3 respectively. These weights were random numbers generated between 0 and1. Each neuron in the hidden layer is represented by 'a' which is the tanh activation function in equation 4. Where b1, b2 and b3 are the bias values for the hidden layer 1, hidden layer 2 and the output layer y respectively.

The deep neural network equations used by Courtial (2017) is adopted in this research work. The input data to the neurons is normalized in order to get the input values to a given range using Equation 2.

$$p = \frac{(p-min1)(p-min2)}{(max1-min1)} + min2 \qquad (2)$$

Where p is the input parameter of the processes, min and max are the minimum and maximum values of the parameter respectively.

On the other hand, the output is scaled within the range of 0 and 1 using equation 3.

$$y = \frac{p - minp}{maxp - minp} \quad (3)$$

Where y is the output value of the network

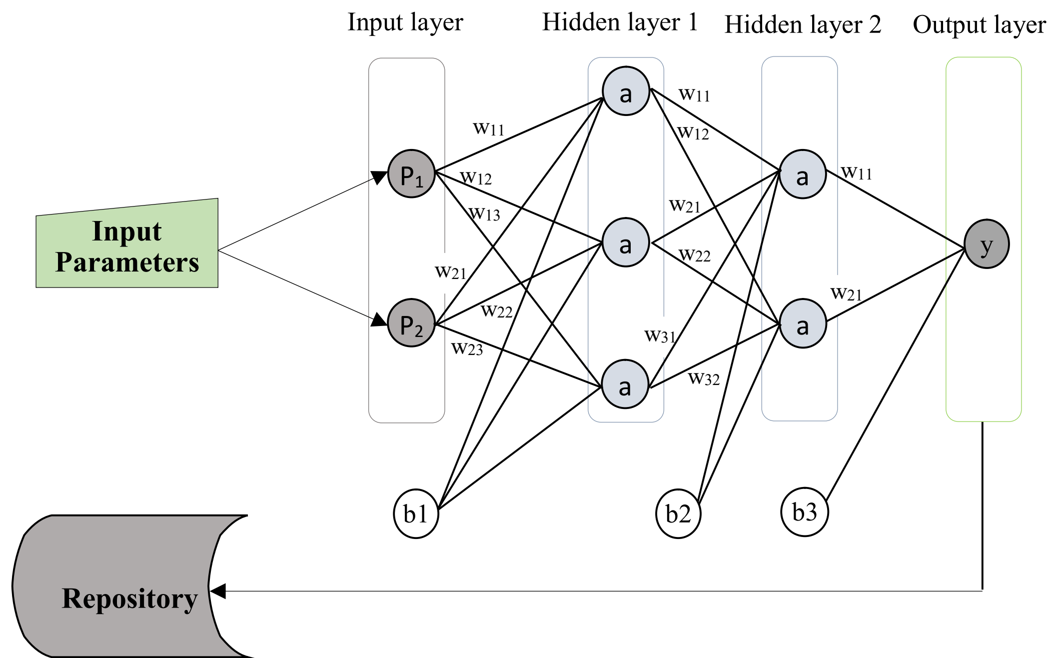

Figure 2: Deep Machine Learning Sub-Model

Equation 4 is the Tan Activation Function used in the forward pass training of the network to segregate the relevant information.

$$\tanh(p) = \frac{e^{p} - e^{-p}}{e^{p} + e^{-p}} \quad (4)$$

Equation 5 is the Mean Square Error (MSE) used to compute the difference between the estimated and what is estimated in training the network in order to improve prediction accuracy.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\bar{y}i - yi)^2 \quad (5)$$

$\bar{y}i$ is the predicted value, and yi is the targeted value of the deep learning network.

A partial differential equation to the cost function (square error) of equation 5 is used in the backward propagation algorithm network to measure the distance between the predicted value and the actual value.

Cost Function (CF) = $j(w) = \frac{1}{2}\sum_{i=1}^{n}(\bar{y}i - yi)^2$ (6)

Equation 7 is used in updating the weights in the Back Propagation Algorithm using a gradient decent:

$Wl = W - \alpha\frac{1}{n}\frac{\partial j(w)}{\partial wl}$ (7)

Where w_l is the new weight, w is the weight, α is the learning rate, and j is the cost function.

The back propagation algorithm was employed as the basis for the learning rule.

3.1 Database Design

Figure 3 depicts the entity relationship diagram used by the framework. The logical structure of the database shows the various entities of the system framework. These entities include: Traincase, normalize, weight, servers, schedulers, jobs, cores, executiondetails and executionsummary. Each of these entities contains attributes with one primary key (PK) that uniquely identifies the various entities, and the foreign key (FK) as well. The relationships between these entities are basically one to many, and many to many relationships. The relationship between Traincase, and the Normalize for instance, is many to one relationship, Jobs and Schedulers many to one, Jobs to Servers many to one, many to many relationship between executionsummary and executiondetails, and so on. This entity relationship diagram of figure 3 gives the logical structure of the new system design framework database.

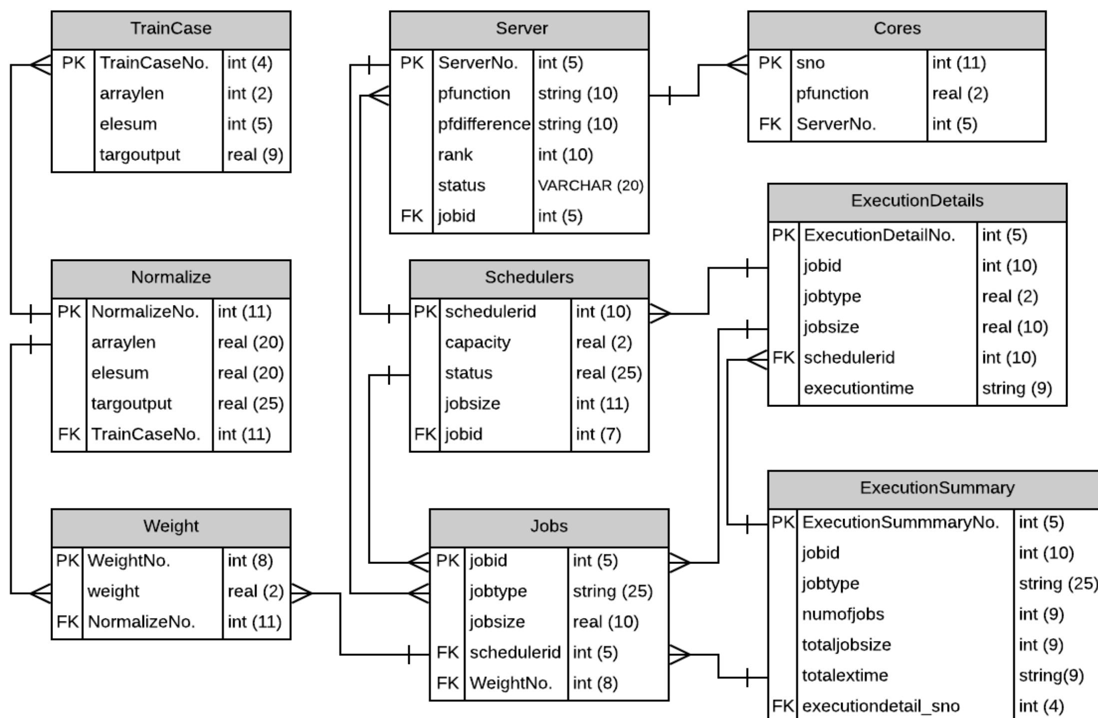

Figure 3: The Entity Relationship Diagram

3.2 Detailed Design

A UML tool, sequence diagram, was used for detailed design. The sequence diagram of the proposed framework is as shown in figure 4. In figure 4, a user initiates the execution activity by lunching an application. Processes are submitted to the activation queue along with their parameters .The process agent uses the processes' parameters to predict or estimates the resource requirement (size) of each process. The resource agent will compute the capacity or performance function of each resource and classified the resources to various operating system servers.The process agent having determined the resource requirements of each process, now selects schedulers for processes.

The scheduler agents schedule processes based on the resource needed and the capacities of the server (processor cores). The scheduler agent in collaboration with the resource agent allocates the resource (processor core) to the processes. The outlined scheduling procedure set is carried out by group of agents, harmonizing the activities of execution that will yield an expected scheduling result.

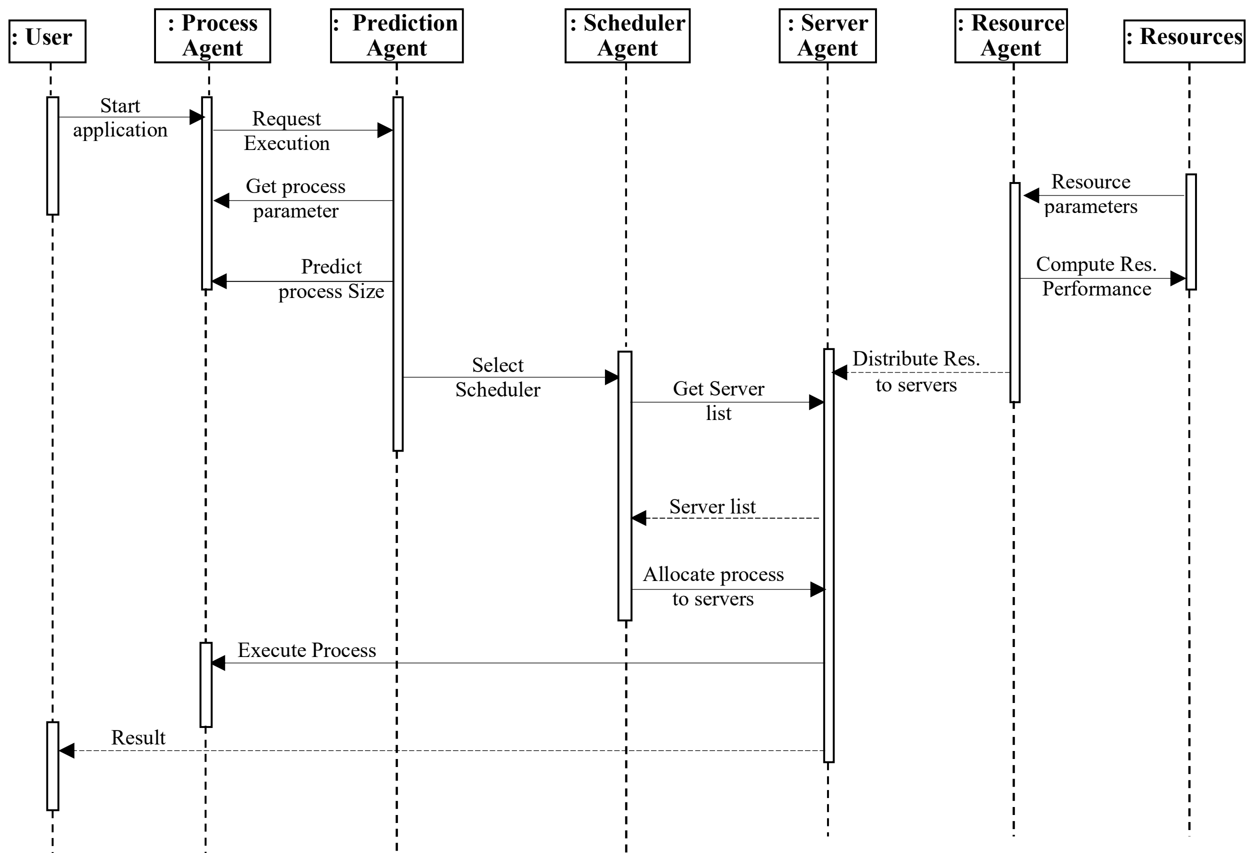

Figure 4: Sequence Diagram of the Proposed System

4. EXPERIMENTAL RESULTS

In simulating the framework Netbeans IDE, MySQL Query Browser, JBOSS 4.2.2.GA Server, and a macromedia Dreamweaver 8.0were used. Also, integer arrays were used to represent application processes. The parameters of the array used were the length of the array and the sum of the elements of the arrays. The maximum length of the array was set to 20; random values between 1to10 were used to represent array elements. The network was trained using 200 random data sets with input fields containing the normalized values of process' parameters (Array Length, and Array Element Sum), and random numbers generated between -1 and 1 that that served as input weights. The input page for the framework with sample input entries is as shown in figure 5. In figure 5 PERFORM SCHEDULING is a command button.

Figure 5: The Scheduling Data Input Page

In figure 5 has a field for Number of Jobs, Fix Job Size and Active Scheduler cores. It also has a PERFORM SCHEDULING command which is used in carrying out the scheduling activities.

Eleven different data sets each made up of job size and number of jobs requesting for system resources were used in the experiments.

Experiment 1: Framework Performance in carrying out Processes Scheduling

Experiment 1 tests the system performance throughput in terms of scheduling of processes or threads. The inputs to the framework are the number of job sets, job size and number of schedulers. Job types were randomly assigned to the job sets. Eleven different job sets with different job sizes and job types were used for this experiment. The result of the experiment was as seen in column 7 of table 1. The result of table 1 showed that serial number 1, one (1) job set came in with job size of 20KB, the scheduling time was observed to be 0.02sec using 1 scheduler of 40GHz. Also in serial number 10, one thousand (1000) job sets came in each with a job size of 20KB, the scheduling time was observed to be 0.23sec using 1000 schedulers of 24476.0GHz.

Table 1: The Results of Experiment1

EXECUTION RESULTS						
S/NO	**Job Type**	**No. of Jobs**	**Job Size(kb)**	**Schedulers**	**Capacity(PF)**	**Scheduling Time (sec)**
1	File Management	1	20	1	40.0	0.02
2	Data Synchro	2	40	2	80.0	0.02
3	Data Synchro	4	80	4	160.0	0.02
4	PC Security Management	8	160	8	320.0	0.02
5	PC Security Management	10	200	10	400.0	0.02
6	Data Synchro	50	1000	50	1981.0	0.02
7	System File Naming	100	2000	100	3877.0	0.02
8	PC Security Management	300	6000	300	10672.0	0.02
9	System File Naming	500	10000	500	16153.0	0.02
10	PC Security Management	1000	20000	1000	24476.0	0.23
11	System File Naming	1250	25000	1250	25523.0	2

Experiment 2: Scalability and Adaptability of Framework

This experiment was carried out to test the scalability and the adaptability of the framework in respect to the scheduling time with increase in the numbers of schedulers. The inputs to the system were number of job set and job sizes. Eleven job sets were used with the same number of jobs and sizes. The numbers of schedulers were increased accordingly to the fixed number of jobs sizes in order to test how the framework scale and adapt to the increase of resources. The results of the experiments were as summarized in table 2. Table 2 indicated that the job set at serial number 1 had 1000jobs each with 20KB of size, were scheduled within a time frame of 49970sec with one (1) scheduler of 40.0GHz. Also, a serial number 9 showed that 1000 job sets were scheduled within a time frame of 0.68sec with three hundred (300) schedulers of 10666.0GHz. Furthermore, a serial number 10 of 1000 job set with the same size of 20KB each, were schedule within 0.14sec with 500 schedulers of 16158.0GHz.

Table 2: Results of Experiment 2

EXECUTION RESULTS						
S/NO	**Job Type**	**No. of Jobs**	**Job Size(kb)**	**Schedulers**	**Capacity(PF)**	**Scheduling Time (sec)**
1	Data Synchro	1000	20000	1	40.0	49970
2	System File Naming	1000	20000	2	80.0	12485.01
3	System File Naming	1000	20000	4	160.0	3117.5
4	Data Synchro	1000	20000	8	320.0	777.51
5	Data Synchro	1000	20000	10	400.0	497.01
6	PC Security Management	1000	20000	50	1982.0	19.4
7	System File Naming	1000	20000	100	3877.0	4.7
8	File Management	1000	20000	1	40.0	0.34
9	Data Synchro	1000	20000	300	10666.0	0.68
10	PC Security Management	1000	20000	500	16158.0	0.14
11	System File Naming	1000	20000	1000	24468.0	0.23

5. RECOMMENDATION

The proposed framework is recommended for use because of its capability to adequately scale and adapt to changing environments, and also its ability that is seen in utilizing the benefits that comes with the advancement in multicore hardware technology as system throughput is enhanced. In future work, we will introduce process job transfer between the processor cores, that is when a higher capacity core has finished execution and is idle, then it can take over from a lower capacity core, this is believed will further increase the system performance throughput.

REFERENCE

1. Air Force Research Laboratory (2012).FOS: A Factored Operating System for High Assurance and Scalability on Multicores. Massachusetts Institute of Technology, Rome, NY 13441.
2. Asmussen, N., Volp, M., Nothen, B., Hartig, H., Fettweis, G., Operating-Systems Chair and Vodafone Chair Mobile Communications Systems (2016). M3: A Hardware/Operating- System Co-Design to Tame Heterogeneous Manycores. Retrieved from https://os.inf.tu- dresden.de/papers_ps/asmussen-m3-asplos16.pdf
3. Courtial, F. (2017). Deep Neural Networks from Scratch. Retrieved on 7th July, 2019, from https://matrices.io/deep-neural-network-from-scratch/
4. Borkar, S. (2007).Thousand Core Chips – A Technology Perspective. In Proceedings of DAC.Retrieved on 18th April, 2018, from http://impact.asu.edu/cse520fa08/ThousandCore.pdf
5. Chapin, J. S. and Weissman, B. J. (2004). Distributed and Multiprocessor Scheduling. Retrieved on 21st October, 2018 from, https://www-users.cs.umn.edu/~weiss039/papers/handbook.Pdf
6. Ezugwu, A. E. (2015). Architectural Pattern for Scheduling Multi-Component Applications in Distributed Systems. Ph.D. Thesis. Department of Mathematics, Ahmadu Bello University, Zaria, Nigeria. 374pp
7. Fedorova, A., Kumar, V., Kazempour, V., Ray S. and Alagheband, P. (2007). Cypress: A Scheduling Infrastructure for a Many-Core Hypervisor. School of Computing Science, Simon Fraser University, Vancouver, Canada. Retrieved on 18th April, 2018, from https://pdfs.semanticscholar.org/7013/915198103d9ec28c2260ce7fe99babd78263.pdf
8. Jain, S. and Jain, S. (2016). A Review Study on the CPU Scheduling Algorithms. International Journal of Advanced Research in Computer and Communication Engineering, 5(8): 22- 31. Retrieved on 11th October, 2018, from https://www.ijarcce.com/upload/2016/august- 16/IJARCCE%205.pdf.
9. Kepner, J., Brightwell, R., Edelman, A., Gadepally, V. Jananthan, H., Jones, M., Madden, S., Michaleas, P., Okhravi, H., Pedretti, K., Reuther, A., Sterling, T. and Stonebraker, M. (2018). TabulaROSA: Tabular Operating System Architecture for Massively Parallel Heterogeneous Compute Engines. Retrieved on 29th August, 2019, from https://arxiv.org/pdf/1807.05308.pdf
10. Krzyzanowski, P. (2015). Operating System: Process Scheduling. Retrieved on 14th of September, 2018 from https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html.
11. Kumari, E. (2015). A Review on Task Scheduling Algorithms in Cloud Computing. International Journal of Science, Environment and Technology, 4(2): 433 – 439.
12. Kvalnes, A., Johansen, D., Renesse, R., Schneider, F. B. and Valvag, S. V. (2015). Omni-Kernel: Operating System Architecture for Pervasive Monitoring and Scheduling. IEEE Transactions On Parallel And Distributed Systems, 26(10): 2849-2862.
13. Laplante, P. and Milojicic, D. (2016). Rethinking Operating Systems for Rebooted Computing. IEEE International Conference on Rebooting Computing (ICRC). Retrieved from https://ieeexplore.ieee.org/abstract/document/7738695/authors
14. Modzelewski, K., Miller, J., Belay, A., Beckmann, N., Gruenwald III, C., Wentzlaff, D., Youseff, L. and Agarwal, A. (2009). A Unified Operating System for Clouds and Manycore: fos Retrieved on 26th June, 2018, from https://www.researchgate.net/publication/40000599_A_Unified_Operating_System_for_Clouds_and_Manycore_fos
15. Martorell, X., Corbalan, J., Nikolopoulos, S. D., Navarro, N., Polychronopoulos, E D., Papatheodorou, T. S. and Labarta, J. (1999). A Tool to Schedule Parallel Applications on Multiprocessors: Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems (PDCS'99), Fort Lauderdale (Florida - USA) http://www.cs.huji.ac.il/~feit/parsched/jsspp00/p-00-7.pdf

16. Nafaa, H. (2015). FOS (Factored Operating System): An Operating System for Multi-core and Clouds. Retrieved on 3rd April, 2018, from http://www.slideshare.net/mobile/naffa1/ factored-operating-system-an-operating-system-for-multicore-and-clouds
17. Nakajima, J. and Pallipadi, V. (2002).Enhancements for Hyper-Threading Technology in the Operating System–Seeking the Optimal Scheduling. Workshop on Industrial Experiences with Systems Software, Boston, MA.
18. Reuther, A., Byun, C., Arcand, W., Bestor, D., Bergeron, B., Hubbell, M., Jones, M., Michaleas, P., Prout, A., Rosa, A. and Kepner, J. (2017). Scalable System Scheduling for HPC and Big Data. Retrieved from, https://arxiv.org/pdf/1705.03102.pdf
19. Schartl, A. (2016). Design Challenges of Scalable Operating Systems for Many-Core Architectures. Retrieved on 7th April, 2018, from http://www4.cs.fau.de/Lehre/WS16/PS_KVBK/slides/slides-schaertl.pdf
20. Vajda, A. (2017). Space-Shared and Frequency Scaling Based Task Scheduler for Many-Core OS. Group Function Technology, Ericsson Hirsalantie 11, Jorvas, Finland. Retrieved on 18th April, 2018, from https://www.sigops.org/sosp/sosp09/papers/hotpower_9_vajda.pdf
21. Wang, Y., Li, L., Wu, Y., Yu, J., Yu, Z. and Qian, X. (2019). TPShare: A Time-Space Sharing Scheduling Abstraction for Shared Cloud via Vertical Labels. ISCA ’19: ACM Symposium on Computer Architecture, Phoenix, AZ. ACM, New York, NY, USA. Retrieved on 10th September, 2019, from https://doi.org/10.1145/1122445.1122456.
22. Wentzlaff, D., Agarwal, A. (2016). Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. Retrieved on 6th April, 2018, from http://www.slideshare.net/mkindika/factored-operating-systems
23. Wentzlaff, D., Gruenwald, C., Beckmann, N., Modzelewski, K., Belay, A., Kasture, H., Youseff, L., Miller, J. and Agarwal, A. (2011). Fleets: Scalable services in a factored operation system. Computer Science and Artificial Intelligence Laboratory Technical Report, Massachusetts Institute of Technology, Cambridge, Ma 01239 USA.
24. Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A. and Prieto, M. (2012). Survey of Scheduling Techniques for Addressing Shared Resources in Multi-core Processors.ACM Computing Surveys, 45(1): 4:1-28. Retrieved on 21st October, 2018, from http://www.ece.ubc.ca/~sasha/papers/zhuravlev_csur11.pdf