Grenze International Journal of Engineering and Technology, June Issue

Design a Smart Memory Management (SaMM) for Embedded System without MMU

Debasis Behera[1] and Suvendu Narayan Mishra[2]
[1]C V Raman Global University/EEE, Bhubaneswar, India
Email: debasis041@cgu-odisha.ac.in
[2]Veer Surendra Sai University of Technology/Electronics & Telecommunication Engg., Burla, India
Email: snmishra_etc@vssut.ac.in

***Abstract*— Memory utilization is an ongoing significant obstacle in the domain of embedded systems lacking Memory Management Units (MMUs). A novel approach is presented for developing and implementing a tailored Smart Memory Management (SaMM) system intended for implementation in such environments. Memory allocation and utilization are optimized by SaMM through the prioritization of memory blocks based on dynamic requirements and usage patterns. SaMM anticipates that by employing both static and dynamic allocation mechanisms, system performance is going to be enhanced and memory fragmentation is going to be reduced. The proposed approach offers efficient and lightweight memory management capabilities. Moreover, it seamlessly incorporates into contemporary embedded systems without the need for MMU support. SaMM posits that the implementation of both static and dynamic allocation algorithms is going to result in an amelioration of memory fragmentation and an enhancement of system performance. The suggested technique provides memory management features that are both efficient and lightweight. Furthermore, it integrates effortlessly into modern embedded systems, eliminating the requirement for MMU support.**

***Index Terms*— smart memory management, embedded system, memory management unit (MMU), dynamic allocation, static allocation and memory allocation.**

I. INTRODUCTION

Embedded systems are software-enabled computer hardware components featuring microprocessors that are designed to perform a specific function either autonomously or in conjunction with a more complex system. Integrated circuits intended to perform calculations for real-time processes are located at its core.

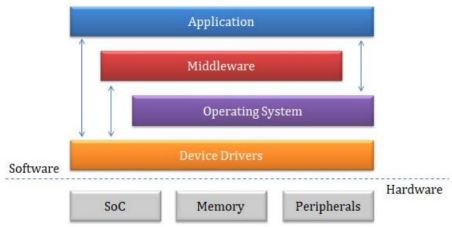

Figure 1. Generic architecture diagram of an embedded system

Grenze ID: 01.GIJET.10.2.832

Fig 1 depicts the generic architecture diagram of an embedded system as shown above. The phrase "Internet of Things" (IoT) denotes a network comprising interconnected mechanical and digital devices. The ability of these devices to transmit data through a network of cloud computing devices is attributed to their unique identifiers (UIDs). Information transfer does not necessitate human-computer interaction [1]. Fig 2 depict the generic architecture diagram of an IoT system as shown below.

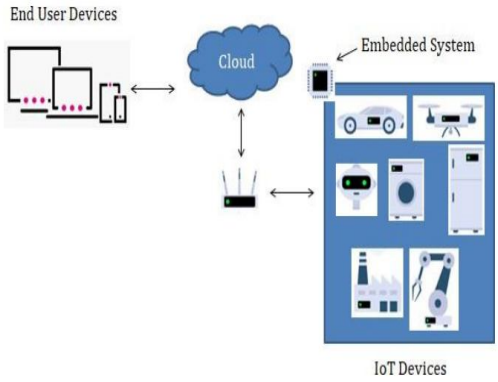

Figure 2. Generic architecture diagram of an IoT system

The need for severe control in the modern era has given rise to the Internet of Things (IoT), a relatively new field that has expanded exponentially over the past few years. In addition to high-tech medical and defines equipment, commonplace appliances such as stoves, ovens, televisions, and laundry machines are also included in system control. Since embedded systems are required to implement the aforementioned capabilities, the significance of the Internet of Things in enhancing embedded systems has been underscored. IoT developers utilize embedded system microprocessors, which are subsequently integrated into the machines or devices under control, to generate software for the control functions mentioned earlier. The memory and algorithm of a software program are both critical components.

The implementation of constraint-platform systems, which utilize real-time operating systems irrespective of the real-time characteristics of the executing application, necessitates dynamic memory management. It is common practice in these systems to decrease the frequency of malloc() and free() invocations; doing so can significantly streamline the process of safeguarding against memory breaches [2]. Additionally, it can significantly reduce the amount of CPU time consumed by repetitive calls to the malloc() and free() functions. Effective performance is only achievable with dynamic allocations derived statically. However, this may not be applicable to all applications, or it may be constrained by the hardware platform's memory capacity. When space and power limitations are present, sensor network nodes often avoid incorporating dedicated dynamic random access memory (DRAM) integrated circuits. Dynamic memory management is frequently discouraged in embedded systems due to resource limitations; however, certain application types are inherently reliant on this capability. The recommended approaches for reactive processing of sensor data streams in near real time are Event Based Systems (EBS). Among other applications, these systems are utilized in sports, RFID systems, stock transactions, and surveillance [3, 4]. In order to effectively manage and assess a priori unknown patterns that may emerge, diminish, or disappear during the development of IoT devices, ad hoc techniques are necessary. These techniques may include adaptive tailored memory management algorithms and pre-allocated buffers.

Furthermore, the ability to modify the configuration of the system in real-time applications to accommodate demand fluctuations and application reconfiguration is becoming an increasingly critical capability. In an attempt to develop tailored real-time application solutions that either allow systems to provide better service quality replies or exactly define the worst-case execution time of each software component, a great deal of investigation has been done on this specific subject [5–11]. Consequently, a memory management technique employed in this domain must possess characteristics such as rapid response times and minimal fragmentation [5]. Nonetheless, response time and runtime resource allocation service must remain in equilibrium in the event of memory fragmentation. This becomes particularly critical when dynamic allocation and de-allocation are utilized extensively, as they have the potential to fragment memory and induce unforeseen system behavior. Although some applications that operate on Internet of Things (IoT)-enabled devices require dynamic memory management, the majority of developers avoid it whenever possible [12]. Implementing dynamic memory in embedded real-time systems nevertheless requires determinism; memory allocation must occur at a predictable time and must not fragment.

A. Dynamic memory allocation

An application can effectively manage its memory space when unforeseen inputs occur and necessitate the storage of data in memory by utilizing dynamic memory allocation, a procedural method [13]. Operational applications can request additional memory from the system by utilizing dynamic memory allocation. When an adequate quantity of memory is accessible, the system is going to allow the application permission to utilize the specified amount. Dynamic memory allocation may outcomes in severe collateral damage, including both internal and external fragmentation. This may occasionally lead to numerous deal location and allocation operations of differing magnitudes, which may result in unforeseen system memory failures despite the fact that the total available space is adequate to fulfil the need [14]. Internal and external fragmentation forms are commonly discussed in the context of memory waste. Nevertheless, it is critical to consider the distinctions that exist between the two. Internal fragmentation is observed when memory allocation is limited to multiples of a sub-unit. Rounding up requests to the next largest multiple of the sub-unit satisfies requests of any size, leaving a negligible quantity of RAM allocated but unused. Minimizing the extent of the sub-unit can mitigate internal fragmentation; nevertheless, exacerbating this improvement is external fragmentation. Memory fragmentation that occurs when components are allocated in arbitrary-sized units is known as external fragmentation. A portion of the memory that is released in large quantities may be utilized to satisfy a subsequent request, thereby leaving behind unused memory that is inadequate to support any further requests [15, 16]. Three potential fragmentation states that may arise in a dynamic allocations system are illustrated in Fig 3. The first use case (a) illustrates a hypothetical situation where fragmentation of memory is unnecessary. However, it is evident from the second use case (b) that the heap memory region is fragmented. Ultimately, use case (c) illustrates a scenario in which fragmentation of available free memory space leads to memory failure; this is a recognized concern in the development of contemporary embedded systems. However, as demonstrated [17], fragmentation issues can be effectively managed by well-designed allocators. Industry proponents argue in favor of a zeroheap approach for IoT devices, which incorporates integrated memory managers into thread-free libraries, as memory fragmentation caused by development packages that depend on a system heap can have an adverse effect on system performance [18].

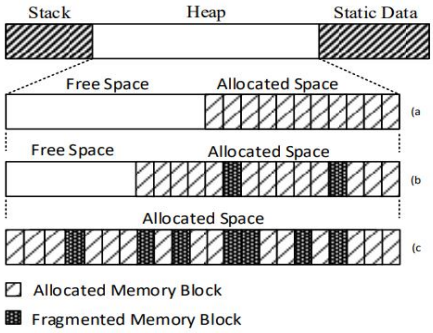

Figure 3. System memory in different time instances with diverse fragmentation states: as dynamic allocation increases fragmented memory can cause memory shortage [26].

B. Memory management unit

It is customary to employ a system that possesses both a dynamic memory management algorithm and a memory management unit (MMU) in order to prevent memory fragmentation in a system susceptible to this tendency. All memory activities related to virtual memory management are carried out by a hardware part known as a memory management unit. The primary objective of a memory management unit is to guarantee the availability of sufficient memory resources through the utilization of virtual addresses, which provide a pragmatic abstraction. Simply stated, the MMU is a physical component responsible for translating a virtual address into its corresponding physical address [19, 20]. Embedded systems endowed with MMUs are capable of defragmentation operations, which involve relocating fragmented memory regions to a single, substantial block or merging segments into a single, coherent physical block. Fragmentation issues are therefore the sole cause of performance issues. Nonetheless, this issue is exacerbated by the absence of virtual address support on MMU-less embedded systems; therefore, remapping is required.

C. Adaptive memeory management

As previously stated, no singular memory management technique is applicable to all application domains. Dynamic memory management algorithms may prove beneficial, albeit solely in particular usage contexts. The

subsequent principles functioned as a strategic guide for the advancement of AMM. The system can initially support the 16-byte smallest memory block, which contains the block's meta-data [21]. Long periods of time spent holding unused memory blocks or storing unnecessary data are inefficient on embedded systems where memory is limited. The suggested memory management technique speeds up the retrieval of such blocks by moving free fragmented memory blocks from the allocated space to the line separating the allocated and free regions.

D. Adaptive memory management structure

To properly implement the suggested technique, the memory data must be structured. We are unique among memory management strategies in that we require the use of a separate structure to hold blocks and meta-data. The memory's data structure is illustrated in Fig 4. The data and pointers sections constitute the two primary components. They are divided by the Margin, which is a compact quantity of accessible memory space. The data section retains the pointer addresses that are associated with a particular data block.

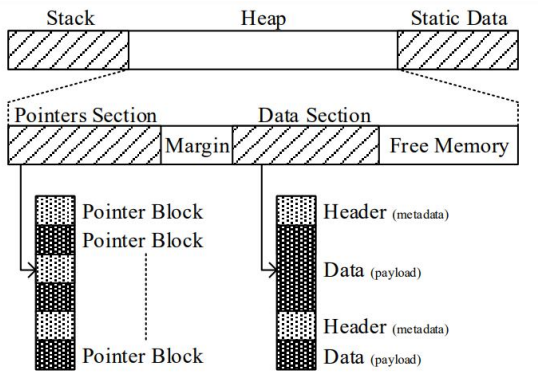

Figure 4. Organization of memory structure

The data elements in physical form are stored in the data section of memory. Developers can readily migrate to the new memory management strategy [22] by employing free techniques and API allocations that are nearly indistinguishable from those utilized by LibC's default memory management design.

II. Literature Review

Z. Ma et.al (2023) [23] discussed the microcontrollers serve as the computational heart of embedded systems. As a result of financial and power-related limitations, these systems lack memory protection units (MPUs) and memory management units (MMUs). Subsequently, embedded software encounters two interconnected obstacles, both of which are stack-related. Initially, physical memory utilized by the stack is typically statically allocated per operation in a multitasking system. It is challenging to detect a stack overflow on low-end microcontrollers lacking an MPU. Segmented stacks combined with Rust may, surprisingly, ensure memory safety in the absence of an MPU or MMU. The dynamic allocation of memory to task-specific stacks is another advantage of segmented stacks; when combined with intelligent scheduling, this can further enhance memory efficiency.

C Heinz et.al (2023) [24] presented the Integrating intellectual property (IP) components into a system-on-a-chip (SoC) raises a number of important issues, some of which might not be dependable. Incorporating unauthorized writes onto memory or peripheral devices can lead to data corruption and exfiltration, two of the aforementioned threats. Traditional security measures, such as memory protection or management units, impose a significant burden on embedded devices' hardware and provide a restricted level of detail in terms of protection. Through the implementation of a technique that actively monitors bus transfers and alternates between different protection rules, the team is able to dynamically adapt to changing system conditions without sacrificing protection granularity or area overhead.

A Jaamoum et.al (2021) [25] determine Cache memory is used by a significant fraction of computing systems to carry out effective activities. A multitude of systems are vulnerable to cache side-channel attacks, as has been demonstrated. There exists a wealth of documented solutions and countermeasures. However, the overwhelming majority of them encounter difficulties in surmounting the constraints and limitations that are imposed by embedded systems. By showcasing the efficacy of this countermeasure in safeguarding the system against cache side-channel attacks and ensuring minimal overheads, it verify that it is applicable to embedded devices as well.

K Bukkapatnam et.al (2020) [26] discussed the comprehension of optimizing memory utilization has assumed paramount importance in light of the pervasive integration of embedded systems facilitated by the Internet of

Things (IoT). Implementing hardware-based memory management units (MMUs) in certain regions of embedded systems may be unfeasible due to spatial limitations. In the absence of an MMU, various DMA concepts are implemented in embedded systems, albeit with some degree of limitation. The suggested strategies have led to a three to fourfold increase in allocating velocities, while deallocation velocities have decreased marginally.
A Hazarika et.al (2020) [27] proposed the scaling of their processing infrastructure to meet the requirements of high-performance computing (HPC) applications and big data is presently a significant challenge that data scientists need to overcome. To optimize the system's capacity, modifications were made to the dynamic random-access memory (DRAM) architecture, a memory-centric hardware accelerator was integrated into the heterogeneous system, and processing-in-memory (PIM) was implemented. With regard to the CPUs, an efficient MMU is responsible for memory protection, cache management, and bus arbitration.
M Salehi et.al (2020) [28] discussed the majority of Internet of Things (IoT) devices are constructed with exposed metal. The firmware of a bare-metal device communicates directly with the hardware, bypassing the requirement for an intermediary operating system. Although bare metal devices offer greater adaptability and efficiency, they continue to be susceptible to memory corruption vulnerabilities. Fuzzing is a well-known and efficient software testing technique utilized to identify vulnerabilities. Memory corruption issues, which circumvent well-known security measures such as MMU, are readily identifiable and thus crucial to the detection and diagnosis of fuzzing techniques. These types of protective measures are unfortunately absent from naked metal devices. After conducting an evaluation, it was concluded that SBS reduced the amount of work demanded of security analysts without identifying the same set of memory error categories as prior techniques.
A Abbasi et.al (2019) [29] discussed the memory corruption vulnerabilities have been pervasive in embedded systems for several decades. Nevertheless, conventional fortification strategies are considerably more difficult to devise and execute due to the constraints imposed by this environment. This phenomenon, coupled with the complex and uncertain characteristics of embedded patch management, leads to extended periods of vulnerability exposure and defects that are relatively easy to exploit. A substantial enhancement is necessary in this circumstance due to the presence of delicate yet vital components in numerous embedded systems. This is the first quantitative examination of the adoption of exploit mitigation across 42 embedded operating systems. The outcomes indicate a substantial disparity between the embedded and general-purpose operating systems.

III. Methodology And Experimentation

A. Smart memory management (SaMM)

As stated previously, there is no universally effective techniques for managing memory. A memory management solution that is both dependable and quick is required to accommodate the growing intricacy of real-time applications. Memory defragmentation is reduced while access performance is maintained through the implementation of a memory management strategy described. Additionally, it strives for efficacy across a broad spectrum of applications encompassing various functions.
SaMM primarily employs these two methods to improve memory management.
E-TLSS stands for Enhanced Two-Level Segregated Fit Memory Allocator.
Expandable free memory slots can be generated through the programmable displacement of memory blocks.
Functioning under the assumption that the smallest available memory block is 16 bytes, SaMM comprises the metadata associated with a specific block allocation. Because embedded systems are so compact and have limited capacity, it is not possible to add a substantial amount of memory. Furthermore, it is critical that memories are promptly discharged and not retained for an extended duration. In order to enhance memory efficiency, the configuration of memory blocks is consistently modified to partition defragmented blocks from available memory.
Given that two cycles of operations occur, Distribute memory
Allocate memory
Move defrag blocks,
The efficacy of SaMM can be evaluated by constructing a simulated embedded system environment. By simulating an MMU-less system in this environment, we would be able to evaluate the memory management capabilities of SaMM. Memory allocation and deallocation time, memory footprint (total memory used), and fragmentation levels (unused dispersed memory chunks) will be defined as critical performance metrics. We can evaluate the performance of SaMM and existing memory management techniques across these metrics by executing simulations with each. Furthermore, an assessment will be conducted on system responsiveness to memory requests, memory access speed, and overall resource utilization in order to obtain a comprehensive understanding of SaMM's efficacy.

IV RESULT AND DISCUSSION

A. Smart memory management structure

Memory is composed of pointer blocks and data blocks; complex data structures may contain multiple pointers. In the space between data blocks and pointer blocks, a nominal buffer is implemented. The subsequent data block is presumed to grant access to additional pointers if the buffer is full. Every designated block consists of two components: the metadata, which comprises the preamble, and the data, on which this discourse is going to concentrate. This metadata section is going to be supplemented with additional data-related information as part of the SaMM strategy, including the quantity of data and the location where the final data package is terminated. In order to optimize and enhance the system's performance, multiples of four bytes are maintained for every block size. Fig 5 illustrate the memory organization as per SaMM as shown below.

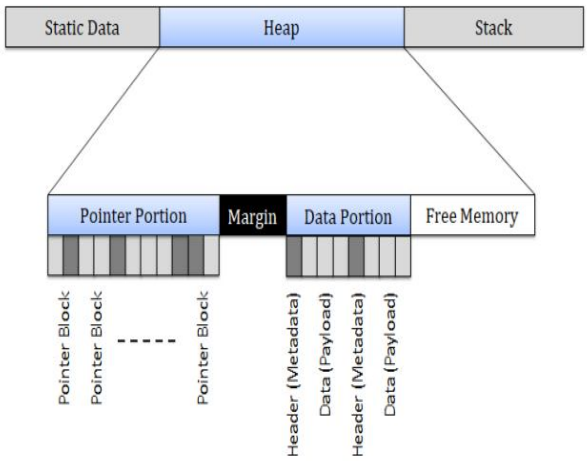

Figure 5. Memory organization as per SaMM

In contrast to simpler methodologies such as linked lists, this data management approach undoubtedly has the capability to manage complex functions. Despite this, it is critical to bear in mind that this approach or techniques is only viable in embedded systems, where physical space constitutes a significant limitation.

B. Smart memory management structure

Memory Management API provides the following methods to make necessary operations.

Public Functions for Allocation	mem_alloc, mem_alloc_p
Public Function for De-allocation	mem_free
Public Functions for Pointer Operations	mem_add_pointer, em_remove_pointer
Allocates size_x volume of data block	void * mem_alloc(size_x)
Combining Data Allocation and adding a Pointer	void * mem_alloc_p(void * , size_x)
Seeking Pointer Address	void * mem_add_pointer(void *)
Deregistering Pointer along with Address	void * mem_remove_pointer(void *)
De-allocation of Data blocks & Defragmentation	void * mem_free(void *)

C. Allocation process algorithm

Fig 6 depict the memory allocation as per SaMM as shown below. Below are the steps in Algorithm of achieving an efficient Allocation.

D. Deallocation Process Algorithm

A previously allocated memory block is removed by the de-allocation operation via reference to the location of the pointer. Initiates the defragmentation procedure in order to prepare the resource for additional allocations. The concept of de-allocation is to combine all available cells with the fractured ones by relocating the fragmented cells to the side where they are already present. Currently, each of them can be defragmented (between allocation cycles) by a distinct process operating within it. Additionally, multitasking programs execute more efficiently on a real-time operating system due to the fact that de-allocation and defragmentation are managed independently. Fig 7 illustrate the memory deallocation as per SaMM as shown below.

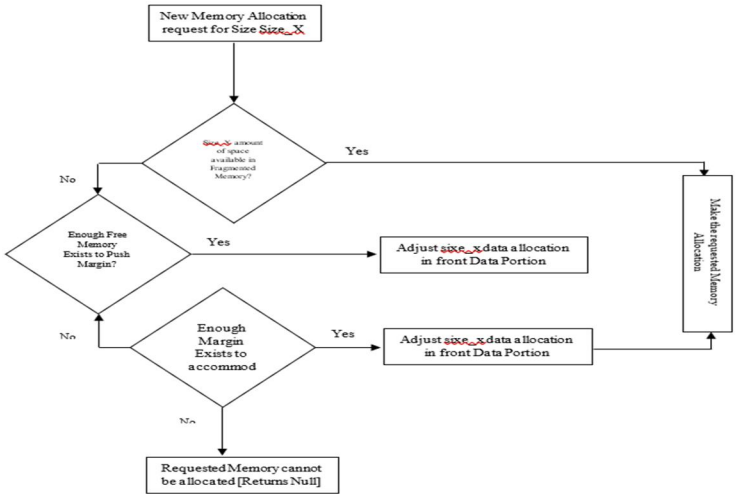

Figure 6. Memory allocation as per SaMM

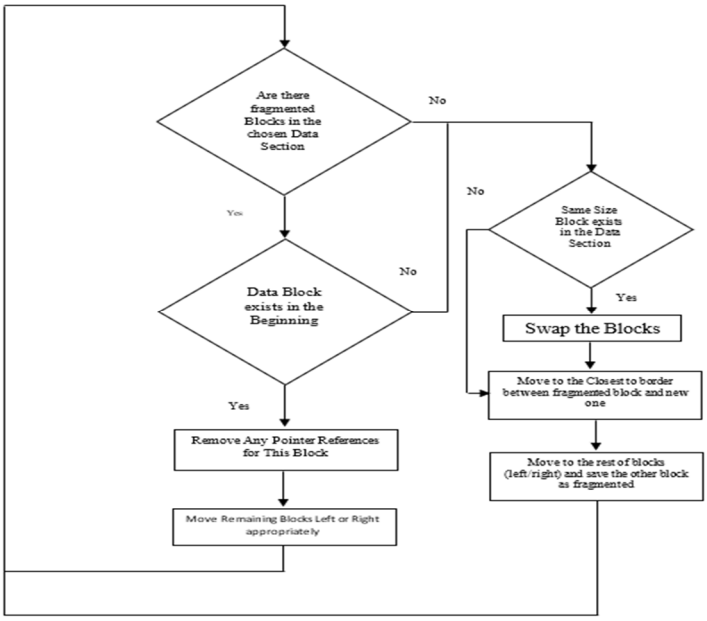

Figure 7. Memory Deallocation as per SaMM

In a nutshell, the process is,

- In the event that blocks are fractured, the remaining blocks must be shifted to the left or right in order to accommodate the accessible data blocks within the largest chunk.
- Identify blocks of comparable dimensions and exchange them, relocating the unbroken blocks to one side if none are damaged. Continue performing this iteratively to ensure optimal segregation.

E. Defragmentation Process Algorithm

Defragmentation, which operates in parallel with de-allocation and is frequently executed to remove fragmented blocks, is terminated throughout each allocation cycle.

Limitations of SaMM A statistical analysis reveals that the CPU cycle time is going to increase marginally as data blocks are shifted in the SaMM solution (since no experiments are conducted and the results are solely theoretically calculated).

- Using default GLIBC technique – memmove
- Data Transfer rate of about 500 MBPS
- In an ARM M4 Microprocessor

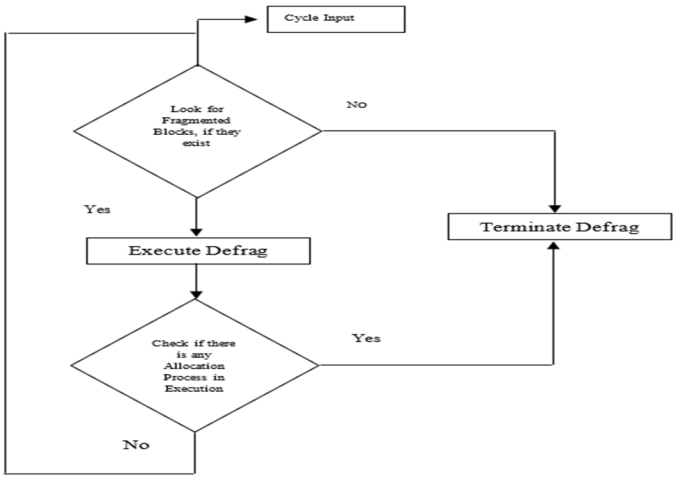

Figure 8. Defragmentation process cycle as per SaMM

SaMM is going to need between three and sixteen CPU cycles to execute an operation under these conditions. This is certain to incur higher expenses compared to a conventional DMA solution involving significant defragmentation, rendering it impracticable for implementation in Internet of Things solutions for embedded devices.

Distribute Memory: In this step, memory blocks are organized and fragmented blocks are identified.

Allocate Memory: The system identifies a memory request, say for 40 bytes. It looks for a suitable free memory block to allocate.

Move Defragmented Blocks: If necessary, block is moved to consolidate free memory.

TABLE 1: MEMORY ALLOCATION STATUS

Memory Block	Size (bytes)	Status	Description
Block 1	32	Allocated	Used for program data
Block 2	64	Free	Initially free memory
Block 3	16	Allocated	Used for stack
Block 4	128	Free	Initially free memory
Block 5	48	Allocated	Used for program instructions

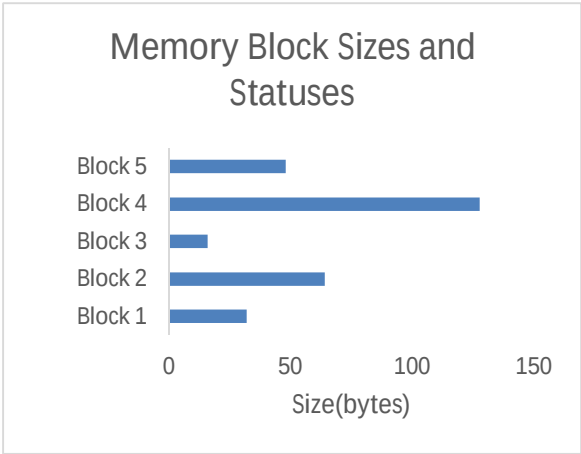

Figure 9. Memory block sizes and statuses

The graph shows that there are five memory blocks, ranging in size from 16 bytes to 128 bytes. Three of the blocks are allocated, and two are free. The allocated blocks are used for program data, stack, and program instructions. The largest block is block 4, which is 128 bytes and is free. The smallest block is block 3, which is 16 bytes and is allocated.

F. Performance Evaluation Results

❖ Testbed and Metrics

To facilitate this assessment, a simulation framework was devised to represent an embedded system lacking an MMU. Various memory management techniques are implemented within this framework, such as SaMM and a default approach (e.g., fixed-size allocation or buddy system). The subsequent performance metrics were assessed:

- *Memory Allocation Time*: Time taken to allocate a memory block.
- *Memory Deallocation Time*: Time taken to deallocate a memory block.
- *Memory Footprint:* Total memory used by the memory management system itself (overhead).
- *Memory Fragmentation***:** Percentage of memory unusable due to small, scattered free blocks.

G. Comparison with Existing Techniques

The graphs below compare the performance of SaMM with a baseline memory management technique on various memory access patterns:

- *Random Allocation/Deallocation*: Memory requests arrive randomly with varying sizes.
- *Sequential Allocation/Deallocation:* Memory requests follow a sequential pattern, allocating and deallocating contiguous blocks.
- *Working Set:* A fixed set of memory blocks is repeatedly allocated and deallocated, simulating a program with a defined memory footprint.

❖ *Memory Allocation Time*

Memory allocation time, as it pertains to embedded systems lacking an MMU (Memory Management Unit), is a term used to describe the velocity at which the SaMM system locates and allocates an appropriate memory block in response to a program's request. A SaMM system should, in an ideal world, strike a balance between efficient memory usage and rapid allocation.

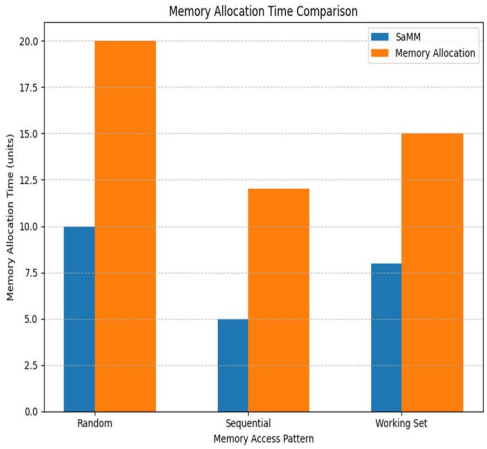

Figure 10: Memory Allocation Time Comparison

Memory allocation timings are compared in the graph you provided in accordance with memory access patterns. The memory access pattern, which may be random, sequential, or working set, is represented along the x-axis. Memory allocation time is represented on the y-axis in indeterminate units.

❖ *Memory Deallocation Time*

Memory deallocation time, within the framework of SaMM for embedded systems lacking an MMU, denotes the velocity at which a previously allocated memory block is once more deemed operational. Nevertheless, SaMM places a higher emphasis on resource efficiency and real-time limitations; as a result, deallocation may occur marginally more slowly, but memory utilization will be more predictable and optimized.

The graph presents a comparison of the deallocation times of Allocate and SaMM, two memory management techniques. SaMM is the most rapid technique across all three memory access patterns, as indicated by the graph. In contrast to SaMM, which has an almost negligible deallocation time, allocate exhibits a considerably longer deallocation time. To illustrate, when a random memory access pattern is utilized, the deallocation time for Allocate is approximately 2.5 times longer in comparison to that of SaMM.

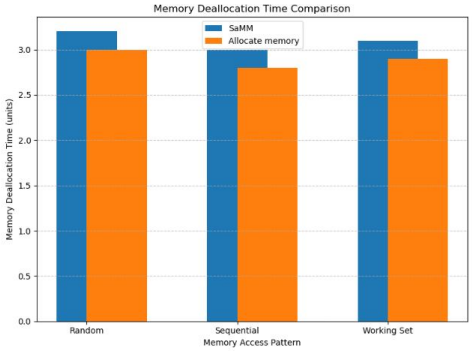

Figure 11. Memory Deallocation Time Comparison

❖ *Memory Footprint*

The graph shows that the random-access pattern has a higher memory footprint than the sequential access pattern. This is likely because random access requires the program to keep more data in memory at once, in order to quickly access any given piece of data. Sequential access, on the other hand, can access data in a more linear way, potentially requiring less data to be held in memory at any one time.

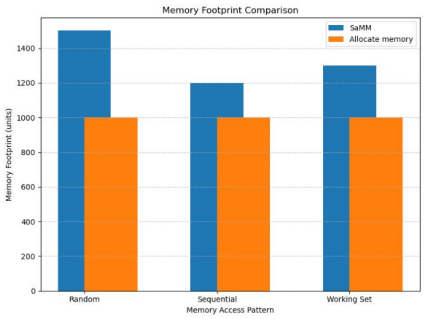

Figure 12. Memory Footprint Comparison

❖ *Memory Fragmentation*

In the absence of Memory Management Units (MMU), embedded systems allocate memory through software. SaMM (Smart Memory Management) is an example of such a method. SaMM is still susceptible to memory fragmentation, however. This occurs when memory that has been released is fragmented into smaller sections across the memory space. SaMM may be unable to allocate a larger block due to these fragmented spaces, even if there is sufficient total free memory; this could potentially result in the system running out of usable memory.

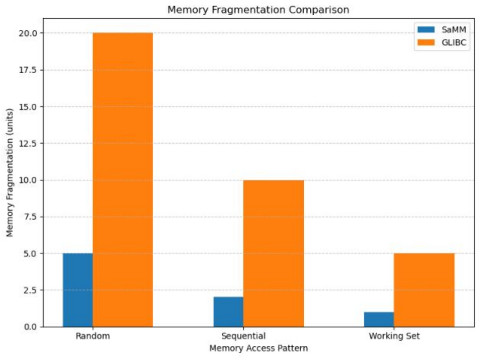

Figure 13. Memory Fragmentation Comparison

SDRAM and memory allocation. Different memory access patterns are represented along the x-axis: random, sequential, and working set. In units, the y-axis represents memory fragmentation. Fragmentation is consistently

greater in allocated memory than in SDRAM across all three memory access patterns, as depicted in the graph. This implies that allocated memory is more susceptible to fragmentation compared to SDRAM.

V. Conclusions

A Smart Memory Management (SaMM) system for Embedded Systems without MMU could efficiently manage memory resources in resource-constrained environments. By optimizing memory consumption through the use of complex memory usage algorithms and dynamic memory allocation mechanisms like heap and stack management, SaMM ensures the performance and stability of embedded systems. SaMM effectively mitigates memory intrusions and fragmentation through the implementation of data structures, memory pooling, and fragmentation mitigation techniques. Furthermore, SaMM enhances the system's resilience and responsiveness by dynamically allocating memory in accordance with consumption trends and system demands through the utilization of adaptive strategies and predictive algorithms. SaMM's flexible and scalable architecture facilitates seamless integration into embedded systems that possess diverse memory and application requirements. Its minimal overhead and compact dimensions, which do not impede system performance, render it well-suited for environments with limited resources. In the absence of MMUs, a Smart Memory Management solution resolves memory management issues in embedded systems with dependability, efficiency, and adaptability. By means of memory optimization, enhanced system stability, and application-specific compliance, SaMM facilitates the development of resource-efficient embedded systems.

References

[1] Cheng, Xiaohui, and Haodong Tang. "Dynamic memory allocation of embedded real-time operating system based on tlsf." Applications and Techniques in Information Security: 9th International Conference, ATIS 2018, Nanning, China, November 9–11, 2018, Proceedings 9. Springer Singapore, 2018.

[2] J.Light, “Embedded Programming for IoT”, In Embedded Linux Conference & OpenIoT Summit, April 2016.

[3] Convent, Lukas, et al. "Hardware-based runtime verification with embedded tracing units and stream processing." International Conference on Runtime Verification. Cham: Springer International Publishing, 2018.

[4] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin, “Efficient processing of uncertain events in rule-based systems,” IEEE Trans. On Knowledge and Data Engineering, vol. 24, no. 1, pp. 45–58, 2012.

[5] Kim, Chung Hwan, et al. "Securing Real-Time Microcontroller Systems through Customized Memory View Switching." NDSS. 2018.

[6] Baccelli, Emmanuel, et al. "RIOT: An open source operating system for low-end embedded devices in the IoT." IEEE Internet of Things Journal 5.6 (2018): 4428-4440.

[7] Nagadi, Khalid, et al. "A hybrid simulation-based assessment framework of smart manufacturing systems." International Journal of Computer Integrated Manufacturing 31.2 (2018): 115-128.

[8] Martins, José Carvalho. Ontology-driven metamodeling towards hypervisor design automation: microkernel infrastructure. Diss. 2018.

[9] T. Kani, "Dynamic memory allocation," ed: U.S. Patent Application 14/396,383, 2012.

[10] J Wickramasinghe, Udayanga, and Andrew Lumsdaine. "rmalloc () and rpipe () a uGNI-based Distributed Remote Memory Allocator and Access Library for One-sided Messaging." Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers. 2018.

[11] Y.-H. Yu, J.-Z. Wang, and T.-Y. Sun, "A Novel Defragmemtable Memory Allocating Schema for MMU-Less Embedded System," in Advances in Intelligent Systems and Applications - Volume 2. vol. 21, J.-S. Pan, C.-N. Yang, and C.-C. Lin, Eds., ed: Springer Berlin Heidelberg, 2013, pp. 751-757.

[12] Wang, Guohua, Song Liu, and Jiangtao Sun. "A dynamic partial reconfigurable system with combined task allocation method to improve the reliability of FPGA." Microelectronics reliability 83 (2018): 14-24.

[13] Mishra, Debadatta, and Purushottam Kulkarni. "A survey of memory management techniques in virtualized systems." Computer Science Review 29 (2018): 56-73.

[14] K. Wang, "Memory Management," in Design and Implementation of the MTX Operating System, ed: Springer, 2015, pp. 215-234.

[15] Gerber, Simon. Authorization, Protection, and Allocation of Memory in a Large System. Diss. ETH Zurich, 2018.

[16] Ausavarungnirun, Rachata, et al. "Mosaic: Enabling application-transparent support for multiple page sizes in throughput processors." ACM SIGOPS Operating Systems Review 52.1 (2018): 27-44.

[17] Khan, Omair Ahmad, et al. "Leveraging named data networking for fragmented networks in smart metropolitan cities." IEEE Access 6 (2018): 75899-75911.

[18] Synopsys Inc., “Synopsys and Cypherbridge Accelerate TLS Record Processing for IoT Communication with Optimized Hardware/Software Security Solution”, Embedded World 2016 in Nuremberg, Germany, February 23-25

[19] Jomaa, Narjes, et al. "Formal proof of dynamic memory isolation based on MMU." Science of Computer Programming 162 (2018): 76-92.

[20] Mishra, Debadatta, and Purushottam Kulkarni. "A survey of memory management techniques in virtualized systems." Computer Science Review 29 (2018): 56-73.
[21] G. Barootkoob, M. Sharifi, E. M. Khaneghah, and S. L. Mirtaheri, "Parameters affecting the functionality of memory allocators," in Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on, 2011, pp. 499-503.
[22] S. Loosemore, U. Drepper, R. M. Stallman, A. Oram, and R. McGrath, The GNU C library reference manual: Free software foundation, 1993.
[23] Ma, Zhiyao, and Lin Zhong. "Bringing Segmented Stacks to Embedded Systems." Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications. 2023.
[24] Heinz, Carsten, and Andreas Koch. "DD-MPU: Dynamic and Distributed Memory Protection Unit for Embedded System-on-Chips." International Conference on Embedded Computer Systems. Cham: Springer Nature Switzerland, 2023.
[25] Jaamoum, Amine, Thomas Hiscock, and Giorgio Di Natale. "Scramble cache: An efficient cache architecture for randomized set permutation." 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2021.
[26] Bukkapatnam, Krishnaveni, et al. "Smart memory management (SaMM) for embedded systems without MMU." IOP Conference Series: Materials Science and Engineering. Vol. 981. No. 3. IOP Publishing, 2020.
[27] Hazarika, Anakhi, Soumyajit Poddar, and Hafizur Rahaman. "Survey on memory management techniques in heterogeneous computing systems." IET Computers & Digital Techniques 14.2 (2020): 47-60.
[28] Salehi, Majid, Danny Hughes, and Bruno Crispo. "{μSBS}: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability." 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). 2020.
[29] Abbasi, Ali, et al. "Challenges in designing exploit mitigations for deeply embedded systems." 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2019.