Smart Memory Management (SaMM) For Embedded Systems without MMU

Krishnaveni Bukkapatnam¹, Prashant², Ch Kranthi Rekha³, Eelandum Kumaraswamy⁴, Rambabu Vatti⁵
¹,²,³,⁵Bharat Institute of Engineering and Technology, Hyderabad, India
⁴Sumathi Reddy Institute of Technology for Women, Warangal, Telangana, India
Email: SriveniZ@Gmail.com

Abstract. In the wake of extensible usage of IOT (Internet of Things) enabled Embedded Systems, it is of great importance to find ways of using Memory in the most efficient way. Embedded Systems are also space constrained, hence at various places, it may not be possible to deploy a Hardware based – (MMU) Memory Management Units. In MMU less Embedded systems, there are various principles of DMA being applied; however, they are all constrained in one way or the other. With this backdrop, this paper is exploring opportunities to achieve higher performance and efficiency in Memory Management by using Smart and Programmable methods. Proposed methods have achieved an overall improvement in Allocation speeds to the tune of 3-4 times, however there is a marginal drop in Deallocation speed. Overall a better bargain and another very good outcome is 0 defragmented memory. This paper also presents algorithms for Allocation, Deallocation and Defragmentation processes, which can be implemented within available Application Software.

Keywords: Memory Management Units, MMU-Less Embedded Systems, Dynamic Memory Allocation Schemes, Internet of Things

1. Introduction
Embedded Systems: An embedded system is a microprocessor-based computer hardware system with integrated software that is designed to perform a dedicated function, either as an independent system or as a part of a larger system. At the core is an integrated circuit designed to carry out computation for real-time operations.
Internet of Things: The Internet of things (IoT) is a system of interrelated computing devices, mechanical and digital machines provided with unique identifiers (UIDs) and the ability to transfer data over a network of Cloud Computing Devices without requiring human-to-human or human-to-computer interaction.

In todays’ age of extreme control need has developed a field called Internet-of-Things or shortly called as IoT, rather this was there for few years, but has exponentially grown in the last couple of years. Systems are programmed to control, advanced Medical and Defense Systems to household gadgets like, Stoves, Ovens, TV, Washing Machines, etc. All this has put in IOT of stress or rather importance in improving Embedded Systems, as they are at the crux of implementing all the above functions.

IoT Developers build software for the above mentioned control functions, in Microprocessors, which are part of Embedded Systems, which are in-turn embedded into Machines/ devices, which are controlled. Important components of any Software are its Algorithm and Memory. Design of Memory itself as well as mechanism of accessing it becomes critical from the perspective of speed and efficiency of access. In Embedded systems, space is also a factor; a lot of Memory cannot be bundled in small spaces.

In certain Embedded systems, additional Hardware based Memory Management Units are deployed for facilitating Memory Actions, but due to various reasons, it is not preferred and hence MMU-Less Embedded Systems are very prevalent. [23]
2. Dynamic Memory Allocation (DMA)
Memory is a on the location, physical storage device, being physical, it needs space and in Embedded systems, physical space will be a challenge. Memory also is consumed, based on processes it is running at that point of time, once that process is completed, then effectively the entire memory should be available for any future processes. Dynamic Memory Allocation, as its name implies, is a dynamic process of allocation and de-allocation of between these functional processes. However there are some challenges, like, fragmentation, for example, if a there were multiple processes executing at a little staggered, in between these memory usages, there will be certain memory locations still in use, but rest of the memory will be free, these in between blocks causes hindrance in a bulk allocation for any next process. So in effect, even though there is memory available, system, will return as not available for it.

Fragmentations are bound to be there, we need to understand them, to come up with a suitable strategy.

We can differentiate fragmentations into Internal and External

| # | Internal Fragmentation | External Fragmentation |
|---|------------------------|------------------------|
| 1 | Fixed size memory blocks | Variable Size Memory Blocks |
| 2 | Internal Fragmentation is caused, when memory requested is more than a single block | External Fragmentation is an effect, when a process completes its execution and allocated memory is released |
| 3 | Solutions is - Create smaller block sizes, so that allocated memory for a process is close fit, with least wastage | Solution is - Segmentation, Paging and by compacting |

![Diagram of System Memory and Fragmentation Levels](image)

2.1. Memory Management Unit
To circumvent the fragmentation issue, a Memory Management Unit is implemented along with Dynamic Memory Access Algorithms. A MMU is a Hardware Component, which handles all Virtual Memory Operations. It generates Physical Address for every Virtual Address.[15,16] Hence MMU supported Embedded Systems can remap Fragmented Memory spaces or put all those Fragmented Memory Spaces to one side, hence Memory Allocation is easier. [7]
However, there are a huge set of Applications, where MMU”s cannot be implemented, either due to higher costs or space constraints. For them, through this paper, we present an approach more suitable for IoT enablement Embedded Systems. We call it “Smart Memory Management (SaMM)”, this is suitable for minute systems, where on board memory will be limited by the space for MMU which is additional hardware.

3. Related Research Work
Research Scholars, Innovators and Technologists are constantly thriving to come up with innovative strategies and Algorithms to segregate the Fragmented Memory units and make it available in a most effective way for Functional Usage. Few of the approaches are – Sequential Fit, Segregated Fit, Fixed Size Blocks Allocation, Buddy Systems, Indexed Fit, Bitmap Fit, etc. [1,2,10]

3.1. Fixed Size Blocks Allocation
Fixed-size blocks allocation, also called memory pool allocation, uses a fixed-size blocks of memory (usually of same sizes). This method is suitable for simple embedded systems where allocations needs are not huge, but definitely suffers from fragmentation, especially when the memory addresses are large. However, if there are simpler and frequent needs of memory (for example in Video Games), this strategy is definitely useful as it reduces the additional overheads.

3.2. Sequential and Segregated Fit
There method consists of an array of free lists, each holding free memory units of a particular range of sizes. Strategy is to allocate the best fit from the array (best fit, first fit). If this fails, a next larger block is reviewed.

3.3. Buddy Systems
Memory is organized in multiple pools or blocks, with each block holding memory units of a certain fixed size only… Now, as a new functional process makes a particular size of memory request, then the best fit from a particular pool is chosen from the list and allocated and if there is any 2 memory units are left out un-allocated, then, these buddies are clubbed into one and then moved into an appropriate pool, based on its combined size.

3.4. Slab Allocator
Slab allocator, creates different sized blocks and when a new functional process makes a request for an allocation, then it will allocated it to the closest one. Majority of memory allocation requests in the kernel is for frequent and small memory. Slab allocator strategy is all about commonly used objects are to be pre-allocated in continuous areas of physical memory called „slabs”. Whenever a new object is to be allocated, the slab allocator returns the first available item from a suitable slab corresponding to the object type and size. Slab Allocator Strategy significantly reduces Internal Fragmentation

3.5. Indexed Fit
Suitable free blocks are listed and indexed.

As we are aware, each of the above strategy is its own way, trying to reduce the impact of memory fragmentation. However, memory fragmentation without hardware MMU components is impossible to completely avoid, and sometimes a custom memory management scheme is developed for a particular application.

4. Smart Memory Management (SaMM)
As outlined before, there is no one method of Memory Management, which will be a fit-all solution. It further gets complicated with Real-time Applications, where the Memory Management solution has to be very fast as well as stable.

This paper proposes a Memory Management Scheme, which minimises memory Defragmentation without affecting the speed of access. Also, it tries to be effective in a wide range of applications across diverse functions.

SaMM mainly uses these 2 approaches to achieve an enhanced Memory handling
- Enhanced Two-Level Segregated Fit Memory Allocator (E-TLSS) [1,2]
- Programmable movement of Memory Blocks to create larger Free Memory Slots [7]

SaMM assumes that, as a smallest memory block available is 16 bytes, which will be including Metadata of a certain block allocation. The Embedded Systems are very small and space constrained, hence a lot of memory cannot be added on to the system. Due to this also, it is important that memory is as and when released and they are not held for too long. Also continuously, Memory blocks are moved around to segregate free memory and defragmented blocks separately, maximising memory usage. Since there are 2 cycles of operations –
- Allocate memory and
- Move defrag blocks,

4.1. Smart Memory Management Structure
Memory is organized into Pointer Blocks and Data Blocks; there can be multiple pointers for making complex data structures. A small buffer is maintained between Pointer Blocks and Data Blocks; to facilitate any additional pointers, if the buffer gets consumed, then subsequent Data Block is taken over. Each allocated block has two parts, metadata, which contains header and data portion, called as payload, which is the main purpose of all this drama. As part of SaMM approach, this metadata portion will be added with more information about the data, like where did the last data package end and the size of data. Each block size is maintained at multiples of 4 bytes to optimize and increase the system performance. [1,2,7,18]

![Fig 4: Memory Organization as per SaMM](image)

Compared to previously studied Data Management approaches, this definitely has the capability to handle complex functions as against simple linked lists, etc. However, it has to be kept in mind that this strategy or approach is suitable in Embedded Systems, where physical space is a major constraint.
4.2. Smart Memory Management Structure
Memory Management API provides the following methods to make necessary operations. [19]

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`

Sample Usage of above methods:
- 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 *)`

4.3. Allocation Process Algorithm
Below are the steps in Algorithm of achieving an efficient Allocation

We have discussed a few allocation processes in III Related Research Work section, compared to those SaMM process is quite simple and effective, it uses margin for effectively allocating Payloads, either on the front or the back, based on availability of enough of Margin space. Obviously in case there is no enough free space in Fragmented Block or Margin or in Free Memory, the process of Memory allocation ends unsuccessful, returns Null.
4.4. Deallocation Process Algorithm
The de-allocation process expunges an allocated memory block, using the Pointer reference location. Also initiates defragmentation to make it available for subsequent allocations.

Our suggested process of De-allocation works on a basic principle of moving all fragmented cells on one side, where there are already such cells existing, so that available cells all can be together and the fragmented ones are also grouped. They all can be defragmented by a separate process, which is running by it (in between Allocation cycles). This also separate handling of de-allocation and defragmentation helps in multitasking applications on a real-time OS.

Fig 6: Memory Deallocation as per SaMM

In a nutshell, the process is,
- If there are fragmented blocks existing, then move the remaining blocks to the left or right appropriately, so that available data blocks are available in the largest chunk.
- If there are no fragmented blocks also, find the similar size block and swap them and keep moving them to one side.

Keep doing this in a loop so that segregation happens optimally.

4.5. Defragmentation Process Algorithm
As we mentioned earlier, Defragmentation is a separate process from De-allocation and runs frequently to clear fragmented blocks, it is aborted during any allocation cycle.
5. Limitations with SaMM

If we review this statistically, (since all experiments are not conducted, it is theoretically computed), as there is movement of data blocks in SaMM solution, there will be some increase in the CPU cycle time… say –

- Using default GLIBC method – memmove [19]
- Data Transfer rate of about 500 MBPS
- In an ARM M4 Microprocessor

Under these conditions SaMM will achieve 3 to 16 CPU cycles to complete an operation. This will be definitely higher than a plain vanilla DMA solution, with extensive Defragmentation and hence it becomes unsuitable for Embedded Systems in IoT Solutions.

However, as far as other limitations are concerned, SaMM is currently designed for low cost and low memory operations Embedded Systems only, more work needs to be done to make this more generic and at some point of time; this can become used as a Sub Allocator and can become a generic solution component in Unix/ Linux Operating Systems.

6. Conclusions and Next Steps

There have been many attempts to address the challenges faced in Memory Allocation of Embedded Systems. During the last few years, due to increased usage of IoT systems, it is the most discussed topic. Industry seeks solutions with stable execution of Embedded Systems. Since earlier Memory Allocation schemes were causing much of defragmentation and hence unstable operations, we have presented a solution through this paper, which will eliminate defragmentation. This is predominantly achieved by segregating fragmented blocks into one place, so that an independent defragmentation process can clear all these in one shot. However as future steps, there are opportunities to expand this solution to more and more devices. Currently this solution is most suitable in small operations and
becomes very cost effective. Then this can be developed into a universal Smart Memory Allocation Scheme (SMALL).

References

[1] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on, 2004, pp. 79-88.

[2] X. Sun, J. Wang, and X. Chen, "An improvement of TLSF algorithm," in Real-Time Conference, 2007 15th IEEE-NPSS, 2007, pp. 1-5.

[3] M. Masmano, I. Ripoll, and A. Crespo, "Dynamic storage allocation for real-time embedded systems," Proc. of Real-Time System Symposium WIP, 2003.

[4] A. Crespo, I. Ripoll, and M. Masmano, "Dynamic memory management for embedded real-time systems," in From Model-Driven Design to Resource Management for Distributed Embedded Systems, ed: Springer, 2006, pp. 195-204.

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

[6] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in USENIX summer, 1994.

[7] Y.-H. Yu, J.-Z. Wang, and T.-Y. Sun, "A Novel Defragmentable 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.

[8] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," ACM SIGMOD Record, vol. 34, pp. 42-47, 2005.

[9] Puaut, "Real-time performance of dynamic memory allocation algorithms," in Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on, 2002, pp. 41-49.

[10] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in Memory Management, ed: Springer, 1995, pp. 1-116.

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

[12] H. J. Boehm and P. F. Dubois, "Dynamic memory allocation and garbage collection," Computers in Physics, vol. 9, pp. 297-303, 1995.

[13] W. E. Croft and A. Henderson, "Eliminating memory fragmentation and garbage collection from the process of managing dynamically allocated memory," ed: Google Patents, 2002.

[14] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: solved?," in ACM SIGPLAN Notices, 1998, pp. 26-36.

[15] D.-B. Koh, "Memory management unit with address translation function," ed: Google Patents, 1997.

[16] J. E. Zolnowsky, C. L. Whittington, and W. M. Keshlear, "Memory management unit," ed: Google Patents, 1984.

[17] G. S. Brodal, E. D. Demaine, and J. I. Munro, "Fast allocation and deallocation with an improved buddy system," Acta Informatica, vol. 41, pp. 273-291, 2005.

[18] 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.

[19] S. Loosemore, U. Drepper, R. M. Stallman, A. Oram, and R. McGrath, The GNU C library reference manual: Free software foundation, 1993.

[20] 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

[21] J.Light , “Embedded Programming for IoT”, In Embedded Linux Conference & OpenIoT Summit, April 2016.
[22] 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.

[23] ARM, “mbed IoT Device Platform”, www.arm.com/products/Internet-of-things-solutions/mbed-IoT-device-platform.php