Intelligent Automation & Soft Computing

Tech Science Press

DOI: 10.32604/iasc.2024.054154

ARTICLE

Data-Oriented Operating System for Big Data and Cloud

Selwyn Darryl Kessler, Kok-Why Ng[*] and Su-Cheng Haw[*]

Faculty of Computing and Informatics, Multimedia University, Cyberjaya, 63100, Malaysia

*Corresponding Authors: Kok-Why Ng. Email: kwng@mmu.edu.my; Su-Cheng Haw. Email: sucheng@mmu.edu.my

Received: 20 May 2024 Accepted: 22 July 2024 Published: 06 September 2024

ABSTRACT

Operating System (OS) is a critical piece of software that manages a computer's hardware and resources, acting as the intermediary between the computer and the user. The existing OS is not designed for Big Data and Cloud Computing, resulting in data processing and management inefficiency. This paper proposes a simplified and improved kernel on an x86 system designed for Big Data and Cloud Computing purposes. The proposed algorithm utilizes the performance benefits from the improved Input/Output (I/O) performance. The performance engineering runs the data-oriented design on traditional data management to improve data processing speed by reducing memory access overheads in conventional data management. The OS incorporates a data-oriented design to "modernize" various Data Science and management aspects. The resulting OS contains a basic input/output system (BIOS) bootloader that boots into Intel 32-bit protected mode, a text display terminal, 4 GB paging memory, 4096 heap block size, a Hard Disk Drive (HDD) I/O Advanced Technology Attachment (ATA) driver and more. There are also I/O scheduling algorithm prototypes that demonstrate how a simple Sweeping algorithm is superior to more conventionally known I/O scheduling algorithms. A MapReduce prototype is implemented using Message Passing Interface (MPI) for big data purposes. An attempt was made to optimize binary search using modern performance engineering and data-oriented design.

KEYWORDS

Operating system; big data; cloud computing; MapReduce; data-oriented

1 Introduction

An operating system (OS) is a critical piece of software that manages a computer's hardware and resources, acting as the intermediary between the computer and the user. They are essential for the proper functioning of a computer, from the smallest electronic device to the largest exascale supercomputer in the world [1,2] and all the servers that make up the Internet.

The popularity of cloud computing and big data has sky-rocketed in recent years. This is mainly due to the power, cost-effectiveness, and flexibility that cloud computing offers to people all around the world. Cloud computing provides a platform to deliver various computing services over the Internet. These services can include storage, servers, software, analytics, databases, raw computing power, and many more. It is fair to say that a large portion of the online world now exists on the cloud.

Big data, on the other hand, has been revolutionized by cloud computing [3–5]. No longer are traditional data management techniques limited by IT infrastructure. Now, organizations can easily deploy a scalable and flexible platform for collecting, storing, and analyzing massive amounts of data. This has given them the ability to extract insights from data and make informed business decisions far better than ever before, and on a far larger scale.

What the general public does not realize is that the vast majority of this magic is powered by Hard Disk Drives (HDD) in data centers around the world. The reality is that HDDs are completely dominant in the cloud computing space. HDDs are very cheap and very scalable at massive scales, compared to Solid-State Drives (SSD) which dominate the client computing space. This means that cloud computing and big data can be optimized for HDD performance, something that is surprisingly lacking in the OS space. There are several problem statements. Firstly, there are not many options for operating systems and this lack of choices extends to embedded systems as well. In addition to that, the current operating systems do not take advantage of hard drive firmware I/O optimization procedures [6–8]. They are also not optimized for big data and cloud computing [9–11]. Furthermore, the adoption of data-oriented design and modern computing concepts in data science is very poor outside of the widely used libraries [12,13]. Lastly, with a new platform, it is now required new data management solutions. Previously implemented solutions no longer work out of the box on our new big data operating system. Therefore, we have to develop our own solutions.

The first aim of this paper is to develop a simple kernel (OS) on an x86 system that is designed for big data and cloud computing purposes. The next aim is to implement a big data algorithm that could be run on this kernel in order to perform big data processing while utilizing the performance benefits from the improved I/O performance. The final aim is to perform performance engineering to implement data-oriented design on traditional data management to improve the speed of data processing by reducing the memory access overheads that are present in conventional data management.

2 Related Work

2.1 Operating System

According to Silberschatz et al. [14], a simple definition of an operating system is software that manages the hardware of a computer. It acts as the mediator between a user and the computer hardware [13]. They also provide a platform in which application programs run. An operating system can also be considered a resource allocator and a control program that manages the execution of programs and processes within a computer.

A typical computer system can be divided into four parts:

- The Hardware: This encompasses the central processing unit (CPU), the memory, the storage, various I/O devices, and other processing units such as the graphics processing unit (GPU).
- The Operating System: This is what it would be working on. This has already been defined just above this list.
- The Application Programs: These are software created to run on top of an operating system that uses computer resources to solve users' computing problems.
- The User: This is the end user of the computer. Their entire system would be designed with the end user in mind.

A brief top-down overview of a computer system is shown in Fig. 1 below.

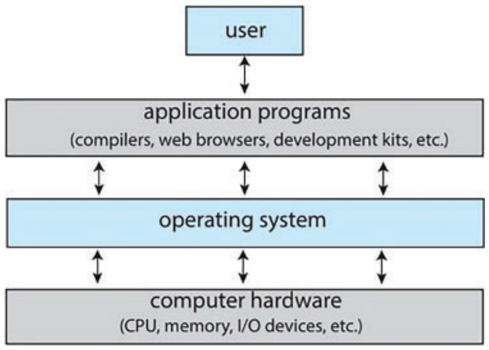

user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

Figure 1: A typical computer system

2.2 Real-Time Systems

A system is defined as a real-time system when we require a quantitative expression of time to describe the behavior of the said system [15,16]. Real-time refers to the quantitative description of time. The concepts of real-time systems are widely used in embedded systems due to the nature of embedded systems in general [17–19]. For example, in an automated machine assembly plant, as machines get assembled on conveyor belts, they have imposed a time constraint where the assigned work must be completed before the work is passed on to the next process on the conveyor belt.

There are a few key characteristics of real-time systems that distinguish real-time systems from non-real-time systems [20,21]. Some key characteristics are as follows:

- Time Constraints. Every single real-time task is associated with some form of time constraint. Every task would be assigned a deadline which states a time limit in which the task must be completed.
- Different Criteria for Correctness. In real-time systems, for a result to be considered correct, they must both produce the logically correct result as well as meet the imposed deadline. Failure to meet the deadline would be considered a failure or an incorrect result.
- Embedded. The sheer majority of real-time systems are embedded systems. An embedded system would use sensors to collect information which is then passed on to a real-time computer for processing.
- Safety-Criticality. Reliability and safety are usually considered two separate metrics in traditional non-real-time tasks. In lots of real-time systems, the opposite is true. Reliability and safety are usually tightly correlated. A safe system does not inflict damage when it fails whereas a reliable system does not fail regularly.

While these are important characteristics of real-time systems, it does not mean that all real-time systems have all these characteristics [22–24]. For instance, missing the deadline may lead to system failure, degraded performance, or unsafe conditions. Real-time systems often have additional correctness criteria related to timing. Producing the right output at the wrong time can also be as problematic as producing the wrong output. Real-time systems encompass a diverse range of other characteristics that have not been addressed in this discussion, indicating the complexity and variability within this field [25–27].

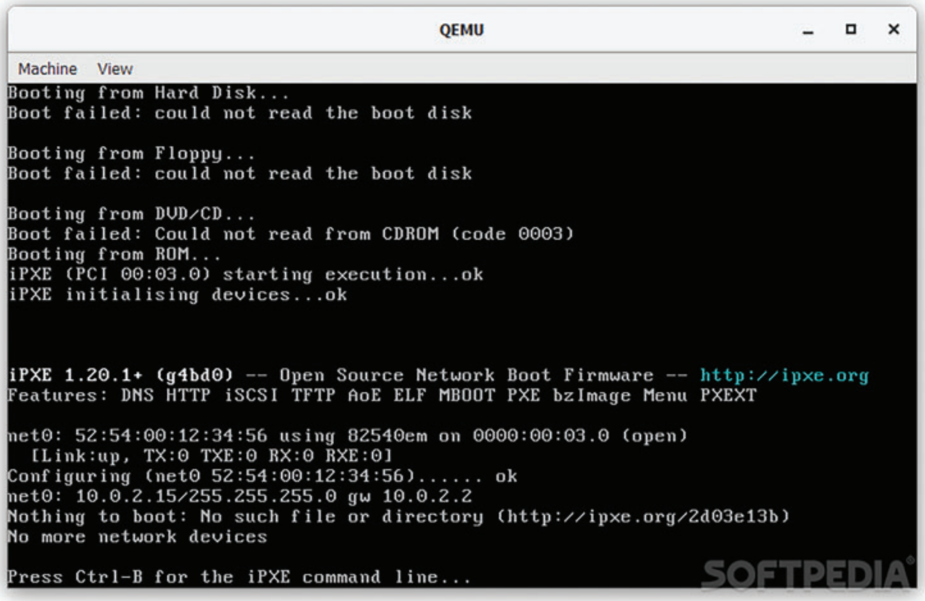
3 Methodology

3.1 GCC (GNU Compiler Collection) Cross-Compiler

We need to prepare our own cross-compiler if we are to have any hope of implementing our own data-oriented operating system. For this paper, a GCC cross-compiler is used where it can build for a generic target (i686-elf). This would mean that the cross-compiler would have no headers or libraries for the development operating system. This is crucial. Otherwise, the native compilers would just assume that we are developing on the same platform and produce various undesirable results.

3.2 Quick Emulator (QEMU) Emulator

QEMU is an open-source machine emulator [28] that serves to emulate hardware that we can run our kernel on (see Fig. 2). It provides options for many different types of architectures which means we can emulate running our kernel on various devices. However, for this paper, the generic i686 target is elected instead.



```
Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
Boot failed: could not read the boot disk

Booting from DVD/CD...
Boot failed: Could not read from CDROM (code 0003)
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok



iPXE 1.20.1+ (g4bd0) -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS HTTP iSCSI TFTP AoE ELF MBOOT PXE bzImage Menu PXEXT

net0: 52:54:00:12:34:56 using 82540em on 0000:00:03.0 (open)
  [Link:up, TX:0 TXE:0 RX:0 RXE:0]
Configuring (net0 52:54:00:12:34:56)...... ok
net0: 10.0.2.15/255.255.255.0 gw 10.0.2.2
Nothing to boot: No such file or directory (http://ipxe.org/2d03e13b)
No more network devices

Press Ctrl-B for the iPXE command line...
```

Figure 2: The QEMU emulator in action

Most of the development of this system would be performed on the QEMU emulator rather than real hardware. This is because it is much easier to debug and develop on QEMU as compared to developing directly on the real hardware. With QEMU, it would have access to the **G**NU **De**Bugger (GDB), something it would not have access to if it were to develop directly on bare metal. Another issue is the matter of security. If it were to accidentally execute dangerous code, it would not damage the actual hardware.

3.3 Memory Management

For the operating system to be functional, it needs to be able to manage its own memory. In this sense, a memory management system that consists of two primary sections are implemented:

a. The Heap.
b. The Stack.

The implementation of the heap that has been chosen is one that splits memory into 4 KB chunks because 4 KB sectors are widely used in the storage industry. It utilizes an allocation table that points to all available chunks in memory. Each entry in the allocation table would correspond to the state of the associated chunk. An entry would consist of 1 byte (8 bits).

Besides memory allocation, the operating system can contain paging which allows us to remap memory addresses from one address to another. Paging works by default on Intel CPUs with 4 KB blocks. When paging is enabled, the memory management unit (MMU) will automatically look at the page tables to resolve virtual addresses into physical addresses.

An example of the implementation of paging is shown in Fig. 3 below.

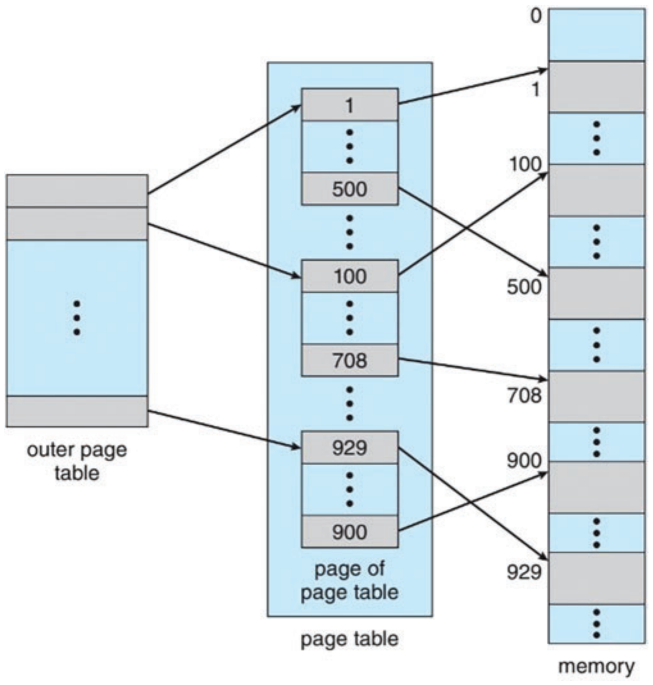


Figure 3: Example of paging

3.4 *Custom I/O Scheduler for Cloud Computing and Big Data*

We would implement a custom I/O scheduler for big data and cloud computing purposes. The I/O scheduler should be designed in such a way that it takes advantage of HDD performance features. The I/O scheduler should also be able to prioritize tail latency and latency optimizations to improve cloud service delivery.

3.5 *Big Data Processing Using MapReduce*

For this paper, a simple MapReduce prototype should be developed to demonstrate that big data processing is possible in a distributed fashion across the cloud.

In order to successfully implement MapReduce, the following steps (as shown in Fig. 4) need to be taken.

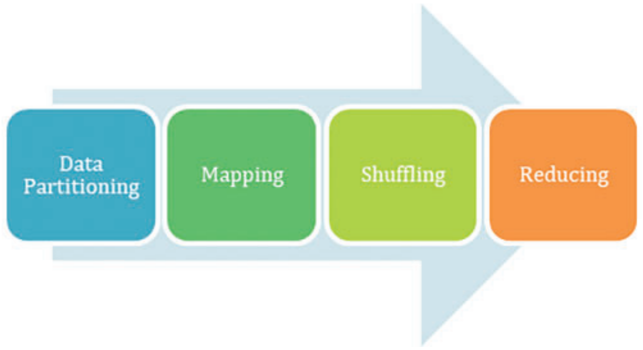


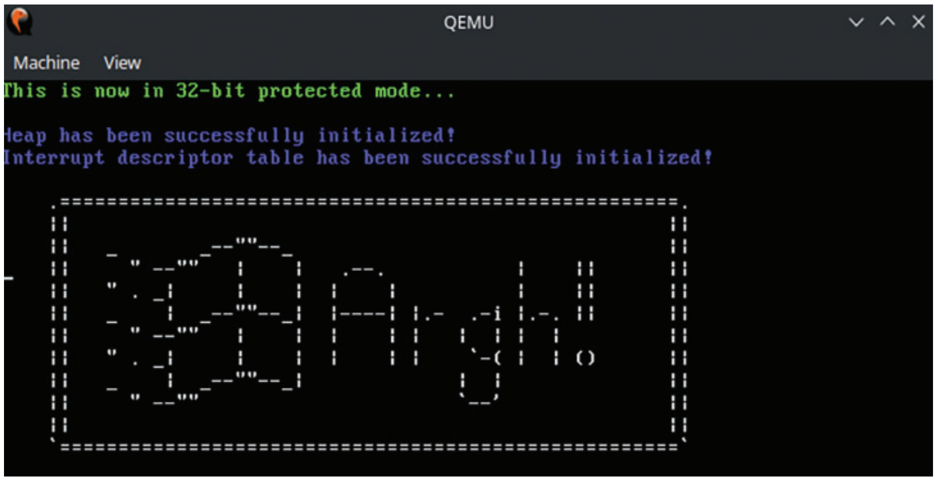
Figure 4: Implementation of the MapReduce

The first step is to divide the data into smaller chunks to be distributed among a cluster of computers through data partitioning. Next, in the mapping process, the partitioned are chunks into intermediate key-value pairs. The next process is shuffling. In this step, we group the key-value pairs into their respective keys. This is to ensure all values with the same key are processed by the same reduced task. Finally, it is the reducing process, whereby each group of data is reduced into a final result. It intends to implement the MapReduce algorithm only as a prototype or a proof-of-concept as implementing it on the new kernel would require Herculean effort: the entire TCP/IP stack would be implemented on a new platform. That is simply unfeasible.

4 Experimental Results

4.1 OS

Most of the work in this paper is very abstract and thus, it may be difficult to present. A lot of work has been done on things that do not produce output that is meaningful to the human eye. Fig. 5 shows a simple screenshot of the OS in action as emulated in QEMU.



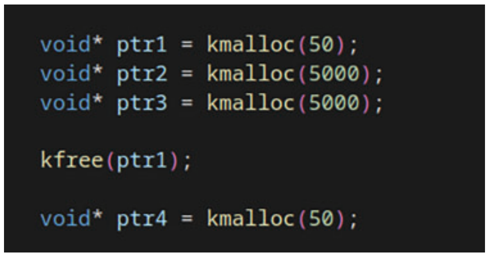
```
This is now in 32-bit protected mode...

Heap has been successfully initialized!
Interrupt descriptor table has been successfully initialized!

    .=====================================================.
    ||                                                   ||
    ||             __""__                                ||
    ||    _  " __""_      _                              ||
    ||         "     |      |  .--.           |   ||     ||
    ||   "  . _|     |      | |    |          |   ||     ||
    ||    _    |  __""__ _| |----| |.-  .-i |.-. ||     ||
    ||    " __""    |      | |    | |   |  | |  |        ||
    ||   "  . _|     |      | |    | |   `-( |  | ()     ||
    ||    _    |  __""__ _|              |  |            ||
    ||     " __""                        `__'            ||
    ||                                                   ||
    `=====================================================`
```

Figure 5: A sample run of the OS on QEMU

Even though the screenshot may not look like much, there is actually a lot going on here. The very fact that this screenshot exists means that the terminal and print functions have been successfully implemented. The text color could also be changed using the available colors provided by x86 VGA mode.

The first line shows that the OS has booted in Intel x86 16-bit real mode and successfully loaded into protected mode. This means that we can now access the full capabilities of an x86 computer.

The terminal also claims that the heap and interrupt descriptor table have been successfully initialized. We can prove these claims. Firstly, let's prove that the heap has indeed been successfully initialized. In this case, the OS also has paging enabled. Some memory was attempted to be allocated using the "kmalloc" function (see Fig. 6).

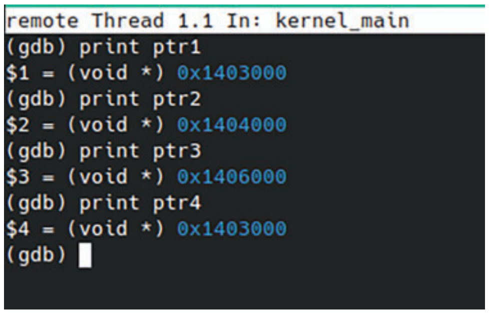


```
void* ptr1 = kmalloc(50);
void* ptr2 = kmalloc(5000);
void* ptr3 = kmalloc(5000);

kfree(ptr1);

void* ptr4 = kmalloc(50);
```

Figure 6: Running kmalloc and kfree on the kernel heap

In Fig. 6, we can see that ptr1 requests 50 bytes. Since a heap block is 4096 bytes, we expect 4096 bytes to be allocated to ptr1. ptr2 request 5000 bytes. This means that 8192 bytes should be allocated to ptr2 and subsequently, 8192 bytes to ptr3.

Now, kfree() was called on ptr1 which marks the block that ptr1 was occupying as free memory. Therefore, we can expect ptr4 to be allocated 4096 bytes on the exact same block that ptr1 was. The debugging results are shown in Fig. 7.

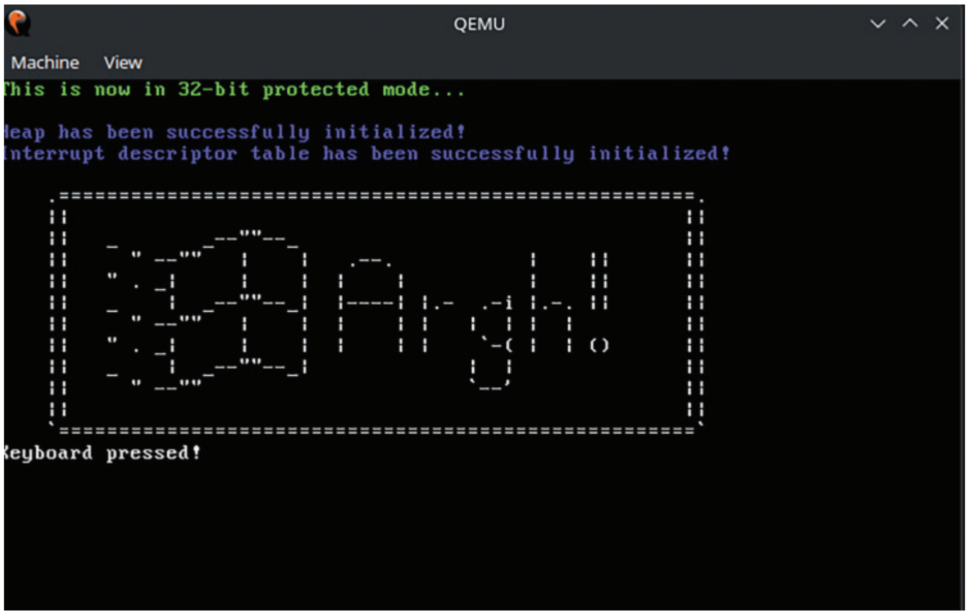


```
remote Thread 1.1 In: kernel_main
(gdb) print ptr1
$1 = (void *) 0x1403000
(gdb) print ptr2
$2 = (void *) 0x1404000
(gdb) print ptr3
$3 = (void *) 0x1406000
(gdb) print ptr4
$4 = (void *) 0x1403000
(gdb) █
```

Figure 7: Results of running kmalloc and kfree on the heap

This is exactly what was expected. 4096 (0×1000) bytes was allocated to ptr1 at virtual address of 0×1403000. The virtual address is due to paging. The physical address is actually 0×1000000. The rest of the pointers were allocated and freed exactly as expected.

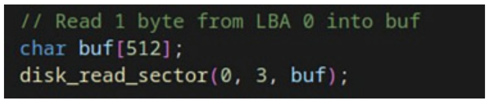
Now, we look at the interrupt descriptor table. A keyboard interrupt was created as entry 0×21 on the interrupt descriptor table. When a keypress is detected, the interrupt is triggered and the text "Keyboard pressed!" should become visible on the terminal (see Fig. 8).



```
This is now in 32-bit protected mode...

Heap has been successfully initialized!
Interrupt descriptor table has been successfully initialized!

Keyboard pressed!
```

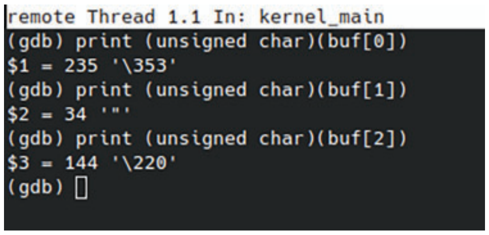
Figure 8: Keyboard press detected as interrupt routine

Basic HDD Serial Advanced Technology Attachment (SATA) I/O has also been implemented. We can attempt to read 3 bytes from the virtual disk (virtual because we are not using a real HDD) with the code snippet depicted in Fig. 9. By using the GDB debugger, we can see that these 3 bytes have been read into the buffer (see Fig. 10).



```
// Read 1 byte from LBA 0 into buf
char buf[512];
disk_read_sector(0, 3, buf);
```

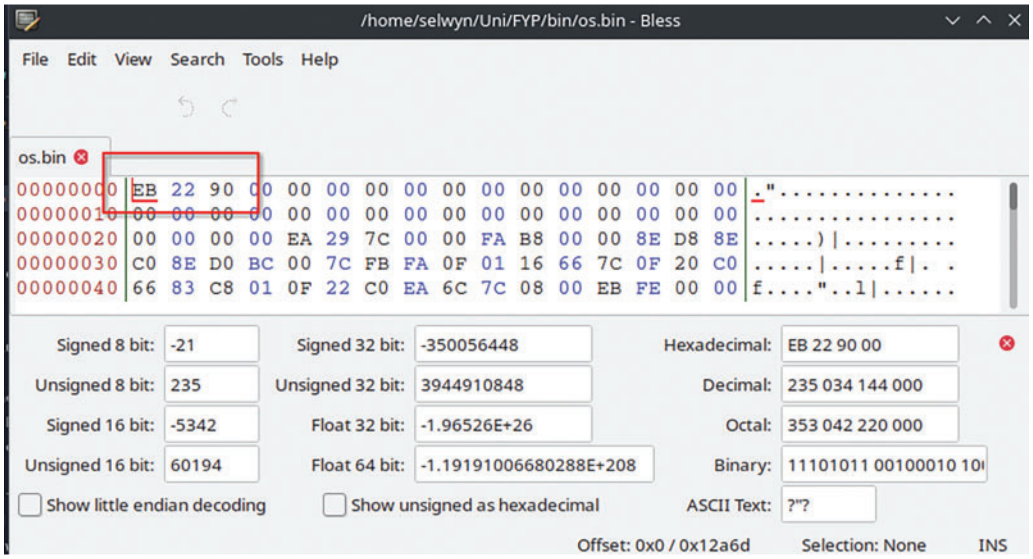
Figure 9: Attempting to read 3 bytes from the Logical Block Addressing (LBA) 0 on the virtual HDD



```
remote Thread 1.1 In: kernel_main
(gdb) print (unsigned char)(buf[0])
$1 = 235 '\353'
(gdb) print (unsigned char)(buf[1])
$2 = 34 '"'
(gdb) print (unsigned char)(buf[2])
$3 = 144 '\220'
(gdb) 
```

Figure 10: The first 3 bytes of the buffer after reading 3 bytes from the virtual HDD

We can attempt to verify this by using a hex editor to examine the memory of the virtual HDD. Using blesses, we can inspect 0 × 00 on os.bin which corresponds to LBA 0 (see Fig. 11).



```
00000000 EB 22 90 00 00 00 00 00 00 00 00 00 00 00 00 00 ."..............
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000020 00 00 00 00 EA 29 7C 00 00 FA B8 00 00 8E D8 8E .....)|.........
00000030 C0 8E D0 BC 00 7C FB FA 0F 01 16 66 7C 0F 20 C0 .....|.....f|. .
00000040 66 83 C8 01 0F 22 C0 EA 6C 7C 08 00 EB FE 00 00 f...."..l|......
```

Signed 8 bit: -21
Unsigned 8 bit: 235
Signed 16 bit: -5342
Unsigned 16 bit: 60194
Signed 32 bit: -350056448
Unsigned 32 bit: 3944910848
Float 32 bit: -1.96526E+26
Float 64 bit: -1.19191006680288E+208
Hexadecimal: EB 22 90 00
Decimal: 235 034 144 000
Octal: 353 042 220 000
Binary: 11101011 00100010 10
ASCII Text: ?"?
Show little endian decoding
Show unsigned as hexadecimal
Offset: 0x0 / 0x12a6d
Selection: None
INS

Figure 11: The contents of the first 3 bytes of the virtual HDD

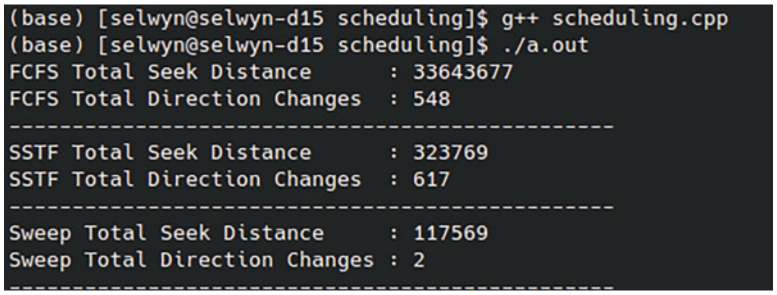
By looking at the first 3 bytes, we see the hexadecimal 0 × EB, 0 × 22, and 0 × 90. Translating them to unsigned decimals gives the following output:

- 0 × EB : 235
- 0 × 22 : 34
- 0 × 90 : 144

This is exactly what was read to the buffer. This proves that the OS can indeed read/write from HDDs.

4.2 I/O Scheduling Algorithms Prototype

For the implementation of simple I/O scheduling algorithm prototypes, Fig. 12 shows the results.



```
(base) [selwyn@selwyn-d15 scheduling]$ g++ scheduling.cpp
(base) [selwyn@selwyn-d15 scheduling]$ ./a.out
FCFS Total Seek Distance      : 33643677
FCFS Total Direction Changes  : 548
------------------------------------------------
SSTF Total Seek Distance      : 323769
SSTF Total Direction Changes  : 617
------------------------------------------------
Sweep Total Seek Distance     : 117569
Sweep Total Direction Changes : 2
------------------------------------------------
```

Figure 12: The simulation of 3 common I/O scheduling algorithms on 100,000 I/O operations

This is just a simple simulation that does not assume much nor does it test the actual stringent running environments of HDDs in real life. However, we can easily see some differences between these 3 algorithms. First, First-Come-First-Serve (FCFS) is very slow on paper. However, it may theoretically have the lowest latency and tail latency relative to the other algorithms. This can make it suitable for cloud service providers. Some major cloud service providers do indeed use such scheduling. Shortest-Seek-Time-First (SSTF) seems much more efficient at first glance. However, it switches the direction of the head very often. This causes significant latency overhead that cannot be captured by these simple prototypes. It would consider this algorithm to be a beginner's trap because it may look good on paper but really causes massive latency and tail latency. This is because, if the HDD continuously has I/O commands coming in, the furthest LBA operations from the current head LBA may take very long to execute or worse, never execute. This is a significant problem for cloud service providers. However, if low-latency I/O is not desired in some number crunching and big data tasks, this algorithm may provide high throughput.

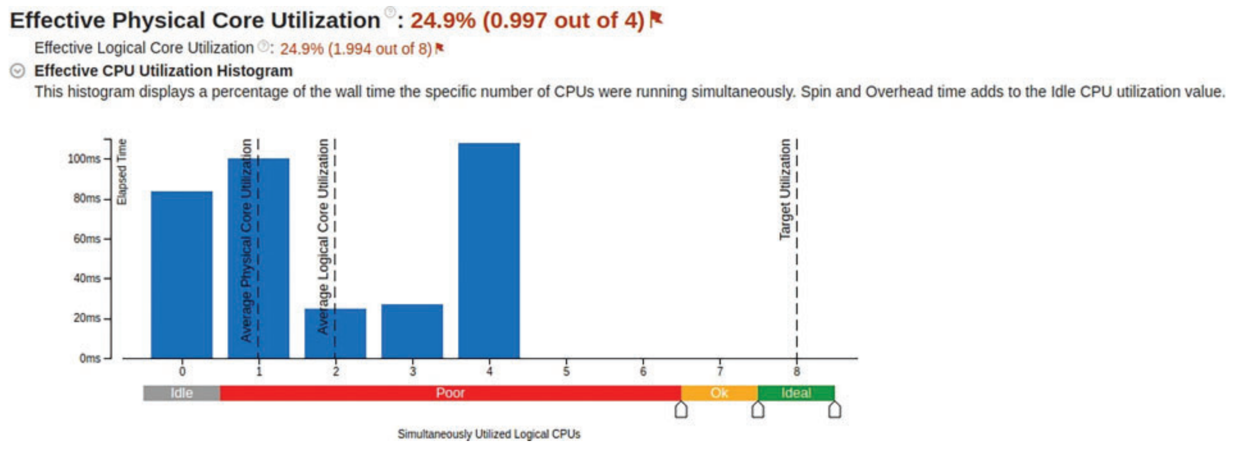
Finally, we have the Sweeping algorithm. This algorithm is my modification of the elevator algorithm. Most people naively believe that sweeping an HDD both ways is better for performance. Unfortunately, the opposite is true. Most enterprise HDDs sold today use a variation of the sweeping algorithm. The sweeping algorithm minimizes the impact of inertia by reducing HDD head direction changes. They also attempt to reduce latency and tail latency by sweeping the regions in the HDD evenly. The fault with the elevator algorithm is that by sweeping both ways, some I/O operations have practically doubled latency and tail latency.

4.3 MapReduce Prototype

The MapReduce prototype [29] is a simple prototype to count the number of words in a large text file. This is a classical Big Data problem. This prototype was implemented in MPI. The prototype aims to efficiently count the words in a large text file by splitting the files between distributed computers, counting the words separately, and then merging them all together.

By running the program locally on a 4-core i5-10210U Intel processor, it was able to count the words in all the works of Shakespeare in approximately 0.04 s. This program is designed to be scalable using a network of distributed computers in a Big Data configuration.

The top 20 words written by Shakespeare are shown in Fig. 13. We can also look at a graph of the effective CPU utilization also obtained using VTune as depicted in Fig. 14.



Effective Physical Core Utilization: 24.9% (0.997 out of 4)

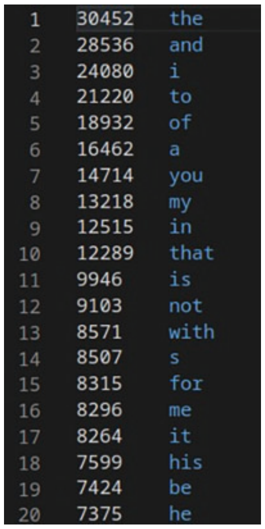
Effective Logical Core Utilization: 24.9% (1.994 out of 8)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

Elapsed Time
100ms
80ms
60ms
40ms
20ms
0ms
Average Physical Core Utilization
Average Logical Core Utilization
Target Utilization
0
1
2
3
4
5
6
7
8
Idle
Poor
Ok
Ideal
Simultaneously Utilized Logical CPUs

Figure 13: Top 20 most frequent words used by William Shakespeare in all his published work



```
30452   the
28536   and
24080   i
21220   to
18932   of
16462   a
14714   you
13218   my
12515   in
12289   that
9946    is
9103    not
8571    with
8507    s
8315    for
8296    me
8264    it
7599    his
7424    be
7375    he
```

Figure 14: Effective CPU utilization of the MapReduce prototype

There is clearly still a lot of room for improvement and scalability. Ideally, the "Effective Physical Core Utilization" should approach 100%. This program has only achieved 25% so far. However, these results may be unreliable as the application execution time was too short to perform reliable profiling. Further testing using much larger datasets and several computers connected through an external network may be required in order to perform actual real-world performance evaluation [30].

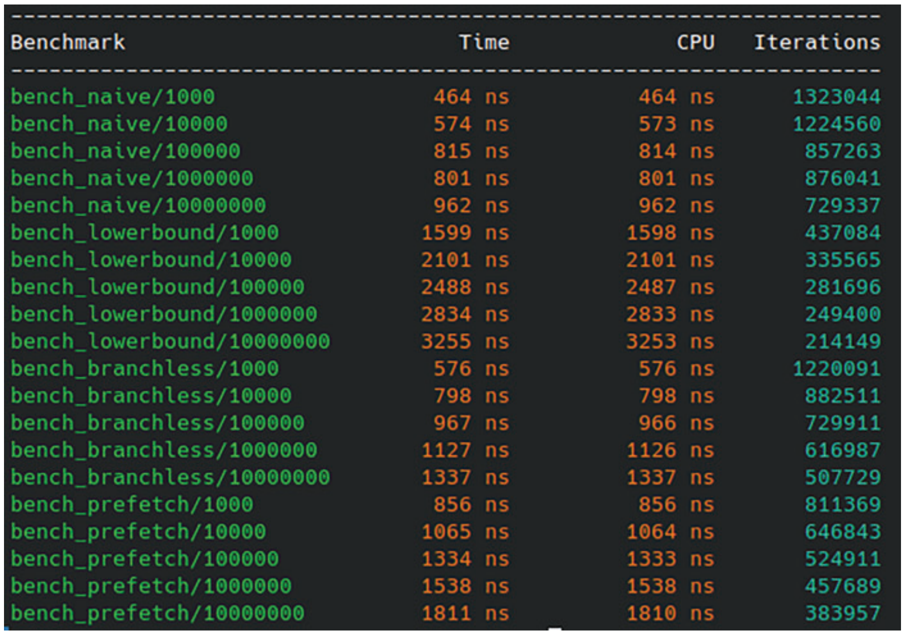
4.4 Binary Search Optimization

We have also implemented 4 different binary search operations to look for better data recovery. These algorithms are (a) Naive algorithm, (b) std: lower bound, (c) Branchless algorithm, and (d) Branchless with prefetch.

Some benchmarks with Google benchmarks can be observed in Fig. 15. Each iteration searches for 9 elements in the array. The number beside the benchmark name represents the search array size.

Unfortunately, the results were unexpected, and thus further examination was halted. The expectation was that each benchmark type should be faster than the one preceding it. There seems to be some problem with the benchmarks and further examination will be required. However, assuming that everything went as expected, we could still further improve the performance of binary search by improving the data layout of the dataset. An ordinary sorted dataset would look like Fig. 16.

However, through better data management, we could instead separate the data according to "hot" data and "cold" data. Doing so should produce a dataset that looks like in Fig. 17. This layout takes advantage of data locality. For example, searching for 15 would very likely load 23 into the CPU cache as well, reducing cache misses.



```
--------------------------------------------------------------------
Benchmark                          Time             CPU   Iterations
--------------------------------------------------------------------
bench_naive/1000                 464 ns          464 ns      1323044
bench_naive/10000                574 ns          573 ns      1224560
bench_naive/100000               815 ns          814 ns       857263
bench_naive/1000000              801 ns          801 ns       876041
bench_naive/10000000             962 ns          962 ns       729337
bench_lowerbound/1000           1599 ns         1598 ns       437084
bench_lowerbound/10000          2101 ns         2101 ns       335565
bench_lowerbound/100000         2488 ns         2487 ns       281696
bench_lowerbound/1000000        2834 ns         2833 ns       249400
bench_lowerbound/10000000       3255 ns         3253 ns       214149
bench_branchless/1000            576 ns          576 ns      1220091
bench_branchless/10000           798 ns          798 ns       882511
bench_branchless/100000          967 ns          966 ns       729911
bench_branchless/1000000        1127 ns         1126 ns       616987
bench_branchless/10000000       1337 ns         1337 ns       507729
bench_prefetch/1000              856 ns          856 ns       811369
bench_prefetch/10000            1065 ns         1064 ns       646843
bench_prefetch/100000           1334 ns         1333 ns       524911
bench_prefetch/1000000          1538 ns         1538 ns       457689
bench_prefetch/10000000         1811 ns         1810 ns       383957
```

Figure 15: Google benchmark results of the binary search algorithms

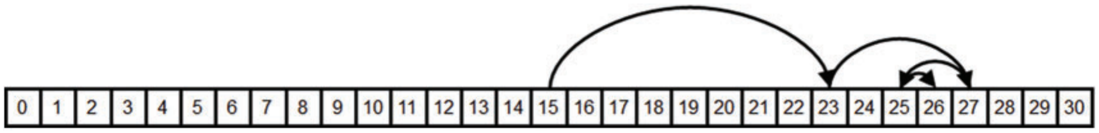


Figure 16: How an ordinary sorted dataset would look like. The arrow represents a sample binary search being performed

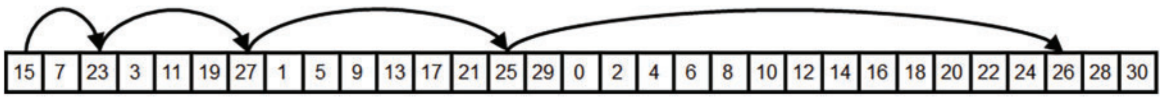


Figure 17: A dataset sorted by "hot" and "cold" data

5 Conclusion

In conclusion, the aim of the paper is to develop an x86 operating system. The operating system features its own BIOS bootloader, memory management, paging, SATA driver, big-data I/O scheduling prototypes, data-oriented design, and a MapReduce prototype.

The intention is to be able to create our own system from scratch without any dependencies from other vendors and possess full control over the technology stack. With the ability to create a big-data-oriented OS with insights from real-time systems, we can even create embedded systems OS suited for big-data purposes. This is ideal as there has been a massive rise in IoT technology in the past decade.

There have also been massive pushes for performance engineering in Computer Science, especially within the last 15 years, since CPU hardware advances are no longer as fast as they used to be [31]. Performance engineering has been slowly dripping into Data Science as most data scientists are simply unaware of this entire field. However, there have been some major leaps in very recent times. One of the large steps forward is the very recent introduction of the Mojo programming language, a language for Data Science with performance engineering directly incorporated into the language.

Acknowledgement: We would like to express our sincere gratitude to the anonymous reviewers for their insightful comments and constructive feedback, which have significantly improved the quality of this manuscript.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Selwyn Darryl Kessler; data collection: Selwyn Darryl Kessler; analysis and interpretation of results: Selwyn Darryl Kessler and Kok-Why Ng; draft manuscript preparation: Selwyn Darryl Kessler, Kok-Why Ng and Su-Cheng Haw. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data to be used in this paper is self-created with the purpose of simulating the efficiency of the 3 common I/O scheduling algorithms.

Ethics Approval: Not applicable.

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

References

[1] W. Wu *et al.*, "Autonomous crowdsensing: Operating and organizing crowdsensing for sensing automation," *IEEE Trans. Intell. Veh.*, vol. 9, no. 3, pp. 4254–4258, 2024. doi: 10.1109/TIV.2024.3355508.

[2] K. Volkov, "Vectorized numerical algorithms to solve internal problems of computational fluid dynamics," *Algorithms*, vol. 17, no. 2, pp. 50, 2024. doi: 10.3390/a17020050.

[3] M. Malami Idina, "The concept of big data and solutions of cloud computing," *Int. J. Adv. Eng. Manag. Res.*, vol. 8, no. 2, pp. 99–106, 2023. doi: 10.51505/ijaemr.2023.8210.

[4] A. K. Sandhu, "Big data with cloud computing: Discussions and challenges," *Big Data Min. Anal.*, vol. 5, no. 1, pp. 32–40, 2022. doi: 10.26599/BDMA.2021.9020016.

[5] B. Berisha, E. Mëziu, and I. Shabani, "Big data analytics in cloud computing: An overview," *J. Cloud Comput.*, vol. 11, no. 1, pp. 62, 2022. doi: 10.1186/s13677-022-00301-w.

[6] S. Mazumdar and S. Dhar, "Hadoop as big data operating system–The emerging approach for managing challenges of enterprise big data platform," in *2015 IEEE 1st Int. Conf. Big Data Comput. Serv. Appl., BigDataService 2015*, 2015. doi: 10.1109/BigDataService.2015.72.

[7] Y. Mei and F. Fu, "Secure big data computing based on trusted computing and key management," *Adv. Intell. Syst. Comput.*, 2021. doi: 10.1007/978-3-030-62743-0_70.

[8] M. Prince and P. M. Joe Prathap, "A novel approach to design distribution preserving framework for big data," *Intell. Autom. Soft Comput.*, vol. 35, no. 3, pp. 2789–2803, 2023. doi: 10.32604/iasc.2023.029533.

[9] P. Pang *et al.*, "Async-Fork: Mitigating query latency spikes incurred by the fork-based snapshot mechanism from the OS Level," in *Proc. VLDB Endow.*, 2023. doi: 10.14778/3579075.3579079.

[10] X. Zhang and Q. Zhu, "HiCa: Hierarchical cache partitioning for low-tail-latency QoS over emergent-security enabled multicore data centers networks," in *IEEE Int. Conf. Commun.*, 2020. doi: 10.1109/ICC40277.2020.9148825.

[11] E. Asyabi, E. Sharafzadeh, S. A. SanaeeKohroudi, and M. Sharifi, "CTS: An operating system CPU scheduler to mitigate tail latency for latency-sensitive multi-threaded applications," *J. Parallel Distrib. Comput.*, vol. 133, no. 4, pp. 232–243, 2019. doi: 10.1016/j.jpdc.2018.04.003.

[12] K. Benaissa, S. Bitam, and A. Mellouk, "On-board data management layer: Connected vehicle as data platform," *Electronics*, vol. 10, no. 15, pp. 1810, 2021. doi: 10.3390/electronics10151810.

[13] V. Gupta and N. Tyagi, "Operating system-concept and comparison," *J. Oper. Syst. Dev. Trends*, vol. 7, no. 2, pp. 24–30, 2020. doi: 10.37591/joosdt.v7i2.2561.

[14] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating Systems Concepts Essentials*, 2nd ed. John Wiley & Sons Inc., 2013.

[15] P. K. Mahato and A. Narayan, "QMine: A framework for mining quantitative regular expressions from system traces," in *Companion 2020 IEEE 20th Int. Conf. Softw. Qual., Reliab., Secur., QRS-C 2020*, 2020. doi: 10.1109/QRS-C51114.2020.00070.

[16] A. Abdelli, "Time distance-based computation of the DBM over-approximation of preemptive real-time systems," *J. Logical. Algebr. Methods Program.*, vol. 136, no. 3, pp. 100927, 2024. doi: 10.1016/j.jlamp.2023.100927.

[17] J. Adelt, J. Gebker, and P. Herber, "Reusable formal models for concurrency and communication in custom real-time operating systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 26, no. 2, pp. 229–245, 2024. doi: 10.1007/s10009-024-00743-4.

[18] A. Thirion, N. Combes, B. Mulliez, and H. Tap, "BCG-VARS: BallistoCardioGraphy vital algorithms for real-time systems," *Biomed. Signal Process. Control*, vol. 87, no. 8, pp. 105526, 2024. doi: 10.1016/j.bspc.2023.105526.

[19] T. Komori, Y. Masuda, and T. Ishihara, "Virtualizing DVFS for energy minimization of embedded dual-os platform," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. E107.A, no. 1, pp. 3–15, 2024. doi: 10.1587/transfun.2023KEP0002.

[20] Y. Ye, Z. Nie, X. Liu, F. Xie, Z. Li and P. Li, "ROS2 real-time performance optimization and evaluation," *Chin. J. Mech. Eng.*, vol. 36, no. 1, pp. 1, 2023. doi: 10.1186/s10033-023-00976-5.

[21] M. Hassan, "DISCO: Time-compositional cache coherence for multi-core real-time embedded systems," *IEEE Trans. Comput.*, vol. 72, no. 4, pp. 1163–1177, 2023. doi: 10.1109/TC.2022.3193624.

[22] N. Mrewa, A. Mohd Ramly, A. Amphawan, and T. K. Neo, "Optimizing medical iot disaster management with data compression," *J. Inform. Web Eng.*, vol. 3, no. 1, 2024. doi: 10.33093/jiwe.

[23] J. Stój and A. Kwiecień, "Temporal costs of computing unit redundancy in steady and transient state," *Studies Comput. Intell.*, vol. 733, pp. 1–14, 2018. doi: 10.1007/978-3-319-65208-5_1.

[24] H. Baek, K. G. Shin, and J. Lee, "Response-time analysis for multi-mode tasks in real-time multiprocessor systems," *IEEE Access*, vol. 8, pp. 86111–86129, 2020. doi: 10.1109/ACCESS.2020.2992868.

[25] B. Sun, T. Kloda, S. A. Garcia, G. Gracioli, and M. Caccamo, "Minimizing cache usage with fixed-priority and earliest deadline first scheduling," *Real-Time Syst.*, pp. 1–40, 2024. doi: 10.1007/s11241-024-09423-7.

[26] A. M. Cheng, "Work in progress: Response time analysis of real-time quantum computing systems," in *IEEE 29th Real-Time Embed. Technol. Appl. Symp. (RTAS)*, IEEE, 2023, pp. 329–332. doi: 10.1109/RTAS58335.2023.00033.

[27] B. Sun, D. Roy, T. Kloda, A. Bastoni, R. Pellizzoni, and M. Caccamo, "Co-Optimizing cache partitioning and multi-core task scheduling: exploit cache sensitivity or not?" in *2023 IEEE Real-Time Syst. Symp. (RTSS)*, IEEE, 2023, pp. 224–236. doi: 10.1109/RTSS59052.2023.00028.

[28] M. Howard and R. I. Bruce, "ESP32: QEMU emulation within a docker container," in *Proc. Future Technol. Conf.*, Cham, Springer Nature Switzerland, 2023, pp. 63–80. doi: 10.1007/978-3-031-47457-6_5.

[29] D. Uprety, D. Banarjee, N. Kumar, and A. Dhiman, "MapReduce: Big data maintained algorithm empowering big data processing for enhanced business insights," In: *ICT: Applications and Social Interfaces*, Springer Nature Singapore, 2024, pp. 299–309.

[30] F.F. Chua, T.Y. Lim, B. Tajuddin, and A.P. Yanuarifiani, “Incorporating semi-automated approach for effective software requirements prioritization: A framework design,” *J. Inform. Web Eng.*, vol. 1, no. 1, pp. 1–15, 2022, doi: 10.33093/jiwe.2022.1.1.1.

[31] S. Matsuoka, J. Domke, M. Wahib, A. Drozd, and T. Hoefler, “Myths and legends in high-performance computing,” *Int. J. High Perform. Comput. Appl.*, vol. 37, no. 3–4, pp. 245–259, 2023. doi: 10.1177/10943420231166608.