Priority Based Scheduler for Asymmetric Multi-core Edge Computing

Rupendra Pratap Singh Hada* and Abhishek Srivastava

Indin Institute of Technology Indore, 452020, India
E-mail: phd2101101004@iiti.ac.in; asrivastava@iiti.ac.in
**Corresponding Author*

Received 07 August 2023; Accepted 30 September 2023;
Publication 23 December 2023

Abstract

Edge computing technology has gained popularity due to its ability to process data near the source or collection device, benefiting from low bandwidth utilization and enhanced security. Edge devices are typically equipped with multiple devices that employ asymmetric multi-cores for efficient data processing. To ensure optimal performance, it is crucial to carefully assign tasks to the appropriate cores in asymmetric multi-core processors. However, the current Linux scheduler needs to consider the capabilities of individual cores when assigning tasks. Consequently, high-priority tasks may be assigned to energy-efficient cores, while low-priority tasks end up on high-performance cores. This sub-optimal task assignment negatively impacts the overall system performance. To address this issue, a new algorithm has been proposed. This algorithm considers both the core's capabilities and the task's priority. However, due to the asymmetric nature of the cores, prior knowledge of each core's speed is necessary. The algorithm fetches the priorities of the tasks and classifies them into high, medium, and low-priority categories. High-priority tasks are scheduled on high-performance cores, while medium and

Journal of Web Engineering, Vol. 22_6, 871–888.
doi: 10.13052/jwe1540-9589.2262

low-priority tasks are allocated to energy-efficient cores. The proposed algorithm demonstrates superior performance for high-priority tasks compared to the existing Linux task scheduling algorithm. It significantly improves task scheduling time by up to 16%, thereby enhancing the system's overall efficiency.

Keywords: Asymmetric multi-core processors; completely fair scheduler; edge computing; scheduling.

1 Introduction

As the personal computer era continues to witness a growing number of daily-generated applications, end-users are increasingly demanding faster and more capable systems to accommodate them. Within the realm of computer architecture, two primary approaches exist: single-core and multi-core. In the context of single-core architecture, enhanced performance is achieved by elevating clock speeds, although this approach is constrained by the challenge of managing heat generation. Conversely, multi-core architecture involves integrating multiple processing cores onto a single chip, thereby enabling the simultaneous execution of multiple instructions and consequently accelerating program execution [3]. However, the adoption of multiple cores on a single-chip processor gives rise to several challenges, including issues related to memory, cache coherence, power consumption, and load distribution among the cores [4]. In response to these challenges, asymmetric multi-core architecture has gained traction in the computing landscape, thanks to its ability to offer high performance and energy efficiency. This architecture is increasingly prevalent in various computing devices, ranging from personal computers and cell phones to General Purpose computing on Graphics Processing Units (GPGPU) [5].

Edge computing, a technology on the rise, involves processing data in close proximity to the source or collection device. It encompasses a variety of networks and devices located near the user. Edge computing offers multiple benefits [25], including (i) Improved data privacy and security, (ii) Lower operational costs, (iii) Enhanced network reliability, and (iv) Facilitation of machine learning and artificial intelligence capabilities in close proximity to the devices.

To meet the user's requirements, these devices need to operate efficiently. Among the challenges posed by multi-core architecture in edge computing devices, the assignment of tasks to the appropriate core is also a significant

challenge because the assignment of tasks or task scheduling is an NP problem [7]. The present Linux (Completely fair scheduler) scheduler is designed with a symmetric multi-core processor keeping in mind that the tasks are scheduled without knowing the capability of the core, which degrades the overall performance of the Asymmetric multi-core systems [11].

The suggested approach is versatile and can function efficiently within both symmetric and asymmetric multi-core systems. Initially, the method assesses each core's inherent tendencies towards high performance or energy efficiency. Subsequently, it classifies tasks into three discrete groups based on their designated priorities. Ultimately, the procedure coordinates task scheduling to align with these priorities, assigning high-priority tasks to high-performance cores and directing medium and low-priority tasks towards energy-efficient cores. The proposed method has shown an improvement in speed of up to 16% for high-priority tasks. This paper is the extended version of [14], in which the efficacy of the proposed method is tested on Linux 20.

The remainder of the paper is organised as follows: Section 2 discusses existing techniques for priority based scheduling in Linux and Windows; Section 3 includes the details of the proposed approach; Section 4 includes experimental results that help validate the efficacy of the proposed approach; and finally, Section 5 concludes the paper with pointers to future work.

2 Previous Works

A considerable amount of literature is available on scheduling techniques for the Linux scheduler, all of which aim to improve its scheduling ability. This section will cover some of these efforts.

An optimal algorithm for task scheduling and data block placement in edge computing devices was suggested by the author [21]. The primary aim of the algorithm is to minimize response time and computational delay. When considering data block placement, the algorithm takes into account three key factors: popularity, data storage capacity of the block, and replacement ratios of an edge server. The algorithm utilizes real data for task scheduling.

The paper [19] introduces a novel task scheduling algorithm for edge computing, leveraging IoT devices. The algorithm allows tasks from edge computing devices to be transferred to nearby IoT devices, as long as it doesn't disrupt the normal operation of the device and enhances the throughput of edge services. By considering energy consumption and offloaded execution time, the edge computing devices intelligently determine the optimal IoT device to which their tasks should be shifted.

The article [31] presents a combined algorithm for task scheduling and containerization. The algorithm begins by conducting experiments to assess the resource utilization of container operations. Subsequently, system models are created to accurately capture the execution characteristics of tasks within containers. Leveraging these models, the algorithm proceeds to perform containerization and task scheduling.

In article [24], a scheduling framework designed to address energy efficiency, shared resource contention, and fairness concerns in heterogeneous multi-core processors is introduced. Author put this framework into practice and assessed its performance on an actual HMP (heterogeneous multi-core processor) platform. The experimental results, obtained using the SPEC CPU2006 benchmark, demonstrate that proposed framework outperforms Linux and four other schedulers in terms of both fairness (with an average improvement of 58%) and energy efficiency (with an average improvement of 37%).

The author [23] considered various factors, including equitable resource utilization, which can result in decreased performance predictability, extended makespan, instances of starvation, and quality of service (QoS) degradation. Additionally, the impact of cluster voltage and frequency settings on fairness was explored through Dynamic Voltage and Frequency Scaling (DVFS). The effectiveness of the proposed approach was assessed on an actual heterogeneous multi-core processor, leading to energy efficiency and fairness enhancements.

In **Task Snatching Technique** the high-performance core snatches the tasks from energy-efficient cores; some times high-performance core may snatch a task with less execution time from an energy-efficient core while another slow core has a task with more execution time and results in poor execution time [2].

CAMP: In CAMP scheduling, two tasks, utility factor and scheduling algorithm, are performed. A metric utility factor produces a single value as CAMP to evaluate the application's work. The rhythmic design helps the scheduler choose the best threads to run on fast cores. The scheduler must recognise the most eligible contenders to run on fast cores [22]. Calculations of the applications are done after utility factors are computed. Depending on the individual utility factors, CAMP recognises the threads that could be placed on different types of cores [30]. As we discussed above, CAMP is a thread-level scheduler because tasks in the same task-based programs can often achieve similar speedup ratios on fast cores; it does not facilitate the

performance of a single parallel program. Therefore, CAMP did not consider the scheduling problem in parallel applications [17].

Bias Scheduling: This scheduling dynamically monitors the thread bias to match them with the suitable cores, eventually amplifying the system throughput. Each application gives a bias that supplements its resource needs appropriately in this work [20]. Though bias scheduling can be easily implemented and requires fewer changes in actual code, the problem with this approach is that it requires a bias scheduling matrix. If the information of any task is absent in this matrix, it performs similarly to regular scheduling, resulting in degradation of performance [8].

Speed-Based Balancing: The Speed Balancing algorithm aims to oversee the migration of threads, ensuring that each thread has an equitable opportunity to execute on the swiftest available core. Rather than focusing on distributing workloads evenly, this algorithm strives to equalize the time each thread spends executing on both faster and slower cores [15]. However, this approach can inadvertently lead to task snatching, which, in turn, introduces significant overhead and diminishes overall system performance.

The task scheduling techniques like task snatching [2] snatch tasks from the slow-core with less remaining execution time and increases the overall execution time of the system. CAMP [16] works based on individual utility factor tasks, and then the task is scheduled according to utility factor, but it only works with threads. Therefore, in the proposed work, the task scheduling problem is addressed via CPU affinity (in which any task can be scheduled explicitly on a particular core). The proposed task scheduling algorithm determines the priority of individual tasks and schedules the task based on priority on an appropriate core as per its capability [13].

3 Proposed Work

The remainder of the section is organised as follows: Section 3.1 discusses the problem of scheduling in Linux operating system; Section 3.2 includes the system architecture; and Section 3.3 represents the proposed solution.

3.1 Detailed Problem Statement

In an Asymmetric Multicore Processor (AMP) or Heterogeneous Multicore system, not all cores share identical characteristics. Instead, various cores exhibit distinct architectures and capabilities. These heterogeneous cores may

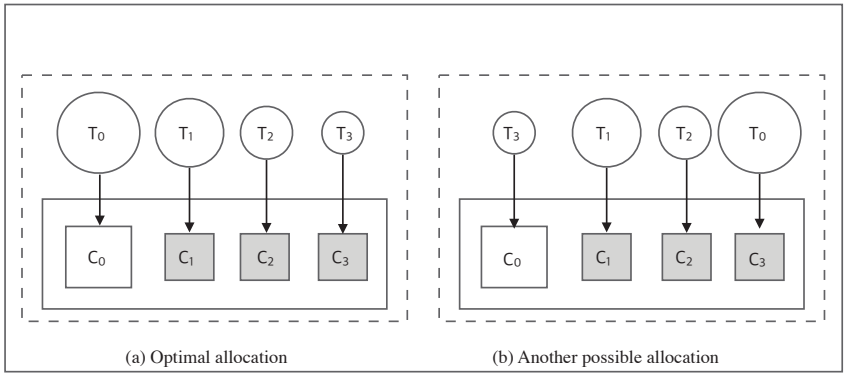


Figure 1 Scheduling in AMP.

differ in terms of factors such as frequency, voltage, and memory model, among others. The crucial feature in asymmetric chip multiprocessing lies in the ability to seamlessly transition between these diverse cores and efficiently power down any unused cores. We are working with an Asymmetric Multi-core Processor (AMP), comprising one high-speed core denoted as C_0 and three slower cores, namely C_1, C_2, and C_3. We have four tasks labeled as T_0, T_1, T_2, and T_3, each assigned a priority level of low, medium, high, and low, respectively, for scheduling across these cores. Let's assume that tasks T_0, T_1, T_2, and T_3 require time intervals of S_0, S_1, S_2, and S_3 for execution on the slower cores, while they require F_0, F_1, F_2, and F_3 time intervals on the fast core, respectively. It is reasonable to infer that for each task, the time required on the fast core is less than the time required on the corresponding slow core, resulting in the relationships $F_0 < S_0$, $F_1 < S_1$, $F_2 < S_2$, and $F_3 < S_3$ [10]. You can find a visual representation of the overall scheduling in AMP in Figure 1 (Figure 1).

If the tasks are allocated according to optimal priority scheduling means allocating the tasks such as high-performance core get high priority tasks, then overall completion time T_{opt} (make-span) can be expressed as,

$$T_{opt} = \mathbf{max}(F_0, S_1, S_2, S_3) = F_0$$

Because $F_0 < S_0$, we can say that $T_{opt} < S_0$.

In another traditional allocation technique, task T_0 is scheduled on slow core C_3, and task T_3 is scheduled on fast core C_0. The make-span for traditional allocation time T_{opt} can be expressed as,

$$T_{trad} = \mathbf{max}(S_0, F_1, S_2, S_3) \geq S_0 > F_0.$$

It's clear that assigning a high-priority task to a slower core could frequently result in a decline in the overall performance of high-priority tasks. Therefore, the objective is to allocate processes in a manner that prioritizes high-performance cores for tasks with high priority, while directing tasks with medium and low priority to lower-performance cores [20]. It's worth noting that finding the optimal allocation of tasks to cores is a computationally challenging problem known as NP-hard. Due to the dynamic nature of runtime behavior, achieving a perfect allocation may not always be feasible, but we can strive to schedule tasks in a near-optimal manner to adapt to changing conditions [12].

3.2 System Architecture of Task Scheduling

The priority-based task scheduling within the Asymmetric Multi-core Processor (AMP) context involves managing multiple tasks denoted as T_1, T_2, and so forth up to T_{n-1}, and T_n, all residing in the ready queue. In this setup, we have a total of *n* cores, ranging from C_0 to C_n, where C_0 to C_k are characterized as high-performance cores, and C_{k+1} to C_n are designated as energy-efficient cores. The primary objective is to establish an effective mapping of tasks that ensures high-priority tasks are scheduled on high-performance cores, while medium and low-priority tasks are dispatched to the low-performance cores. To achieve this task allocation, a mapping function is employed, which classifies tasks into three priority categories: low, medium, and high. Subsequently, based on their respective priorities, tasks are assigned to cores that align with their priority class.

Figure 2 shows the task scheduling process in AMP, in which we have tasks from T_1 to T_n and cores from C_0 to C_3, where cores C_0 and C_1 are high-performance cores and C_2 and C_3 are energy-efficient cores. The map function schedules the task on the basis of priority, such as the task with high priority will schedule on high-performance cores and the tasks with medium and low priority will schedule on energy-efficient cores.

3.3 Proposed Algorithm

The proposed algorithm is structured into two distinct stages: (i) CPU rank assignment, and (ii) Parallel priority class-based scheduling. In the initial phase, the algorithm determines the rank of each CPU by assessing its performance. This is achieved by executing a dummy task on all cores and using the resulting execution time as the basis for ranking. As a result of this ranking, each CPU is classified as either "High-performance" or "Energy-efficient".

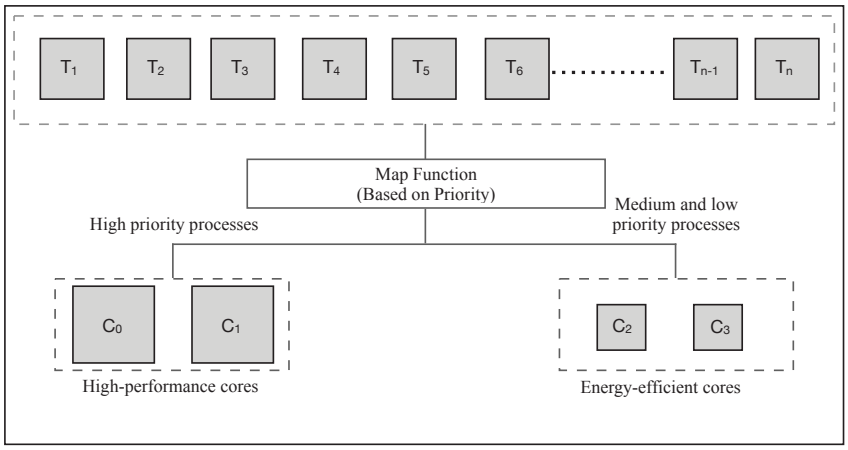


Figure 2 Proposed task scheduling process.

Algorithm 1 Algorithm for Parallel Priority Class

Input: Different Tasks, CPU Ranks
Output: Execution Time of Task according to their Priority
Process:

1. Initiate all the cores according to CPU Ranks
   i.e. $(C_0, C_1) > (C_2, C_3)$ and so on.
2. Get the priority of the tasks.
3. Divide the tasks in to classes according to priority
   (in low, medium and high class).
4. Bind the tasks to particular core according to class
   (medium & low class tasks on energy-efficient cores and high class tasks on high-performance cores).
5. Calculate Execution Time of each high priority tasks.
6. Calculate Speedup of the task execution.

Subsequently, in the second phase, the algorithm retrieves the priorities of all tasks and employs them to schedule the tasks onto specific cores. The scheduling process takes into account the priority, niceness, and weight of each process. Several critical factors influence the task scheduling process, including:

- **Priority and Niceness of process:** The utilization of idle CPU time can be regulated through the "nice" parameter, albeit with some trade-off in terms of speed. In Linux, the "nice" value (NI) is employed within a range of -20 to $+19$, where -20 represents the highest value, 0 is

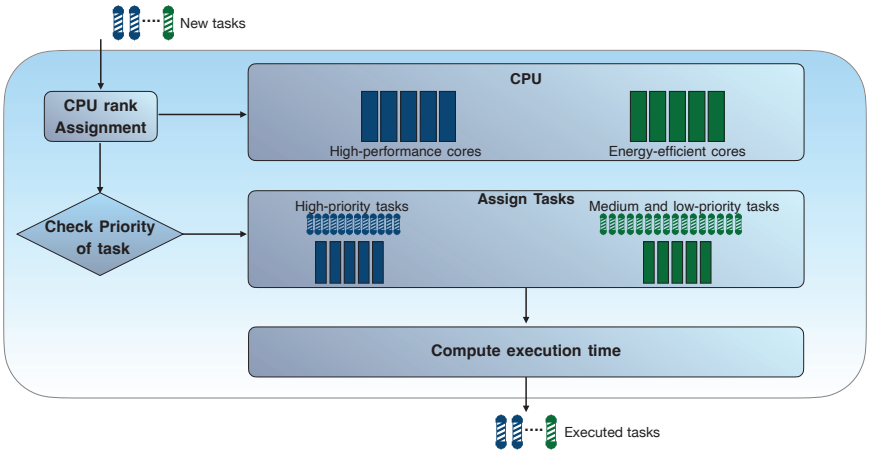


Figure 3 Flow chart of proposed task scheduling process.

the default value, and +19 indicates the lowest value. The priority (PR) denotes the actual priority of a process as utilized by the Linux kernel. There exists a correlation between the "nice" value and the priority, which can be expressed as follows:

$$PR = 20 + NI \tag{1}$$

The priority (PR) ranges from 0 to 39 and maps from 100 to 139 for user processes. The nice value can be changed using a command or system call.

- **Relation between Nice and Weight of Process:** The weight calculation in the Linux Completely Fair Scheduler (CFS) is determined by an exponential increase as the nice value decreases. The weight can be approximately calculated using the following method:

$$Weight = 1024/1.25^{\text{nice value}} \tag{2}$$

3.3.1 CPU rank assignment

In Asymmetric Multi-core Architecture, at least two different cores are present. One core is performance-efficient (big cores), and another is energy-efficient (LITTLE cores). The big cores or performance-efficient cores are designed for efficient performance, and the LITTLE cores are designed for efficient energy [27].

3.3.2 Parallel priority class-based scheduling algorithm

In Multi-core Architecture, OpenMP (Open Multi-Processing) is used to create task or thread-level parallelism in Operating Systems to run concurrently on different cores. With the help of this, the overall execution of the task becomes fast. In OpenMP, the task can run on the same or different cores.

In this, tasks are scheduled on different cores according to their priority. Suppose we have four cores C_0, C_1, C_2, C_3 where (C_0 and C_1) $>$ (C_2 and C_3), so the task having high priority will schedule on core C_0 or C_1, medium and low on core C_2 or C_3.

In the proposed work, CPU Affinity is used to bind the task with a particular core and the set of CPUs on which a thread can be eligible to run is determined by the thread's CPU affinity mask [29]. On a multi-core processor system, the performance benefits can be obtained with the help of CPU affinity mask. By dedicating one CPU to a particular thread (i.e. setting the affinity mask of that thread to specify a single CPU and setting the affinity mask of all other threads to exclude that CPU), [28] it is possible to ensure maximum execution speed for that thread. Restricting a thread to run on a single CPU also avoids the performance cost caused by the cache invalidation that occurs when a thread ceases to execute on one CPU and then recommences execution on a different CPU.

4 Evaluation

This section provides the implementation details and performance evaluation of the proposed model. Here are several noteworthy function calls employed in the implementation of the suggested algorithm.

- **CPU set:** Cpusets offer a mechanism within the Linux kernel that enables the restriction of CPU and Memory Node usage for a process or a group of processes.
- **Set affinity:** It is a Linux system call used for binding a thread thread_RP to the cpu core present in variable cpuset.

$$pthread\_setaffinity\_np(thread\_RP, sizeof(cpu\_set\_t), \&cpuset) \tag{3}$$

- **Schedule parameters:** Param contains priority of threads which can anything between (0-99) for SCHED_RR and policy defines the scheduling policy which is used by cores for scheduling tasks for per cpu core queue. (SCHED_RR means Round Robin Scheduling is used in this

experiment).

$$pthread\_setaffinity\_np(thread\_RP, sizeof(cpu\_set\_t), \&cpuset) \tag{4}$$

- **Sched get priority:** It is used for extracting the priority of any thread or process in Linux.

The rest of this section is structured as follows: Subsection 4.1 outlines the necessary system requirements for conducting the experiment, Subsection 4.2 analyzes the execution time of the suggested method on a Linux system, and Subsection 4.3 provides a performance comparison between CFS and the proposed method.

4.1 System Requirements

The experiments are performed on Dell G15, AMD Ryzen processor, with 6.8 GHz frequency, 8 cores and 20 MB cache size. The proposed method is deployed on Ubuntu, Linux kernel 4.20. The implementation utilizes the following relevant technologies:

1. OpenMP [6] is an Application Programming Interface designed for facilitating parallel programming in languages such as C [18], C++ [26], and FORTRAN [1]. It provides support for various platforms, encompassing Unix-based systems as well as Windows NT systems. OpenMP is primarily oriented towards achieving portability and scalability. The effectiveness of OpenMP is assessed based on the ease and adaptability it affords developers during the implementation process. Essentially, OpenMP serves as a set of compiler specification
2. MPI [9] is a standard message passing system that aimed to work in parallel computing applications. The standard currently supports message-passing programs in FORTRAN, C and C++. Two implementations of the MPI standard will be introduced.

4.2 Comparison

The experiments are performed on multiple tasks with a proper mixture of high, medium and low priorities. The tasks are first scheduled on a normal scheduler Completely Fair Scheduler (CFS), then the same tasks are scheduled with our proposed approach. The overall execution time of high prioritised tasks are calculated, that can be shown in Table 1.

Table 1 Comparison of the proposed algorithm

			Ubuntu 18		Ubuntu 20	
Batch*	**NT**†	**AP**‡	**ETL**§	**ETP**¶	**ETL**§	**ETP**¶
Batch_1	990	99	6.13975	5.54364	5.97126	5.38963
Batch_2	910	99	6.12872	5.52790	5.95716	5.28789
Batch_3	1000	79	6.12692	5.39783	5.98182	5.28907
Batch_4	870	81	6.12898	5.41726	5.99546	5.11901
Batch_5	980	91	6.28981	5.23411	5.90159	5.03341
Batch_6	880	85	6.23981	5.24321	6.02671	5.04233
Batch_7	780	93	9.76146	7.98019	9.56109	7.68614
	Total		46.8155	40.3441	45.3951	38.8475

NT† denotes the total number of high-priority tasks in particular batch, **AP**‡ denotes average priority of high-priority tasks, **ETL**§ denotes execution time (in seconds) required in Linux scheduler, and **ETP**¶ denotes execution time (in seconds) required in proposed method.

Figure 4, compares the execution time of high priority tasks (which are scheduled in form of different batches) of our class based algorithm (in red) with existing Linux based task scheduling(in blue) for Ubuntu 18 and 20. It shows that our proposed algorithm take less execution time as compared to existing Linux scheduling for high priority tasks.

4.3 Performance Measure

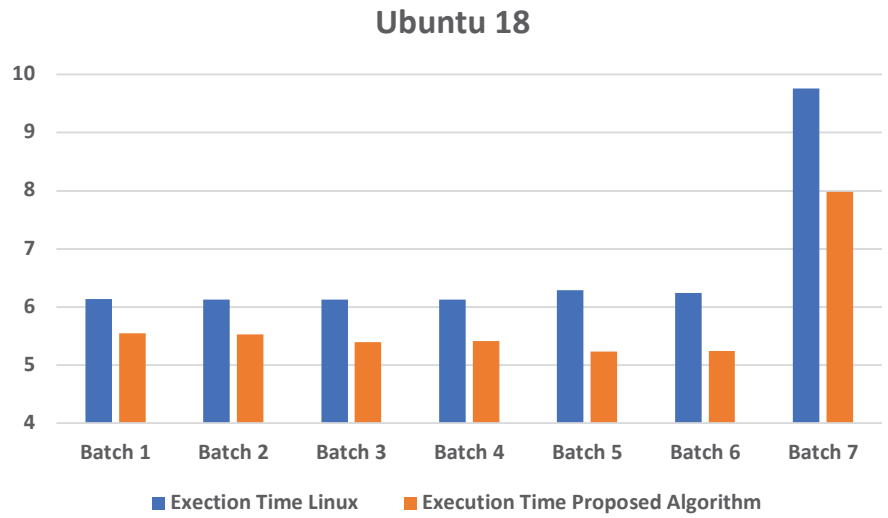
To measure the performance of our proposed algorithm, we have used *speedup*, which can be calculated as,

$$Speedup = \frac{\text{Total time in linux}}{\text{Total time in new algorithm}} \tag{5}$$

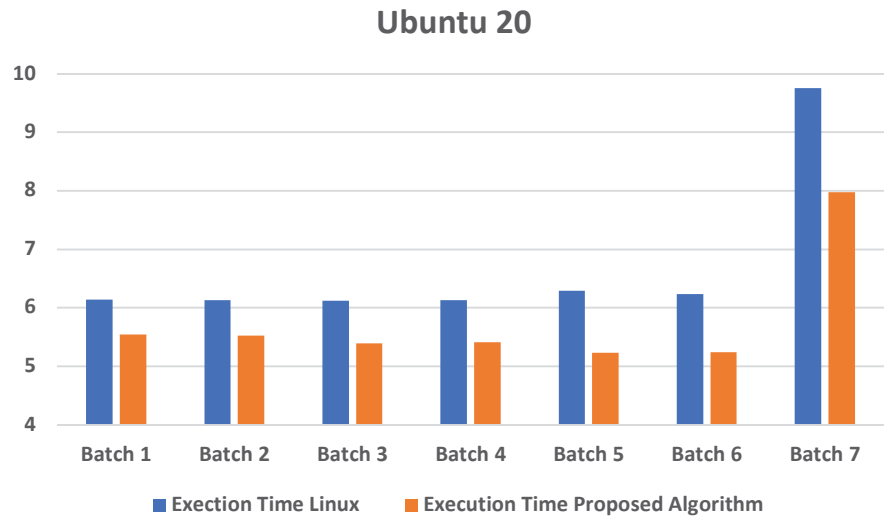
$$Speedup\, Ubuntu18 = \frac{46.8155}{40.3441} = 1.1604 \tag{6}$$

$$Speedup\, Ubuntu20 = \frac{45.3951}{38.8475} = 1.1685 \tag{7}$$

The proposed algorithm shows the speedup of 1.1604 or up to 16 %.



(a) Ubuntu 18



(b) Ubuntu 20

Figure 4 Comparison of proposed algorithm with Linux scheduling algorithm on Ubuntu 18 and Ubuntu 20.

5 Conclusion

In this study, we have made modifications to the existing Linux scheduling technique by introducing priority class-based scheduling. Within the context of Asymmetric Multi-core Processors (AMP), where cores exhibit varying levels of performance (as indicated by task execution times), tasks are typically scheduled on cores without prior knowledge of their priority. To address this, we've implemented a Parallel Priority Class-Based Scheduling approach using OpenMP techniques and CPU affinity. The proposed algorithm leverages a parallel mechanism, allowing tasks to be scheduled on different cores. This approach proves highly effective for high-priority processes, resulting in notable reductions in execution time.

Our algorithm was subjected to testing with a diverse mix of high, medium, and low-priority processes. It demonstrated a substantial improvement of up to 16% in the execution time of high-priority tasks when compared to the existing Linux scheduling algorithm. However, it's important to note that the performance of the proposed algorithm may decrease when there is an increased number of high-priority tasks scheduled simultaneously.

To address this issue, future work in this direction could explore potential solutions. One approach could involve refining the Parallel Priority Class-Based Scheduling strategy to better manage the allocation of low and medium-priority tasks to low-performance cores, minimizing the potential impact on their performance. In future, we also try to in execute the proposed method on SPEC CPU 2017 and compare the performance of it. Additionally, the proposed method can be integrated with load balancing techniques to facilitate the migration of low and medium-priority tasks to high-performance cores when they are idle, thereby enhancing overall system performance.

References

[1] Backus, J.: The history of fortran i, ii, and iii. ACM Sigplan Notices **13**(8), 165–180 (1978)

[2] Bender, M.A., Rabin, M.O.: Scheduling cilk multithreaded parallel programs on processors of different speeds. In: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures. pp. 13–21 (2000)

[3] Bhadauria, M., McKee, S.A.: An approach to resource-aware co-scheduling for cmps. In: Proceedings of the 24th ACM International Conference on Supercomputing. pp. 189–199 (2010)

[4] Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. pp. 72–81 (2008)
[5] Cao, K., Liu, Y., Meng, G., Sun, Q.: An overview on edge computing research. IEEE access **8**, 85714–85728 (2020)
[6] Chandra, R.: Parallel programming in OpenMP. Morgan kaufmann (2001)
[7] Chen, Q., Guo, M., Deng, Q., Zheng, L., Guo, S., Shen, Y.: Hat: history-based auto-tuning mapreduce in heterogeneous environments. The Journal of Supercomputing **64**, 1038–1054 (2013)
[8] Chronaki, K., Rico, A., Casas, M., Moretó, M., Badia, R.M., Ayguadé, E., Labarta, J., Valero, M.: Task scheduling techniques for asymmetric multi-core systems. IEEE Transactions on Parallel and Distributed Systems **28**(7), 2074–2087 (2016)
[9] Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
[10] De Vuyst, M., Kumar, R., Tullsen, D.M.: Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium. pp. 10–pp. IEEE (2006)
[11] Dupont, C., Giaffreda, R., Capra, L.: Edge computing in iot context: Horizontal and vertical linux container migration. In: 2017 Global Internet of Things Summit (GIoTS). pp. 1–4. IEEE (2017)
[12] Fahringer, T.: Optimisation: Operating system scheduling on multi-core architectures. In: seminar parallel computing (2008)
[13] Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-first and help-first scheduling policies for async-finish task parallelism. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–12. IEEE (2009)
[14] Hada, R.P.S., Srivastava, A.: A novel priority based scheduler for asymmetric multi-core edge computing. 3rd International Workshop on Big data driven Edge Cloud Services (BECS 2023) Co-located with the 23rd International Conference on Web Engineering (ICWE 2023), June 6-9, 2023, Alicante, Spain.
[15] Hofmeyr, S., Iancu, C., Blagojević, F.: Load balancing on speed. ACM Sigplan Notices **45**(5), 147–158 (2010)

[16] Hofmeyr, S., Iancu, C., Blagojević, F.: Load balancing on speed. ACM Sigplan Notices **45**(5), 147–158 (2010)
[17] Keckler, S.W., Hofstee, H.P., Olukotun, K.: Multicore processors and systems. Springer (2009)
[18] Kernighan, B.W., Ritchie, D.M.: The c programming language (2002)
[19] Kim, Y., Song, C., Han, H., Jung, H., Kang, S.: Collaborative task scheduling for iot-assisted edge computing. IEEE Access **8**, 216593–216606 (2020)
[20] Koufaty, D., Reddy, D., Hahn, S.: Bias scheduling in heterogeneous multi-core architectures. In: Proceedings of the 5th European conference on Computer systems. pp. 125–138 (2010)
[21] Li, C., Bai, J., Tang, J.: Joint optimization of data placement and scheduling for improving user experience in edge computing. Journal of Parallel and Distributed Computing **125**, 93–105 (2019)
[22] Saez, J.C., Prieto, M., Fedorova, A., Blagodurov, S.: A comprehensive scheduler for asymmetric multicore systems. In: Proceedings of the 5th European conference on Computer systems. pp. 139–152 (2010)
[23] Salami, B., Noori, H., Naghibzadeh, M.: Fairness-aware energy efficient scheduling on heterogeneous multi-core processors. IEEE Transactions on Computers **70**(1), 72–82 (2020)
[24] Salami, B., Noori, H., Naghibzadeh, M.: Online energy-efficient fair scheduling for heterogeneous multi-cores considering shared resource contention. The Journal of Supercomputing pp. 1–20 (2022)
[25] Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE internet of things journal **3**(5), 637–646 (2016)
[26] Stroustrup, B.: The design and evolution of C++. Pearson Education India (1994)
[27] Torvalds, L.: Kernel code. http://www.kernel.com (2021)
[28] Torvalds, L.: Kernel documentation. https://www.kernel.org/doc/readme/ (2021)
[29] Windows: Windows specification. https://msdn.microsoft.com/en-us/library/windows/ (2021)
[30] Xu, C., Lau, F.C.: Load balancing in parallel computers: theory and practice, vol. 381. Springer Science & Business Media (1996)
[31] Zhang, J., Zhou, X., Ge, T., Wang, X., Hwang, T.: Joint task scheduling and containerizing for efficient edge computing. IEEE Transactions on Parallel and Distributed Systems **32**(8), 2086–2100 (2021)

Biographies

Rupendra Pratap Singh Hada is a PhD Research Scholar in Indian Institute of Technology (IIT), Indore, India. Previously, he worked as an assistant professor at the BCST, Indore, India. He received his first degree in computer science engineering from the RGPV University, Bhopal, India in 2015, and he is postgraduate in Computer Engineering from the Shri Govindram Seksaria Institute of Technology and Science, Indore, India in 2019. His PhD research is focused on the Applications of Machine Learning in WSNs.

Abhishek Srivastava is a Professor in the Discipline of Computer Science and Engineering at the Indian Institute of Technology Indore. He completed his Ph.D. in 2011 from the University of Alberta, Canada. Abhishek's group at IIT Indore has been involved in research on service-oriented systems most commonly realised through web-services. More recently, the group has been interested in applying these ideas in the realm of Internet of Things. The ideas explored include coming up with technology agnostic solutions for seamlessly linking heterogeneous IoT deployments across domains. Further, the group is also delving into utilising Machine Learning adapted for constrained environments to effectively make sense of the huge amounts of data that emanate from the vast network of IoT deployments.