Grenze International Journal of Engineering and Technology, January Issue

Runtime Performance Improvement of Embedded Linux System through Comparative Analysis of Memory Management Algorithms

Manasi Natu[1], Supriya Rajankar[2], Madan Mali[3] and Omprakash Rajankar[4]
[1-3]Electronics and Telecommunication Engineering, Sinhgad College of Engineering, Pune, India
Email: manasi0597@gmail.com, supriya.rajankar@gmail.com, mbmali.scoe@sinhgad.edu
[4]Shri Tuljabhavani College of Engineering, Tuljapur, India
Email: online.omrajankar@gmail.com

***Abstract*— This paper focusses on the comparative analysis for two vital memory management algorithms for the purpose of achieving improved performance of an Embedded Linux System. The operating system chosen for this study is Debian on the hardware of BeagleBone Black. Buddy System and Swapping are the two algorithms selected for comparative analysis. This study assesses how well these improvements work with the Embedded Linux operating system. Because embedded systems are required to function under tight or moderate real-time constraints, the operating systems on embedded devices must minimize latency. Some applications require the smooth execution of multiple concurrent processes at the same time. This objective may be mostly accomplished effectively by making sure that memory management is done well.**

***Index Terms*— System memory, performance, optimization, CPU Utilization.**

I. INTRODUCTION

The memory used by the CPU of an embedded device to perform its functions and keep the device operating is known as embedded memory. The system-on-a-chip may house the memory of the embedded device. The primary memory is the main internal memory of a computer system. The CPU of the system has direct access to the primary memory. Secondary memory is often stored on external storage devices. The CPU does not directly access secondary memory. The CPU may access primary memory quickly even though it is frequently volatile memory. This suggests that the device loses this data when its power supply runs out. Secondary memory's data is secure since it never runs out of power [1]. Compared to primary memory, secondary memory operates more slowly.

Therefore, a good memory management strategy is essential to the overall operation of the system [2]. Embedded systems are strongly tied to the concept of the Internet of Things. A single battery should last an IOT device for several years. Therefore, an operating system should have built-in energy-saving capabilities that maximize the low-power modes that IoT devices support.

IoT devices need to respond rapidly in a variety of scenarios, such as when they detect alarms or receive remote commands. Thus, an operating system should be able to offer soft real-time capabilities at the very least.

Grenze ID: 01.GIJET.12.1.45

It's crucial to keep in mind that, depending on the strategies used, enhancing performance in one area could affect another. Certain memory efficiency strategies, for example, may result in increased copying and CPU usage, which would be detrimental to energy efficiency. In a similar vein, attempting to save more energy by adopting deep sleep modes may result in unanticipated delays in system recovery, which would reduce the system's responsiveness. [3] The task of efficiently and successfully arranging various processing processes falls to scheduling algorithms.

To guarantee that the work is completed on schedule, tasks and procedures are scheduled. While another process must wait for resources like input/output, scheduling enables one process to utilize the CPU to its fullest. CPU scheduling increases the system's speed, equity, and efficiency. When the CPU is not in use, the operating system has to choose a task from the queue that is ready to start [3]. The selection process is managed by a temporary CPU scheduler, which assigns the CPU to one of the memory processes that is ready to run.

The aim of this research is to study the two existing algorithms of memory management and modify operating parameters of both to achieve desired performance. The algorithms selected for this study are swapping and buddy system allocator, which are elaborated in the subsequent sections.

The aim of this research is to evaluate the following memory management strategies on the selected hardware to asses the efficiency of each algorithm. The algorithms of swapping and buddy system are described in the following sections.

II. Swapping

Random Access Memory is used in very small amounts when a software is operating on an embedded device. When there are few applications running, the system uses the available RAM. However, if there are too many apps open or if they use a lot of RAM, the system has problems. If all of the RAM is being used and an application needs extra memory, it will crash [5]. Swap acts as a buffer when the system's RAM runs low. In this instance, when the RAM runs out, the operating system uses some of the hard disk memory and allocates it to the active application.

Swap space uses a section of the hard disk to temporarily store data that would normally be in the computer's physical memory (RAM). This data ends up on the hard drive since the running processes consume all of the RAM. The OS moves some data from RAM to the swap region to provide space for new data. This process prevents the system's memory from running out and is also known as switching out or paging out.

Because data stored in the swap space is more difficult to access than data stored in RAM, system performance is affected. When data is needed again, the operating system returns it to RAM through a process known as "swapping in" [12].

Although swap space offers several advantages, if you use it excessively, your performance will suffer since hard disk data takes longer to access than RAM. Therefore, balancing swap space with an appropriate amount of physical RAM is the best way to ensure optimal system performance.

III. Buddy System Allocator

The buddy system is a memory allocation technique used by computer operating systems to efficiently manage and allocate memory. This method divides the memory into fixed-size blocks, and each time a process requests memory, the system finds the smallest block that can accommodate the specified memory amount. [10]

To provide efficient allocation and minimize fragmentation, it splits memory chunks, called "buddies." When a process is deallocated, its buddy can be merged back into a larger block to reduce wasted space.

Operational systems dynamically allocate memory using a remembrance control method known as the Buddy System. Usually, it is employed in systems that allocate and deallocate memory continuously, including multitasking environments or structures with varying memory needs.

The Buddy System's memory is separated into fixed-length blocks, which are usually 1KB, 2KB, 4KB, and so on. When a memory allocation request is submitted, the processor looks for the block of the right size. If a suitable block is found, it is dispersed to the enquiring way. However, if the required size does not precisely shape any existing blocks, the device allocates a larger block and then splits it into smaller blocks until a properly sized block is obtained.[11]

IV. Methodology

The following experiment has been conducted on a Debian based application on BeagleBone Black, a system having memory with physical address space of 128 KB. We have made process partitions with size of 16KB So,

size of partition for 16 KB process = 32 KB. 5 processes identified to be the most common and processor intensive are run on the system for analysis of Swapping.

A. Swapping

- Swap partion in OS is created.
- Swap in OS is enabled.
- The File System Table (fstab) is configured to enable automatic mounting.
- A swap partition is created with the help of following commands:
  fdisk: A disk partition of type 82 (Linux swap) is created.
  parted: A disk partition of type linux-swap is created.
- 5. The partition is initialized as a swap partition, the command mkswap /dev/sda2 is used.

The recommended size of a swap partition depends on how much RAM you have:
Less than 1 GB of RAM : The swap partition should be at least the same size as the RAM, but no more than double it.
More than 1 GB of RAM : The swap partition should be at least the square root of the RAM, but no more than double it.

B. Buddy system Allocator

This algorithm is implemented on the exact same setup as mentioned earlier. The size of the "Buddy" partition is varied in proportion to the available RAM on the Beaglebone Black board in multiples of 2Kb, 8Kb and 16Kb. Three processes of different RAM requirements were run on the target hardware with Buddy system algorithm, and same three processes' performance has been measured with Swapping algorithm.

V. Results and Discussion

Table I and Table II show a comparative analysis of 2 swap partition settings as mentioned in the methodology section. Table I shows the CPU utilization with standard swap size and Table II shows modified swap size. Here, PID indicates the process ID running on Debian in the same sequence in both the cases. VIRT indicates the virtual memory allocated to each process. RES is the Resident size which us the non-swapped physical memory a task has used. The Shared Memory size (SHR) which is the quantity of shared memory used by a process. It indicates that memory which could be shared with other processes. The standard size of swap partion has been modified to half, double and four times as mentioned in the methodology section.

Table I. Swap Partition 1

PID	VIRT	RES	SHR	%CPU	%MEM
1354	4596853	330368	132035	5	8.3
1985	561524	51562	39557	1	1.3
4452	968045	168589	48527	0.3	4.2
2658	21793	4096	3352	0.3	0.1

Table II. Swap Partition 2

PID	VIRT	RES	SHR	%CPU	%MEM
13245	1137234	279878	6675	10	3.5
65645	156432	12324	59889	8	14.5
2324	968785	185625	8604	0.6	2.3
3458	26853	6525	12536	0.6	0.2

As we can see in the tables above, we can observe that the CPU utilization increases as we increase the swap space to a higher value, but the memory usage decreases significantly, which is the main aim of this study. We can see that virtual memory space per process is differently allocated for the same process, just because of the change in swap partition size.

A. Comparison Between CPU utilization and Memory Consumption in swapping

Figure 1 and Figure 2 show the comparison between CPU utilization and memory consumption in standard and modified swapping algorithms in. graphical nature. Here, X axis represents the number of processes used to evaluate memorty consumption. Two line plots on Y axis represent the memory utilization percentage with red line and CPU utilization percentage with blue line. As we can observe, when swapping partition size is increased, memory utilization and CPU utilization increases significantly, resulting in faster system performance.

Thus the optimal swap size can be derived by analysing the above parameters, based on the particular set of applications running on an embedded Debian system

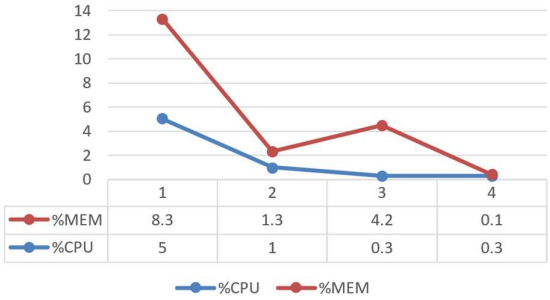

Figure 1. Comparison of CPU and Memory Utilization in standard swapping

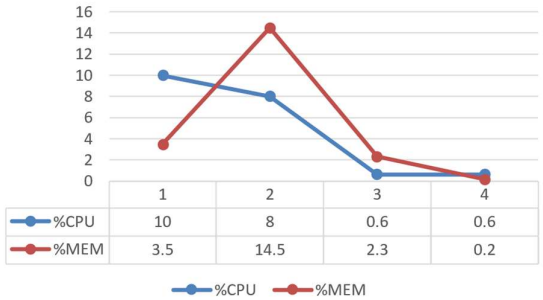

Figure 2. Comparison of CPU and Memory Utilization in modified swapping

Table III shows a comparative analysis of Buddy allocation algorithm with the standard swapping algorithm.

TABLE III. SWAPPING V/S BUDDY SYSTEM ALGORITHM

Utility	Run	Swapping Algorithm		Buddy System	
		Memory	Address	Memory	Address
1	Run 1	52362	45145	32521	35245
	Run 2	125462	152423	90251	112052
2	Run 1	85742	95632	74125	75412
	Run 2	75414	85542	63256	63857
3	Run 1	20320	25232	11020	23002
	Run 2	36524	39856	15698	26359

Here, 3 program utilities were run twice (Run 1 and Run 2) on the target system, and each gives different memory allocation data naturally due to its size and priority of operation. We can observe that Buddy system allocator is much efficient in comparison to standard swapping. We have modified the page size parameters of both the algorithms as compared to the benchmark values. As lesser memory is consumed, system organically runs faster as compared to standard memory management algorithms.

B. Comparison between Standard Memory Allocation and Buddy System allocation

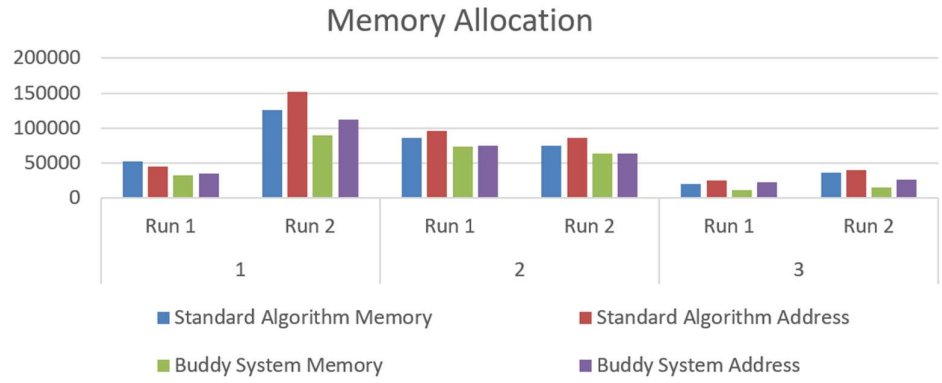

Figure 3. Comparison Between Standard Memory Allocation and Buddy System allocation

Figure 3 shows the comparison of Standard Memory Allocation and Buddy System allocation in graphical nature. X axis represents the two runs of three processes each. Y axis gives bar charts of four parameters, as mentioned in the legends section of the above figure. We can observe that memory utilized is much more optimized with Buddy syetm than standard algorithm of swapping. Thus Buddy system algorithm can be implemented by analysing the above parameters, based on the particular set of applications running on an embedded Debian system. By making smaller memory partitions as compared to standard sizes, we can increase the efficiency of the overall system.

VI. CHALLENGES

Despite the benefits of swap space, there may also be certain drawbacks and challenges, such as performance degradation. Compared to data from swap space on a hard drive or SSD, data from RAM may be accessed far faster. Therefore, a strong reliance on swap space may lead to performance loss as processes wait longer to access data from slower storage.[12]

A hard disk's lifespan is shortened and wear is increased by frequent use of swap space. Furthermore, because SSDs have limited write cycles, frequent swapping may hasten the drive's deterioration.

When the operating system actively uses swap space, other I/O processes on the drive vie for resources. This could lead to disputes and a general slowdown in the system.

The user has less disk space available since swap space takes up space on the hard drive. This can be quite challenging since, once the disk has been partitioned, permanent swap partitions cannot be changed. Memory management is also challenging. Effective memory management is made easier by swap space, but configuring and optimizing it for optimal system performance adds complexity.

Putting confidential information in swap space could lead to unauthorized access if proper encryption and management practices are not followed.

VII. CONCLUSIONS

The increasing use of heterogeneous embedded systems with multi-core CPUs and GPUs has made it harder to meet the throughput needs of digital signal processing applications. Code that has been manually optimized to satisfy system-level memory requirements is inefficient and prone to mistakes, which may result in underutilization of computer resources or delays in development. By conducting the experiments with varied parameters for the two selected memory management algorithms, we can utilize the CPU to its maximum optimum capacity. In embedded systems architecture, synchronous dataflow representations are frequently used for signal and information processing.

Since the 1960s, dynamic memory allocation has been a crucial part of computer systems, and techniques for streamlining and optimizing it have been developed. In order to control fragmentation issues and minimize empty space, allocators keep an eye on the free and used memory regions. The majority of studies on dynamic memory allocators concentrate on examining typical memory waste behaviour and allocation times. We have analysed performance of swapping individually as well as with comparison with Buddy system allocator. We can thus conclude that with the appropriate swap area or Buddy size, maximum memory optimization of Embedded RAM can be achieved.

REFERENCES

[1] K. Siva Sundari, R. Narmadha, and S. Ramani, "A Classy Memory Management System (CyM2S) using an Isolated Dynamic Two-Level Memory Allocation (ID2LMA) Algorithm for the Real Time Embedded Systems," International Journal of Electrical and Electronics Research, vol. 10, no. 2, pp. 387–393, Jun. 2022, doi: 10.37391/IJEER.100254.

[2] G. K. Adam, N. Petrellis, and L. T. Doulos, "Performance assessment of linux kernels with preempt_rt on arm-based embedded devices," Electronics (Switzerland), vol. 10, no. 11, Jun. 2021, doi: 10.3390/electronics10111331.

[3] J. Han and S. Lee, "Performance Improvement of Linux CPU Scheduler Using Policy Gradient Reinforcement Learning for Android Smartphones," IEEE Access, vol. 8, pp. 11031–11045, 2020, doi: 10.1109/ACCESS.2020.2965548.

[4] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-Optimizing Performance and Memory Footprint Via Integrated CPU/GPU Memory Management, an Implementation on Autonomous Driving Platform," in Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, Institute of Electrical and Electronics Engineers Inc., Apr. 2020, pp. 310–323. doi: 10.1109/RTAS48715.2020.00007.

[5] M. Bazzaz, A. Hoseinghorban, and A. Ejlali, "Fast and Predictable Non-Volatile Data Memory for Real-Time Embedded Systems," IEEE Transactions on Computers, vol. 70, no. 3, pp. 359–371, Mar. 2021, doi: 10.1109/TC.2020.2988261.

[6] M. Carminati and G. Scandurra, "Advances in measurements and instrumentation leveraging embedded systems," Dec. 01, 2021, American Institute of Physics Inc. doi: 10.1063/5.0070073.

[7] A. Varshney et al., "Challenges in Sensors Technology for Industry 4.0 for Futuristic Metrological Applications," Mapan - Journal of Metrology Society of India, vol. 36, no. 2, pp. 215–226, Jun. 2021, doi: 10.1007/s12647-021-00453-1.

[8] V. Alonso, A. Dacal-Nieto, L. Barreto, A. Amaral, and E. Rivero, "ScienceDirect ScienceDirect Industry 4.0 implications in machine vision metrology: an overview," 2020, [Online]. Available: www.sciencedirect.comwww.sciencedirect.com

[9] H. Jylhä and J. Hamari, “Development of measurement instrument for visual qualities of graphical user interface elements (VISQUAL): a test in the context of mobile game icons,” User Model User-adapt Interact, vol. 30, no. 5, pp. 949–982, Nov. 2020, doi: 10.1007/s11257-020-09263-7.
[10] G. K. Adam, “Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers,” 2021, doi: 10.3390/computers10050064.
[11] H. Jang, K. Han, S. Lee, J. J. Lee, and W. Lee, “MMNoC: Embedding Memory Management Units into Network-on-Chip for Lightweight Embedded Systems,” IEEE Access, vol. 7, pp. 80011–80019, 2019, doi: 10.1109/ACCESS.2019.2923219.
[12] L. Shaofeng, Q. Lei, and Y. Mengfei, “Verification of Design of Memory Management Module for Embedded System Based on Coq,” in Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS, IEEE Computer Society, Oct. 2020, pp. 44–47. doi: 10.1109/ICSESS49938.2020.9237731.