# REDUCING IMPROPER MEMORY USAGES IN THE INTEREST OF EFFICIENTLY MANAGING MEMORY

Name: Abhishek Parekh
Course: Operating Systems

University/Organization: Syracuse University
Location: Syracuse, NY, USA
Email: apparekh@syr.edu

Date: Nov 29th, 2022

***Abstract*—An operating system's ability to handle or manage primary memory and switch between main memory and the storage during the execution of processes is defined as memory management. Regardless of whether a memory location is free or assigned to a process, memory management keeps track of every memory location. It determines how much memory should be set aside for processes. It selects which processes will receive memory at what intervals. It keeps track of when memory is released or unallocated and changes the status accordingly. This paper dives into the summary of findings from varies research papers and other sources. Furthermore, it analyzes how well such simulations have been conducted and offers improvements for the betterment of the techniques. A solution is proposed to derived more creative and innovative ideas, and later implements a technique discovered from the sources.**

***Keywords—memory management, fragmentation issues, internal fragmentation, internal fragmentation, efficient memory usage, processes, best fit, worst fit, first fit, higher priority, lower priority, execution time, allocation, deallocation, efficient memory utilization, partitioning, free space, memory blocks, primary memory, secondary memory, memory segment, swapping, paging, nonvolatile memory, DRAM, pointers, new, delete, low memory killer, out of memory killer, memory shortage, thrashing, mobile systems memory, reclamation, small memory capacity, cache efficient, NVRAM, Cache line counters, virtual memory protection, novel memory allocator, Network on chip, memory architecture, multi-layer.***

## I. INTRODUCTION

Memory management is essential for multiprogramming success and efficient memory use [1]. The efficiency of each algorithm varies depending on the circumstance, and there are several memory management techniques reflecting different approaches [1]. This paper explores the topic of memory management of processes within the operating system and attempts to solve the problem of how to efficiently maintain memory space for processes and how to properly utilize the main memory to minimize fragmentation issues.

## II. SUMMARY OF FINDINGS CONDUCTED THROUGH RESEARCH THE PAPERS

### *A. Memory Management in Operating system*

The operating system occupies a small portion of memory in a multiprocessing computer, whereas other portions are shared by many applications [1]. Memory management is the activity of dividing the memory across many processes [1]. The operating system utilizes memory management as a technique to regulate system and main memory accesses while the processes are running [1]. The efficient use of memory is the primary goal of memory management.

There are two types of loaders that load the process onto the main memory, static loading, and dynamic loading. Statis loading is where it loads the entire program into a fixed address, this can lead to higher memory usage [1]. Whereas dynamic loading stores the entire program and its data used in the processes into a physical memory, waiting for the process to be called and executed [1]. This leads to better memory performance, since the routine is not loaded until it is called

Swapping is the process of briefly moving a process from the main memory into secondary memory, which is quicker than secondary memory [1]. More processes may run and fit into memory at once due to the advantage of swapping processes. Transferring time makes up most of the swapping, and the quantity of memory exchanged directly relates to the overall time [1]. Moreover, Swapping is also referred to as "roll-out, roll-in" because the memory management can swap out the lower priority process and then load and run the higher priority process if one with a higher priority arrives and sends a request for execution [1]. Later, the lower priority process is flipped back into memory and will continue into the execution phase after the higher priority activity has finished executing [1].

Multiple partition allocation is a method where a process is chosen and loaded into the free partition from the input queue [1]. This partition will later become accessible to other processes when the procedure finishes [1].

Fixed partition allocation is a method where the OS keeps a table that lists the portions of memory that are free and those that are being used by processes, this helps in tracking used and unused memory addresses [1]. We look for a free block big enough to hold this process when it comes along and demands memory [1]. We allocate memory to process if the need is met; otherwise, we maintain the remaining space available to accommodate future requests [1].

Some solutions include first fit, best fit, and worst fit. In first fit, the first free space that fulfils the requirement is filled by the process [1]. In best fit, it finds the smallest

memory block that fulfils the memory requirements, this is accomplished by searching through the entire memory list [1]. In worst fit, it finds the largest available free space and fills it with the process, this is the most inefficient method since many memory blocks will not be utilized [1].

Fragmentation is explained as the small free block created when a process is loaded into memory, executed, and then deleted [1]. Since they're not merged or do not meet the process's memory requirements, these blocks cannot be given to new processes [1]. Our main goal is to reduce memory waste or fragmentation issues to accomplish proper memory utilization.

Two forms of fragmentation exist, internal fragmentation and external fragmentation. Internal fragmentation occurs when much more memory is allocated than requested, leading to unused space [1]. External fragmentation is when there is a free memory block, however a process cannot be allocated to that block since free memory cannot be contiguous, thus resulting in an external fragmentation, which separates it from other free memory spaces that are not adjacent to each other [1].

To resolve fragmentation, compaction can be used, it is a technique where all free memory spaces are combined and makes one large block of memory, resulting in processes to be allocated to that block and ending any fragmentation [1]. Furthermore, we can allow the logical address space of the processes to be noncontiguous, this authorizes a process to be allocated anywhere in the physical memory where available [1].

### *B. Operating System – Memory Management*

More memory can be accessed by a computer than is present on the system. This additional memory is referred to as virtual memory, and it is a region of a hard drive that is configured to simulate the RAM of a computer [2]. As part of the paging memory management strategy, the process address space is divided into pages, which are identical-sized blocks, the size is power of 2, between 512 bytes and 8192 bytes [2]. The number of pages in the procedure is used to gauge its magnitude, consequently, maximizing the use of the main memory and prevent external fragmentation, main memory is divided into small fixed-sized blocks of (physical) memory called frames. The size of a frame is kept the same as that of a page [2].

Although paging lessens internal fragmentation, outward fragmentation still exists [2]. Paging is a widely accepted and easy-to-use memory management strategy [2]. Swapping is made incredibly simple by the pages and frames being the same size [2]. Page tables demand more memory; therefore, they might not be ideal for systems with limited RAM [2].

A memory management approach called segmentation divides each task into several smaller segments, one for each module that comprises components that carry out related tasks [2]. Each portion of the program is essentially a distinct logical address space [2].

A computer segment includes the primary function, as well as support functions, data structures, and other elements [2]. Every process has a segment map table that the operating system keeps track of, and it also keeps a list of free memory blocks with the segment numbers, sizes, and matching main memory addresses [2]. Both the segment's initial address and its length are retained in memory. A segment identification value and an offset are included in a reference to a memory location [2].

### *C. C++ memory management: new and delete*

The memory of a variable or an array can be allocated during runtime thanks to C++, the actual event is called Dynamic memory allocation is what is happening here [3]. Variable memory allocation is automatically managed by the compiler in other programming languages like Java and Python. But in C++, this is not the case [3].

After we have finished using a variable, we must manually deallocate the memory that was dynamically created in C++ [3]. Using operators new and delete, we can allocate and deallocate memory. Pointers needs to be utilized to point towards an address where memory allocation needs to occur, the new operator returns the address of the memory location or the first element of the array.

We can deallocate the memory occupied by a dynamically declared variable after we are no longer required to use it. Using the delete operator, it returns the memory to the OS, this process is known as memory deallocation [3].

### *D. Memory Management*

The extent of the storage needed to run a programme was known at compile time because to the design of previous programming languages e.g., Fortran [4]. The stack, a primary data structure that is dynamically controlled, was used to create subsequent languages e.g., Algol 60 [4]. Information is put onto a stack, utilizing memory space that may later be reclaimed by popping [4]. The unpredictable nature of computations makes memory management solutions uniquely problematic [4]. Since the halting issue is intractable, it follows that it is impossible to predict in advance how many ceils will be required to complete a complex calculation [4]. Reclamation of unused memory blocks is done automatically using a runtime process called garbage-collection, whereas languages like C, C++, LISP, & Prolog require that memory be allocated and deallocated to manage the efficiency of the program.

There are two methods for carrying out storage reclamation: one is incremental, in which the implementor chooses to combine the collecting task with the actual computation; the other is to wait until all the memory is used up before starting the laborious operation of identifying useless cells and making them available for future use [4]. A perfect reclamation results in all useless blocks will be reclaimed

and no used cells will be mistakenly freed to be reclaimed [4].

### E. *Hardware Memory Management for Future Mobile Hybrid Memory Systems*

The memory footprints of current mobile applications are expanding quickly, which presents a significant challenge for memory system design [5]. Frequent data swaps between memory and storage caused by insufficient DRAM main memory impacted performance, used energy, and reduced write endurance of standard flash storage systems [5]. This results in bigger DRAM storages to have higher risk of leakage power, leading to loss in battery life and higher thermals, degrading the quality of the device [5].

These problems may be resolved by emerging nonvolatile memory (NVM), which has a larger capacity per dollar than DRAM and requires less static power [5]. A wide range of NVM technologies, including as 3D XPoint, memristors, and phase-change memories (PCM), have recently come into existence [5]. NVM has a greater access latency than DRAM despite the benefits noted, and NVM writes can have higher latencies and wear costs [5]. As can be seen, that it requires an integration of different types of memory system, an architectural design which would lead to robust usages for mobile devices.

In comparison to an unrealistic and unscalable AllDRAM architecture, the adaptive AdpComb strategy successfully cuts energy by 40% while only losing performance by a modest 12% [5]. Between the PageMove and the StatComb block migration policies, AdpComb attempts to select the best option [5].

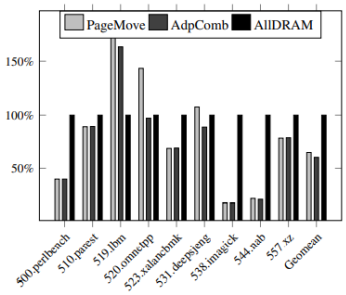


Fig. 1. Comparison of energy usage from [5]

### F. *Virtual Memory Partitioning for Enhancing Application Performance in Mobile Platforms*

Due to their portability, mobile devices, unlike servers and desktop computers, do not have a memory slot that may increase the memory capacity [6]. In mobile systems, memory management techniques like low memory killer (LMK) and out-of-memory killer (OOMK) are frequently employed [6]. Since memory is of concern when the amount of physical memory accessible runs out, the OS will forcefully end programs. Due to the lack of memory, it can lead to thrashing and fragmentation, as a result tremendously slowing down the performance of the device [6]. There is a page reclamation method that has the objective to gather and secure existing free memory, however, this can lead to the device becoming much more sluggish in terms of user responsiveness [6]. Thus, it is crucial manage memory on mobile devices with smaller memory capacities.

The main goal is to reduce and eliminate as much deterioration of existing application's performance induced by low memory killer and out-of-memory killer. As a result, to have improvements in application execution time whilst running on low memory.

A technique called Virtual Memory Node was implemented to overcome the disadvantages of low memory killer and out-of-memory killer

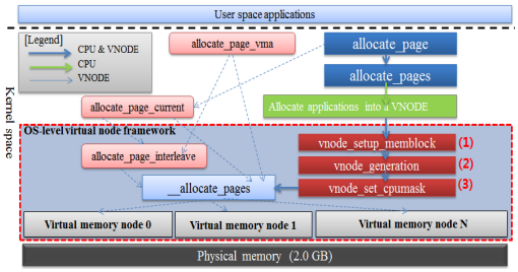


Fig. 2. Implementation of VNODE from [6]

VNODE consists mainly of three components, they include vnode setup memblock, vnode generation, and vnode set cpumask [6]. The design architecture is proposed mainly for mobile devices that run on low memory. Through this it gains the advantages of having complete memory isolation and reduce the number of LMK/OOMK processes [6]. Furthermore, thrashing and memory fragmentation are greatly reduced.

### G. *Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory*

The study presented here discusses three key techniques: consistency-preserving updates that are cache-efficient, robust wear-aware memory allocation, and avoiding erroneous writes [7]. By demonstrating a B+-tree implementation that has been changed to fully utilize our toolkit, it is demonstrated through the assessment that these approaches are effectively implementable and efficient [7].

However, to take use of this speed, NVRAM must be directly connected to the memory bus [7]. Applications must make sure that this persistent data is protected against deterioration and corruption with the help of user-space libraries and the OS. Cache line counters, a virtual memory protection mechanism, and a novel memory allocator for NVRAM can significantly simplify the process of developing secure, high-performance permanent data structures for upcoming non-volatile memory [7].

### H. *Architectural Support for Dynamic Memory Management*

Applications built using OOP and the GUI must allocate dynamic RAM frequently [8]. Garbage collection (GC), an automated dynamic memory reclamation function, has since gained popularity in contemporary computer languages. As a result, dynamic storage management can take up to one-third of the total time required to run a program [8]. This demonstrates the fulfilment of requirements for an efficient memory management system.

For a certain heap size, the block size has an impact on the bit-map size. A smaller bit-map size would result from the bigger block size [8]. Lower cost for the bit-map results from a smaller bit-map size. The bigger block size might, however, result in more internal fragmentation during the allocation process. Higher internal fragmentation may result in the worst fit method being implemented (highest memory address allocated), which is a result of fragmentation [8]. Evidently, in the suggested scheme, the highest block allocated is regarded as the memory overhead. This will lead to severe mismanagement in memory, resulting in a decrease in the performance for the OS.

### *I. Memory Architecture and Management in an NoC Platform*

NoC stands for Network-on-Chip, it is a communication infrastructure, that has a shared serial bus, giving it the characteristics of parallelization [9]. Every NoC-based platform design must carefully consider how to organize and manage the memory space [9]. It is suggested that a Data Management Engine (DME) with a robust memory architecture, which is a programmable hardware building block that is a component of each processing unit [9]. By controlling memory access, memory space, and communication across the on-chip network, it offloads the processing element (CPU, DSP, etc.) [9]. Virtual address translation, shared and private memory management, cache coherence protocol, support for memory consistency models, and synchronization and security procedures for shared memory communication are the core duties of the DME [9]. The DME can handle high level data management characteristics like dynamic memory allocation and abstract data types with customizable support since it is totally programmable and configurable [9].

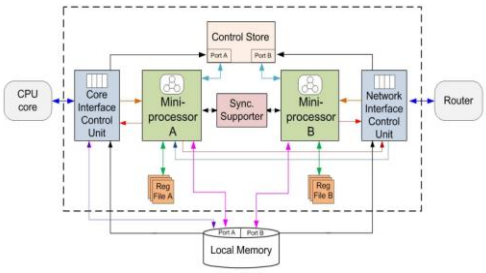


Fig. 3. From [9] shows the Memory Architecture in a NoC platform

The architecture design consists of the Core Interface Control Unit, Network Interface Control Unit, Mini-processor A, Mini-processor B, Synchronization Supporter, and a control store [9]. Features of the DME include dual interfaces and dual processors, cooperation with the interfaces and the mini processors, dual port shared between the control store and the local memory, hardware support for mutex synchronization, and dynamic uploading of data into the control store [9].

### *J. Robust Memory Management using Real Time Concepts*

First it creates a new memory partition with block size 5 and the first task for memory block allocation, releases it and displays the status [10]. Memory blocks with unused space and used space are displayed [10]. It updates every time when a block is allocated with memory, moving it to used memory space counter. Proving how memory allocation is conducted in real time [10].

When there is a lot of memory available for allocation, the search time increases significantly. In this experiment, the allocator rounds up the requested size to a permissible figure and allows the user to choose a larger block from another unoccupied list [10]. The constant time, excellent fit allocator is called TLSF [10]. By processor bit instructions and word size bitmaps, an appropriate list is found [10]. Timing performance is being compared to those of different allocators. According to the author's data, the proportion of fragmentation in the buddy allocator is significantly higher than it is in the TLSF allocator [10]. The findings of the suggested buddy allocator, which was written in the Keil C programming language, demonstrate a significant increase in the reduction of fragmentation as well as constrained execution time without memory space waste.

This study demonstrates how well memory may be managed when it is used as a resource in real-time applications [10].

### *K. Multi-Layer Memory Resiliency*

A multi-layer hardware/software strategy is necessary because memories are prone to deterioration and failure from a variety of manufacturing, operating, and environmental impacts [11]. This technique must be able to withstand, adapt to, and even take advantage of these effects when they arise [11]. Variability and operational stress have a very negative impact on the whole memory hierarchy [11]. This paper first reviews the main memory degradation and failure mechanisms before outlining the difficulties in maintaining dependability throughout the memory hierarchy and outlining research attempts to develop multi-layer memory resilience using a hardware/software strategy [11]. It outlines two main components to visually see how memories can become more resilient at a multi-layer level.

It is demonstrated how software features may be communicated to the architecture to reduce the ageing of huge register files in GPGPUs [11]. It is discussed that static and dynamic approaches are implemented to achieve energy savings in caches via aggressive voltage scaling along with deleting erroneous blocks [11]. In order to improve memory dependability across various abstraction levels, including applications, compilers, run-time systems, and hardware

platforms, the approaches described above can also profit from semantic retention of application intent [11].

## III. Analyzing the Research Conducted

In [1], there are some great insights on how to improve the management of memory and allow the system to work on its best possible state. [1] talks about different types of loaders to give the least burden on memory usage, the swapping technique aids in prioritizing processes therefore clearing the process queue in an efficient manner, the multiple partitioning method that gives access to all processes to the memory block, allocating methods like best fit, first fit, worst fit, and the prevention of fragmentation. Since two forms of fragmentation can occur, internal and external. For both internal and external fragmentation, we can use a technique called compaction, which will essentially free up any unnecessary blocks of memory that may lead to fragmentation issues.

I believe that these methods lead to a better utilization of the main memory, which is the crucial part of any operating system, since all processes of an application or the user's response will travel through the memory. Therefore, it becomes essential for the OS to better manage this region for utmost quality for the system to run on.

Mentioned in [2] about the paging technique, it becomes crucial when more memory is required by the system, thus creating virtual memory, as a result it will maximize the usage of the main memory and avoid creating any unnecessary fragmentation. Furthermore, as discussed in [2] about segmentation, it aids in creating several smaller memory blocks filled with components having assigned similar tasks, basically this cuts down on the load the system has to handle, since smaller memory blocks will have a smaller stack size, resulting in a lightweight execution. Moreover, this will increase execution time and lead to better overall system health of the OS.

In [3], it discusses about how allocating, and deallocating is important in C and C++, whereas languages like Java does not need to worry about garbage collection. The process of allocation and deallocation in a program will help it clear any memory blocks that are unused and have finished execution and are not needed for further calculations. In C++ operators new and delete are used for the allocation and deallocation, alongside with pointers and references, since items stored on addresses need to be grabbed in order to be used by the program.

As discussed in [4] about reclamation of unused memory blocks in languages like C, C++, & LISP. It becomes crucial to gather the unused memory blocks for betterment of organizing the remaining memory, thus it can be better utilized. A proper reclamation technique will not touch any memory blocks that are being used currently, or else it can lead to files getting lost and corrupted.

Memory management becomes utmost important when dealing with devices that have low capacities of memory, as seen in [5]. A few types do exist on mobile devices, such as DRAM, and NVM, however DRAM uses too much energy, having DRAM would simply drain the battery and CPU power from the device, rendering it useless. Therefore, a combination of memory techniques and a robust memory architecture should be used on mobile devices, for the sole purpose of managing the minimum resources available on the device.

Like [5], the research in [6] also discusses about managing memory on portable devices, with non-modular memory slots. It implements a technique called Virtual Memory node to overcome the disadvantages of low memory killer and out of memory killer. Essentially, the goal for VMN is to reduce as much damage it can toward the health of the device and the application, and results in the increase of execution time. VMN basically gives a boost in the capacity of the memory, in terms of virtually. Since hardware wise portable devices cannot simply add more memory cards to improve memory management. This is a great way to improve the performance of a device from being sluggish and prevents the device from creating any fragmentation, which could severely harm the health of the system.

Discussion in [7] talks about how to protect data from getting lost, corrupted, or deteriorated. The implementation consists of cache line counters, virtual memory protection mechanism, and an allocator, all used for the security and high performance of the data on the system. This is a great implementation, since many forget that memory blocks deteriorate over time, and most of the focus tends to be on how to manage memory, rather than securely protect the data. A well protected data will ensure great performance for a longer time, compared to data that is on the verge of being corrupted.

Many applications do require more memory than other applications, as seen in [8], it discusses how to implement a dynamic memory reclamation function. This is very much useful when running applications that are resource hungry such as CAD software, or Adobe premiere. While on other hand, some applications may not need too much memory, such as text editors, instead of forcing too much resources out of the system all the time, the dynamic reclamation function should be used. When using small memory, the user does not want it to be allocated into a bigger memory block.
As can be seen in [9], that a data management engine is crucial for a robust and efficient memory architecture for system. It is a linkage of many components, such as the control unit, processors, local memory, control store, the synchronizer, and the interface unit. They all work together to create an architecture that dynamically processes data and its processes.

Through [10], we get to understand how memory management is of the utmost importance, since data is being processed real time, it must allocate and deallocate memory to provide the program with reasonable response time to send back to the user. Such as medical equipment needs to be working in real time, medical practitioners or surgeons

cannot wait on the device to get a reply, it needs to be working on par with the medical team.

As seen before in [7] about the deterioration of memory, [11] discusses on the same topic but uses a different mechanism. It aims to implement a multi-layer architecture, having the aim to create a much more resilient memory architecture. Having a multi-layer memory system, it aims to reduce the aging of memory blocks on a particular layer, since the data can be stored on many layer instead of one, this ensures that if one layer were to lose the data, the mechanism would simply export the data to a different layer where it can be securely stored and provide high-performance level when running the application, since memory blocks will not be used too often.

## IV. Proposal for a Solution

Solution for a real time system

Create a dynamic memory system that handles data in a storage control and fetches it from the local memory. While having multiple processors and multiple interfaces, it can quickly allocate and deallocate data from memory blocks. A dynamic storage unit would allow for long-lasting high-level performance, preventing any sluggish response from the device. A dynamic architecture will avoid having any fragmentation or thrashing occur, since it is a portable device, it will be having a fixed amount of memory. Therefore having to reclaim the unused memory blocks will prove useful in maintaining quick memory speeds.

## V. Solution Implementation

Implementation of the best fit algorithm

1. Create input memory buffers or can create have one buffer with a size of 2048.
2. Create a certain number of processes, let's say 10 processes, with their own stack size, priority number and its function call.
3. Clear all buffers, resulting them to be free memory blocks, so that it resolves any fragmentation that may have occurred.
4. Create a for loop that iterates through each process and inside the for loop there is a search algorithm, its task is to look for the smallest block of memory that the current process can be allocated to.
    a. Find minimum_block_size for current process
    b. If it isn't smallest best fit, it keeps searching for that memory block
    c. Once found, allocate the current process to that memory block
5. Iterate to the next process in the for loop to search for its best fit.

## References


[1] "Memory management in operating system," *GeeksforGeeks.com*, 16-Nov-2022. [Accessed: 13-Nov-2022]. [Online]. Available: https://www.geeksforgeeks.org/memory-management-in-operating-system/

[2] "Operating system - memory management," *Tutorials Point*. [Accessed: 15-Nov-2022]. [Online]. Available: https://www.tutorialspoint.com/operating_system/os_memory_management.htm

[3] "C++ memory management: New and delete," *Programiz*. [Accessed: 10-Nov-2022]. [Online]. Available: https://www.programiz.com/cpp-programming/memory-management

[4] Y. Bekkers and J. Cohen, "Dynamic Memory Management for Sequential Logic Programming Languages," in *Memory management*, Springer Berlin Heidelberg, 1992, pp. 82–102.

[5] F. Wen, M. Qin, P. V. Gratz, and N. Reddy, "Hardware memory management for future Mobile Hybrid Memory Systems," arXiv.org, 12-Apr-2020. [Accessed: 1-Nov-2022]. [Online]. Available: https://arxiv.org/abs/2004.05518

[6] G. Lim, C. MIN, and Y. I. Eom, "Virtual memory partitioning for enhancing application performance in Mobile Plaforms," *IEEE Xplores*, Nov-2013. [Accessed: 1-Nov-2022]. [Online]. Available: https://ieeexplore.ieee.org/document/6689690

[7] I. Moraru, D. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, Durable, and Safe Memory Management for Byte-addressable Non-Volatile Main Memory," *Semanticsscholar.org*, 03-Nov-2013. [Accessed: 3-Nov-2022]. [Online]. Available: https://semanticscholar.org/paper/387d1eeedf554f07f16ca1bc2daec51dbc224b60

[8] J. Chang, W. Srisa-an, and C.-T. Lo, "Architectural support for dynamic memory management," *IEEE Xplore*, 2000. [Accessed: 2-Nov-2022]. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/878274/

[9] A. Jantsch, X. Chen, A. Naeem, Y. Zhang, S. Penolazzi, and Z. Lu, "Memory Architecture and Management in an NoC platform," *Semanticsscholar.org*, 2012. [Accessed: 1-Nov-2022]. [Online]. Available: https://semanticscholar.org/paper/4dee7c2d91fd976a84f949e37fc55b5505a03e2f

[10] V. Karthikeyan, S. Ravi, and M. Anand, "Robust Memory Management using Real Time Concepts," *Science Publications*, 01-Apr-2014. [Accessed: 2-Nov-2022]. [Online]. Available: https://doi.org/10.3844/jcssp.2014.1480.1487

[11] A. Nicolau, N. Dutt, P. Gupta, A. B. Mofrad, M. Gottscho, and M. Shoushtari, "Multi-layer Memory Resiliency," *Academia.edu*, 31-May-2022. [Accessed: 1-Nov-2022]. [Online]. Available: https://www.academia.edu/50675374/Multi_Layer_Memory_Resiliency