*Computer Systems Science & Engineering*
DOI: 10.32604/csse.2023.026238
***Article***

Tech Science Press

# Algorithms for Pre-Compiling Programs by Parallel Compilers

**Fayez AlFayez***

Department of Computer Science and Information, College of Science, Majmaah University, Al-Majmaah, 11952, Saudi Arabia
*Corresponding Author: Fayez AlFayez. Email: f.alfayez@mu.edu.sa
Received: 19 December 2021; Accepted: 10 March 2022

**Abstract:** The paper addresses the challenge of transmitting a big number of files stored in a data center (DC), encrypting them by compilers, and sending them through a network at an acceptable time. Face to the big number of files, only one compiler may not be sufficient to encrypt data in an acceptable time. In this paper, we consider the problem of several compilers and the objective is to find an algorithm that can give an efficient schedule for the given files to be compiled by the compilers. The main objective of the work is to minimize the gap in the total size of assigned files between compilers. This minimization ensures the fair distribution of files to different compilers. This problem is considered to be a very hard problem. This paper presents two research axes. The first axis is related to architecture. We propose a novel pre-compiler architecture in this context. The second axis is algorithmic development. We develop six algorithms to solve the problem, in this context. These algorithms are based on the dispatching rules method, decomposition method, and an iterative approach. These algorithms give approximate solutions for the studied problem. An experimental result is implemented to show the performance of algorithms. Several indicators are used to measure the performance of the proposed algorithms. In addition, five classes are proposed to test the algorithms with a total of 2350 instances. A comparison between the proposed algorithms is presented in different tables discussed to show the performance of each algorithm. The result showed that the best algorithm is the Iterative-mixed Smallest-Longest- Heuristic (ISL) with a percentage equal to 97.7% and an average running time equal to 0.148 s. All other algorithms did not exceed 22% as a percentage. The best algorithm excluding ISL is Iterative-mixed Longest-Smallest Heuristic (ILS) with a percentage equal to 21,4% and an average running time equal to 0.150 s.

**Keywords:** Compiler; encryption; scheduling; big data; algorithms

## 1 Introduction

This work addressess the time challenge of transmitting a big number of encrypted files through a network. Sending several data at the same time by activating the encryption mode is a challenging task. Indeed, compiler data encryption requires time to process data. Faced with huge data we will end up facing a problem of assignment otherwise a problem of waiting will appear and can lead to problems and delay sending data.

Authors in [1], proposed a technique utilizing the scheduling algorithms for memory-bank management, register allocation, and intermediate-code optimizations. The objective was the minimization of the overhead of trace scheduling and the Multi-flow compiler was described. The trace scheduling compiler using the VLIW architecture was examined and investigated in [2–6].

In [7], authors proposed compiler-assisted techniques for operating system services to ensure sufficient energy consumption. The simulation shows the effectiveness of the proposed techniques to achieve better results using dynamic management systems.

A recent study in [8] presented a manner to perform the whole-program transfer scheduling on accelerator data transfers seeking to reduce the number of bytes transferred and enhanced program performance and efficiency.

Other studies such as [9–12] investigate the register allocation and instruction assignment.

A novel domain-specific language and compiler were developed to target FPGAs and CGRAs from common source code in [13]. Authors prove that applications written in spatial are, on average, 42% shorter and achieve a mean speedup of 2.9x over SDAccel HLS.

Several patents were developed regarding the allocation compiler code generation and scheduling, such as in [14,15]. Parallel architectures such as multiprocessors are still difficult to exploit in the presence of compilers [16].

Several authors treated the fair load balancing into computer [17–19]. Other domains that others treated the load blanching are the projects assignment and gas turbines aircraft engines [20–25]. Recently, a novel research work to fight COVID-19 using load balancing algorithm is developed in [26]. Other works treated the scheduling algorithms and the load balancing are [27,28].

The aim of this work is to introduce a novel architecture manipulating the transmission of data through network guaranteeing the effectiveness of encryption. This paper introduces a novel architecture to show the necessity of security into a network. The proposed control aims to ensure a fair distribution of the set of files to different compilers which are responsible for the encryption. To the best of authors knowledge, this problem has never been studied in the literature. Several interesting papers in the literature deal with trace scheduling compiler [29–32].

This paper is organized as the following in Section 2, we begin with an overall presentation of the novel proposed architecture. The studied problem is described in Section 3. The proposed algorithms solving the studied problems are developed in Section 4. Section 5, presents experimental results to show the performance of the proposed algorithms. A conclusion is presented in Section 6.

## 2 Novel Proposed Architecture

In this paper, we propose a new architecture for the data transmission process that requires several security processes. Authors in this project discovered the need of fare distributing that appears when there are files need to be encrypted using compilers before being sent through the network. In presence of big data, we must impose appropriate algorithm that organize the manner of transmission of the files among compilers to guarantee the optimum usage of recourses by distributing the files among compilers for encryption process taking into account the file size. In order to do that, we propose to add a new component into network architecture. This component will be known as “scheduler” that responsible to give a good planning of the dispatching files to different compilers to guarantee time efficiency in the encryption tasks.

The novel architecture proposed in this paper is described in Fig. 1. The component "reception process" is represented by "Scheduler r" and "decrypter" in Fig. 2. This architecture is decomposed into 7 components. These components are given as following:

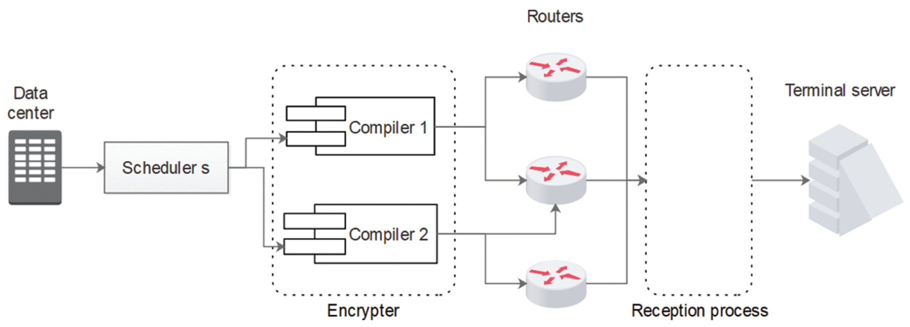

**Figure 1:** Data sent process

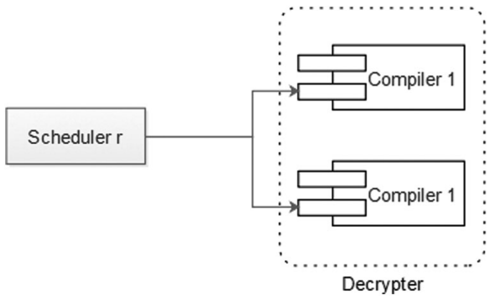

**Figure 2:** The reception process component

- Data center: this component is responsible to collect all files that need to be sent through the network. The collection of files must be done in a secure place.
- scheduler s: is the scheduler responsible to give an appropriate distribution of files that need to be encrypted by the compiler before sending the files through routers.
- Encrypter: this component is responsible to encrypt files. The encrypter contain different compilers that are responsible of executing the encryption algorithm.
- Routers: Scheduler r: scheduler that responsible to receive files from routers and apply the appropriate algorithm to assign files to compilers (see Fig. 3).
- Decrypter: this component applies the algorithm that can decrypt files to be readable by the receiver.
- Terminal server: the receiver accounts.

## 3 Problem Description

In a specific network, any data breach can impact directly to national security, therefore, enforcing a strong encryption system is very important. A strong security system must be applied for any confidential national data to guarantee security and avoid any data leaks.

This section describes the proposed new network architecture that focused on the scheduler component. Assuming that big data saved in *DC* need to be sent through a network at the same time, a problem of concurrency can appear. The data is stored as files in *DC* and each file has its size in bytes. Assuming that file sizes are known and have to be encrypted by a specific coding generator algorithm before sending. The coding algorithm that encrypts files is executed by a compiler. A compiler requires a fixed time to encrypt each file from $DC$. Processing time varies depending on the file size. This work deals with the problem when there are several identical compilers and addresses the following question. How to assign a file to a compiler while maintaining fair load distribution among these compilers? The case where one compiler is still processing encryption tasks, while the second compiler is paused because it has finished all encryption requests shows the importance of fair distribution among compilers. A new scheduling algorithm is required to address such a challenge.

Before we give the solution to the scheduling problem, we must give some notation as follows.

Denoting by $j$ the index of each file $f_j$ which will be sent and by $s_j$ its corresponding size with $j = \{1, \ldots, n_f\}$ and $n_f$ is the number of files. Now, the problem becomes as to how to schedule files having $s_j$ sizes on the $n_c$ compilers. Each file $f_j$ can be assigned to only one compiler $Co_i$ with $i = \{1, \ldots, n_c\}$. We denote by $Cs_j$ the cumulative sizes executed by the compiler $Co_i$ when the file $f_j$ is scheduled. The total executed sizes for each compiler $Co_i$ after finishing assignments will be denoted by $Ts_i$ which will be represent the total size executed by the compiler $Co_i$. The minimum (maximum) total sizes on compiler after finishing scheduling on all compilers is denoted by $Ts_{min}$ ($Ts_{max}$). The total sizes executed by each compiler is sorted as follows: $Ts_2 \leq Ts_2 \leq \ldots \leq Ts_{n_c}$. The following example can explain the presented problem.

***Example 1***

We consider a given instance, with two compilers $n_c = 2$ with five files $n_f = 5$. The sizes of the compiled files are presented In Tab. 1.

**Table 1:** 2-compilers and 5-files instance example

| $j$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $s_j$ | 2314 | 1235 | 4658 | 897 | 3698 |

We chose an algorithm to schedule files on compilers. The algorithm will give the schedule shown in Fig. 3. It is clear to see that compiler 1 execute the encryption program for files {2,3,4}. Contrariwise, for compiler 2, files {1,5} are picked. Based on Fig. 3, the compiler 1 has a total executed size 6790. However, compiler 2 has a total size of 6012. The size gap between compiler 1 and compiler 2 is $Ts_1 - Ts_2 = 778$. The main objective is to search a schedule that improve the result by reducing the obtained gap. In this regards we must search another schedule (if exist) more efficient with gap less than 778.

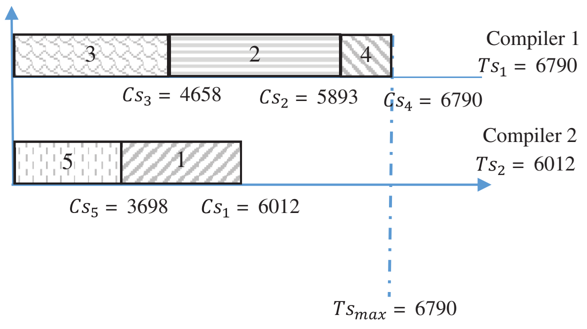

**Figure 3:** Files-compilers dispatching for example 1

We need some indicators to evaluate the performance of different algorithms and the impact of the chosen algorithm on the reducing of the calculated gap. In this paper, we propose to follow the indicator that calculate the difference between all compilers executed sizes and the compiler having the minimum total size. For Example, 1, the indicator is $Ts_1 - Ts_2$. In general, we must calculate $Ts_i - Ts_{min}$ with $i = \{1, \ldots, n_c\}$. Therefore, considering the $n_c$ compilers the total size gap is given in Eq. (1) below:

$$Min \sum_{i=1}^{n_c} [Ts_i - Ts_{min}] \tag{1}$$

Hereafter, let $Ts(g) = \sum_{i=1}^{n_c} [Ts_i - Ts_{min}]$ the final gap between the total sizes executed by compilers. In this paper, Eq. (1) is the objective of the studied problem. The problem is considered as NP-hard in the strong sense. In this study, we utilize several heuristics to solve the problem.

## 4 Approximate Solutions

In this section, we present several approximate solutions to solve the problem. We develop algorithms that return results within a good timing. The proposed heuristics are based essentially on the longest and smallest sizes dispatching rules with some variants. We choose the dispatching rules because the running time of those algorithms is more suitable.

### *4.1 Longest Size Heuristic (LS)*

The files are sorted in non-increasing order of their sizes. After that, we schedule the files on the compiler which has the minimum total size at this time.

### *4.2 Smallest Size Heuristic (SS)*

The files are sorted in increasing order of their sizes. After that, we schedule the files which on the compiler that has the minimum total size at this time.

### *4.3 Half-mixed Longest-Smallest Heuristic (HLS)*

This heuristic is the moderation between the *LS* and the *SS* heuristics. Indeed, the half number of the files will be treated applying *LS* and the remaining files will be treated by applying *SS*. Hereafter we denote by "$Schedule(f_j)$" the procedure that able to assign the file $f_j$ to the most available compiler. The first step to apply *HLS*, we must order files according to the non-decreasing order of its sizes.

The algorithm of the *HLS* is given as follows.

**Algorithm 1:** Half-mixed Longest-Smallest algorithm (*HLS*)

```
Step 0     Set $j = n_f$.
Step 1     While $(j > \frac{n_f}{2})$
Step 2         $Schedule(f_j)$
Step 3         $j--$
Step 4     EndWhile
Step 5     While $\left(j \le \frac{n_f}{2}\right)$
Step 6         $Schedule(f_j)$
Step 7         $j++$
Step 8     EndWhile
Step 9     Calculate $Ts(g)$
Step 10    Return $Ts(g)$.
```

### *4.4 Half-mixed Smallest-Longest Heuristic (HSL)*

This heuristic is the moderation between the *SS* and the *LS* the heuristics. Indeed, the half number of the files will be treated applying *SS* and the remaining files will be treated by applying *LS*.

### *4.5 Iterative-mixed Longest-Smallest Heuristic (ILS)*

For this heuristic, instead of scheduling half of the larger files, we iterated over larger $\beta$ and smaller $n_f - \beta$ by doing $\beta$ variant from 1 to $n_f - 1$. The first step to apply *ILS*, we must order files according to the non-decreasing order of its sizes. The algorithm of the heuristic *ILS* is given as follows.

**Algorithm 2:** Iterative-mixed Longest-Smallest algorithm (*ILS*)

```
Step 0    Set $j = n_f$.
Step 1    For $(\beta = 1 \ to \ n_f - 1)$
Step 2      While $(j > n_f - \beta)$
Step 3    $Schedule(f_j)$
Step 4          $j--$
Step 5       EndWhile
Step 6       $j = 1$
Step 7      While $(j \le n_f - \beta)$
Step 8         $Schedule(f_j)$
Step 9       $j++$
Step 10     EndWhile
Step 11     Calculate $Ts^{\beta}(g)$
Step 12     $j = n_f$
Step 13   EndFor
Step 14   Calculate $Ts(g) = \min_{1 \le \beta \le n_f - 1} Ts^{\beta}(g)$
Step 15   Return $Ts(g)$.
```

### *4.6 Iterative-mixed Smallest-Longest- Heuristic (ISL)*

For this heuristic, we adopt the same idea developed for *ILS*. The modification is instead we start by the largest files, here we start by the smallest files after that the $n_f - \beta$ largest files.

## 5 Experimental Results

In this section, we adopt and examine several classes that gives different manner of generation of instances in order to discuss the results and examine the assessment of the proposed algorithms.

All developed algorithms are coded with Microsoft Visual C++ (Version 2013). The proposed algorithms were coded and executed on an Intel(R) Core (TM) i5-3337U CPU @ 1.8 GHz and 8 GB RAM. The operating system utilized throughout the research work is windows 10 with 64 bits. these algorithms were examined on five different types of sets of instances. We generate the size $s_j$ based on different distributions and each distribution represents a class. Let $U[n, m]$ be the uniform distribution between $[n, m]$ and $N[n, m]$ be the normal distribution. The generation of all instances in this paper based on classes will be as follows.

- Class 1: $s_j$ in $U[100, 500]$.
- Class 2: $s_j$ in $U[1000, 3000]$.
- Class 3: $s_j$ in $U[100, 2000]$.
- Class 4: $s_j$ in $N[50 - \ 500]$.
- Class 5: $s_j$ in $N[25 - \ 1000]$.

The triple $n_f$, $n_c$ and *Class* is the criterion that allows us to generate sizes.

The pair $(n_f, \ n_c)$ have varied values. We choose the varied values presented in Tab. 2.

**Table 2:** Generation of $\left(\boldsymbol{n_f}, \ \boldsymbol{n_c}\right)$

| $n_f$ | $n_c$ |
|---|---|
| $20, 50$ | $2, 3, 5$ |
| $100, 200, 250, 300$ | $2, 3, 5, 10$ |
| $500, 1000$ | $2, 3, 5, 10, 15$ |
| 1500, 2000, 2500 | $2, 3, 5, 10, 15, 20$ |

Tab. 2 have in total 2350 instances. Different variables are selected to measure the performance assessment of the proposed algorithms. These variables are:

- $H_b$ the best returned value of all algorithms.
- $H$ the value returned by the studied heuristic.
- $GAP = \dfrac{H - H_b}{H}$.
- *Time* the running time for algorithm in seconds. We denoted by "–" is the time is less than 0.001 s.
- *Perc* the percentage among all instances that the condition $H_b = H$ is satisfied.

An overall of results, present *Perc* and *Time* given by all proposed algorithms, is depicted in Tab. 3.

**Table 3:** Overall, of all algorithms

| | *LS* | *SS* | *HLS* | *HSL* | *ILS* | *ISL* |
|---|---|---|---|---|---|---|
| *Perc* | 17.4% | 0.0% | 0.0% | 16.4% | 21.4% | 97.7% |
| *Time* | – | – | – | – | 0.150 | 0.148 |

Tab. 3 shows that the algorithm that conduct the best value is *ISL* with $Perc = 97.7\%$ and $Time = 0.148\ s$, compared to *SPT* which have a *Perc* equal to 16.2%. On other hand, the heuristic *ILS* is participated with 21.4%. Tab. 4 shows the behavior of *GAP* according to $n_f$. From Tab. 4 we can see that when varying the number of files $(n_f)$, the *GAP* of the proposed algorithms changes. The latter table shows that there isn't any correlation between the number of files and the evaluated average gap.

**Table 4:** Variation of *GAP* according to $n_f$

| $n_f$ | *LS* | *SS* | *HLS* | *HSL* | *ILS* | *ISL* |
|---|---|---|---|---|---|---|
| 20 | 0.42 | 0.82 | 0.75 | 0.54 | 0.39 | 0.08 |
| 50 | 0.63 | 0.88 | 0.84 | 0.74 | 0.61 | 0.03 |
| 100 | 0.69 | 0.92 | 0.89 | 0.73 | 0.69 | 0.02 |
| 200 | 0.72 | 0.93 | 0.90 | 0.66 | 0.69 | 0.00 |
| 250 | 0.71 | 0.94 | 0.92 | 0.85 | 0.71 | 0.00 |
| 300 | 0.72 | 1.00 | 0.99 | 0.67 | 0.72 | 0.00 |
| 500 | 0.69 | 0.90 | 0.86 | 0.66 | 0.67 | 0.00 |
| 1000 | 0.67 | 0.90 | 0.87 | 0.69 | 0.66 | 0.00 |
| 1500 | 0.63 | 1.00 | 1.00 | 0.68 | 0.63 | 0.00 |
| 2000 | 0.60 | 0.94 | 0.92 | 0.61 | 0.58 | 0.00 |
| 2500 | 0.55 | 0.95 | 0.93 | 0.71 | 0.55 | 0.00 |

For algorithms $[LS, SS, HLS, HSL, ILS, ISL]$ the worst *GAP* values was obtained for the following values of $n_f$ [200,300,1500,250,300,20] respectively. In addition, Tab. 4 shows that heuristic *ISL* have the best $GAP = 0$ for all $n_f$ excluding $n_f = \{20{,}50{,}100\}$. The corresponding running time for algorithms detailed in Tab. 4 is given in Tab. 5 below.

**Table 5:** Running time for each algorithm and each number of files

| $n_f$ | *LS* | *SS* | *HLS* | *HSL* | *ILS* | *ISL* |
|---|---|---|---|---|---|---|
| 20 | – | – | – | – | – | – |
| 50 | – | – | – | – | – | – |
| 100 | – | – | – | – | 0.001 | 0.001 |
| 200 | – | – | – | – | 0.002 | 0.002 |
| 250 | – | – | – | – | 0.003 | 0.003 |
| 300 | – | – | – | – | 0.003 | 0.003 |
| 500 | – | – | – | – | 0.016 | 0.015 |
| 1000 | – | – | – | – | 0.087 | 0.085 |
| 1500 | – | – | – | – | 0.219 | 0.219 |
| 2000 | – | – | – | – | 0.417 | 0.407 |
| 2500 | – | – | – | – | 0.660 | 0.658 |

Tab. 5 shows that running time increase when the number of files increase for *ILS* and *ISL*. For other algorithms the time is less than 0.001 s. Tab. 6 presents the results of the *GAP* value according to the number of compilers $n_c$. The worst $GAP = 0.99$ value is given for *SS* when $n_c = \{2, 10\}$ and for *HLS* when $n_c = 2$. The best *GAP* value is obtained by *ISL* when $n_c = 3$ and $n_c = 20$.

**Table 6:** Variation of ***GAP*** according to ***nc***

| $n_c$ | *LS* | *SS* | *HLS* | *HSL* | *ILS* | *ISL* |
|---|---|---|---|---|---|---|
| 2 | 0.58 | 0.99 | 0.99 | 0.64 | 0.57 | 0.02 |
| 3 | 0.65 | 0.79 | 0.71 | 0.71 | 0.61 | 0.00 |
| 5 | 0.65 | 0.98 | 0.97 | 0.64 | 0.65 | 0.02 |
| 10 | 0.69 | 0.99 | 0.98 | 0.68 | 0.69 | 0.00 |
| 15 | 0.65 | 0.75 | 0.67 | 0.72 | 0.63 | 0.00 |
| 20 | 0.65 | 0.98 | 0.98 | 0.82 | 0.64 | 0.00 |

The corresponding running times for algorithms detailed in Tab. 6 are given in Tab. 7 below. Tab. 7 shows that running time increase when the number of files increases for ILS and ISL. For other algorithms, the time is less than 0.001 s.

**Table 7:** Running Time for each algorithm and each number of compilers

| $n_c$ | *LS* | *SS* | *HLS* | *HSL* | *ILS* | *ISL* |
|---|---|---|---|---|---|---|
| 2 | – | – | – | – | 0.109 | 0.110 |
| 3 | – | – | – | – | 0.111 | 0.113 |
| 5 | – | – | – | – | 0.119 | 0.117 |
| 10 | – | – | – | – | 0.167 | 0.161 |
| 15 | – | – | – | – | 0.065 | 0.061 |
| 20 | – | – | 0.001 | 0.001 | 0.690 | 0.685 |

The behavior of the average gap according to different classes is showed in Tab. 8. This last table shows that there is not any correlation between the different classes and average gap. Thus, all classes having the same difficulty because the average gap differed slightly between class and other one.

**Table 8:** Variation of ***GAP*** according to ***Class***

| *Class* | *LS* | *SS* | *HLS* | *HSL* | *ILS* | *ISL* |
|---|---|---|---|---|---|---|
| 1 | 0.49 | 0.99 | 0.98 | 0.76 | 0.49 | 0.01 |
| 2 | 0.67 | 0.99 | 0.97 | 0.76 | 0.66 | 0.02 |
| 3 | 0.65 | 0.99 | 0.99 | 0.83 | 0.65 | 0.02 |
| 4 | 0.75 | 0.89 | 0.83 | 0.60 | 0.71 | 0.00 |
| 5 | 0.65 | 0.78 | 0.75 | 0.47 | 0.64 | 0.00 |

The corresponding running times for algorithms detailed in Tab. 8 are given in Tab. 9 below.

**Table 9:** Running time for each algorithm and each *Class*

| *Class* | *LS* | *SS* | *HLS* | *HSL* | *ILS* | *ISL* |
|---|---|---|---|---|---|---|
| 1 | – | – | – | – | 0.149 | 0.148 |
| 2 | – | – | – | – | 0.152 | 0.151 |
| 3 | – | – | – | – | 0.153 | 0.150 |
| 4 | – | – | – | – | 0.147 | 0.146 |
| 5 | – | – | – | – | 0.146 | 0.146 |

Tab. 9 shows that the algorithms *ILS* and *ISL* have the running time higher than *LS*, *SS*, *HLS* and *HSL*.

## 6 Conclusion

This work mainly focused on the fair distribution problem of files before the encryption process in compilers. The main contribution of this paper is based essentially on two approaches. Firstly, the proposal of a novel architecture regarding the transmission of different files in the presence of big data into the network using encryption process for files before transmitting to a network. Secondly, scheduling problem of assigning fair file loads to compilers in order to ensure timely compilation process. The problem studied in this paper is an NP-hard problem. In order to assure a fair distribution of files among different compilers, this work gives several solutions and approaches. These approaches improve the network performance and allow sending the maximum data in remarkable time. Several algorithms are proposed to solve the current challenge. The experimental results show that no dominance between algorithms and time is very interesting. The proposed heuristics can be used in the future to compare with other algorithms and develop an exact method to solve the problem.

**Acknowledgement:** The author would like to thank the Deanship of Scientific Research at Majmaah University for supporting this work under Project Number No. R-2022-85.

**Funding Statement:** The author received Funding for this study from Majmaah University.

**Conflicts of Interest:** The author declare that they have no conflicts of interest to report regarding the present study.

## References

[1] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. Lichtenstein, R. P. Nix *et al.,* "The multiflow trace scheduling compiler," *The Springer International Series in Engineering and Computer Science*, vol. 234, pp. 51–142, 1993.

[2] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 967–979, 1988.

[3] R. P. Colwell, R. P. Nix, J. J. O'donnell, D. B. Papworth and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *ACM SIGARCH Computer Architecture News*, vol. 15, no. 5, pp. 180–192, 1987.

[4] J. R. Ellis, "Bulldog: A compiler for VLIW architectures," Ph.D. dissertation, Yale Univ., New Haven, CT (USA), 1985.

[5] O. Mutlu, S. Mahlke, T. Conte and W.-m Hwu, "Iterative modulo scheduling," *IEEE Micro*, vol. 38, no. 1, pp. 115–117, 2018.

[6] C. Six, S. Boulmé and D. Monniaux, *Certified compiler backends for VLIW processors highly modular postpass-scheduling in the compcert certified compiler*, France, hal, 2019.

[7] D. Mosse, H. Aydin, B. Childers and R. Melhem, “Compiler-assisted dynamic power-aware scheduling for real-time applications,” in *Workshop on Compilers and Operating Systems for Low Power*, Louisiana, USA, 2000.

[8] M. B. Ashcraft, A. Lemon, D. A. Penry and Q. Snell, “Compiler optimization of accelerator data transfers,” *International Journal of Parallel Programming*, vol. 47, no. 1, pp. 39–58, 2019.

[9] R. C. Lozano and C. Schulte, “Survey on combinatorial register allocation and instruction scheduling,” *ACM Computing Surveys (CSUR)*, vol. 52, pp. 1–50, 2019.

[10] R. C. Lozano, M. Carlsson, G. H. Blindell and C. Schulte, “Combinatorial register allocation and instruction scheduling,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, pp. 1–53, 2019.

[11] Ł. Domagała, D. van Amstel, F. Rastello and P. Sadayappan, “Register allocation and promotion through combined instruction scheduling and loop unrolling,” in *Proc. of the 25th Int. Conf. on Compiler Construction*, Barcelona Spain, pp. 143–151, 2016.

[12] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev *et al.,* “Register optimizations for stencils on GPUs,” in *Proc. of the 23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Vienna Austria, pp. 168–182, 2018.

[13] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang and S. Hadjis, “Spatial: A language and compiler for application accelerators,” in *Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Philadelphia PA USA, pp. 296–311, 2018.

[14] H. M. Jacobson, R. Nair, J. K. O'brien and Z. N. Sura, “Power-constrained compiler code generation and scheduling of work in a heterogeneous processing system,” *Google Patents*, 2015.

[15] A. F. Glew, D. A. Gerrity and C. T. Tegreene, “Entitlement vector for library usage in managing resource allocation and scheduling based on usage and priority,” *Google Patents*, 2016.

[16] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg and A. Nicolau, “Parallel processing: A smart compiler and a dumb machine,” in *Proc. of the 1984 SIGPLAN Symp. on Compiler Construction*, Montreal Canada, pp. 37–47, 1984.

[17] M. Jemmali and H. Alquhayz, “Equity data distribution algorithms on identical routers,” in *Int. Conf. on Innovative Computing and Communications*, Ostrava, Czech Republic, pp. 297–305, 2020.

[18] H. Alquhayz, M. Jemmali and M. M. Otoom, “Dispatching-rule variants algorithms for used spaces of storage supports,” *Discrete Dynamics in Nature and Society*, vol. 2020, pp. 1–9, 2020.

[19] H. Alquhayz and M. Jemmali, “Max-min processors scheduling,” *Information Technology and Control*, vol. 50, pp. 5–12, 2021.

[20] M. Alharbi and M. Jemmali, “Algorithms for investment project distribution on regions,” *Computational Intelligence and Neuroscience*, vol. 2020, pp. 1–13, 2020.

[21] M. Jemmali, “An optimal solution for the budgets assignment problem,” *RAIRO—Operations Research*, vol. 55, pp. 873–897, 2021.

[22] M. Jemmali, “Budgets balancing algorithms for the projects assignment,” *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 10, pp. 574–578, 2019.

[23] M. Jemmali, L. K. B. Melhim, S. O. B. Alharbi and A. S. Bajahzar, “Lower bounds for gas turbines aircraft engines,” *Communications in Mathematics and Applications*, vol. 10, no. 3, pp. 637–642, 2019.

[24] M. Jemmali, “Projects distribution algorithms for regional development,” *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal*, vol. 10, no. 3, pp. 293–305, 2021.

[25] M. Jemmali, L. K. B. Melhim and M. Alharbi, “Randomized-variants lower bounds for gas turbines aircraft engines,” in *World Congress on Global Optimization*. Metz, France, 949–956, 2019.

[26] M. Jemmali, “Intelligent algorithms and complex system for a smart parking for vaccine delivery center of COVID-19,” *Complex & Intelligent Systems*, Vol. 8, pp. 597–609, 2021.

[27] M. Jemmali, M. M. Otoom and F. AlFayez, “AlFayez Max-min probabilistic algorithms for parallel machines,” in *Int. Conf. on Industrial Engineering and Industrial Management*, Paris, France, pp. 19–24, 2020.

[28] M. Jemmali and A. Alourani, “Mathematical model bounds for maximizing the minimum completion time problem,” *Journal of Applied Mathematics and Computational Mechanics*, vol. 20, no. 4, pp. 43–50, 2022.

[29] S. M. Freudenberger, T. R. Gross and P. G. Lowney, “Avoidance and suppression of compensation code in a trace scheduling compiler,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1156–1214, 1994.

[30] G. Shobaki, K. Wilken and M. Heffernan, “Optimal trace scheduling using enumeration,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 4, pp. 1–32, 2009.

[31] J. Eisl, D. Leopoldseder and H. Mössenböck, “Parallel trace register allocation,” in *Proc. of the 15th Int. Conf. on Managed Languages & Runtimes*, Linz Austria, pp. 1–7, 2018.

[32] J. Eisl, *Divide and allocate: The trace register allocation framework*, San Francisico, CA, USA, ACM, 2018.