Real Time Memory Regeneration in Constrained Embedded Systems

Lakshmi Prasanna, Kashmira Kapoor, Tushar Vrind, Diwakar Sharma and Raju Udava
Samsung Semiconductor India R&D
Samsung Electronics,
Bangalore, India

Abstract— **Embedded devices are mostly used in products where amount of available memory is limited and often we encounter situations where the installed user applications end up consuming more memory than designed for; which is reported as a system failure. The memory requirements of the embedded system's applications frequently overrun the initial estimates & could lead programmers to over budget the memory usage easily by over 20%. We discuss a software based novel approach which can be deployed in a Real Time Operating System (RTOS) for an embedded processor, which does not have a demand paged and virtual-physical address translation architecture (Memory Management Unit (MMU)-Less). In this paper, we propose the algorithm to be adopted by the RTOS Scheduler, which finds the transient unused block of memory to be compressed, and allocates the remainder block to the task which may need additional stack or dynamic memory. The proposed algorithm discusses the steps of compression, decompression and management to utilize the regenerated memory. With this optimizations, we can regenerate memory dynamically (theoretically to almost double) and system failure rate can be improved based on above probability improvement.**

I. Introduction

With the increase in embedded products especially fueled by Internet of Things (IOT) and wearables, designers are constantly trying to optimize the power and cost of components used in the product design. Memory being one of the most vital elements, the reduction in usage of memory leads to both power and cost optimization.

Over the Top (OTT) applications once installed on such embedded systems can very frequently cause memory constraints cropping up in the system leading to failures and crashes.

In complex architectures like for Personal Computers (PC), a Memory Management Unit (MMU) handles the address space and through virtualization and demand paging, is able to load pages into the main memory to address the dynamic memory of an application executing on the processor. However Embedded devices we don't use virtual memory and demand paging concepts due to the associated inherent latencies.

Typically, to counter such unforeseen needs for dynamic memory in an embedded system, the designers over budget the memory requirements easily by 20% of actual estimated requirement, which on one hand is not fool proof and on the other creates portions of unused memory during the life span of the product.

In this paper, we describe a novel technique to compress and reduce the size of allocated memory, which is un-accessed for a significant amount of time. The compressed memory is decompressed dynamically before it is accessed again. The regenerated memory region is linked together, and utilized as either a dynamic heap or for task's stack. Thus increasing the amount of dynamic memory available in the system.

Through mathematical modelling, we show that we can theoretically double the amount of available dynamic memory in a system, thus reducing the system failure rate by almost the same probability as we increase the available dynamic memory in the system.

The rest of the paper is organized as follows. Section II discusses the proposed scheme for compression, regeneration and reuse. Section III defines the abbreviations used, Section IV discusses the mathematical model and probabilistic model for system improvement. Section V discusses the conclusions.

II. Proposed Scheme – Memory Regeneration

We propose to add a new scheme in RTOS scheduler to maintain 'Set' of in-frequently scheduled tasks, at any given point of time. Such that when system available memory goes below a new defined Threshold System Memory Capacity (SMC'), then the data and stack for all the tasks in 'Set' is compressed sequentially and the remaining space is treated as freed-space and used as a new memory pool. The same is described in Fig.1, and the steps described below.

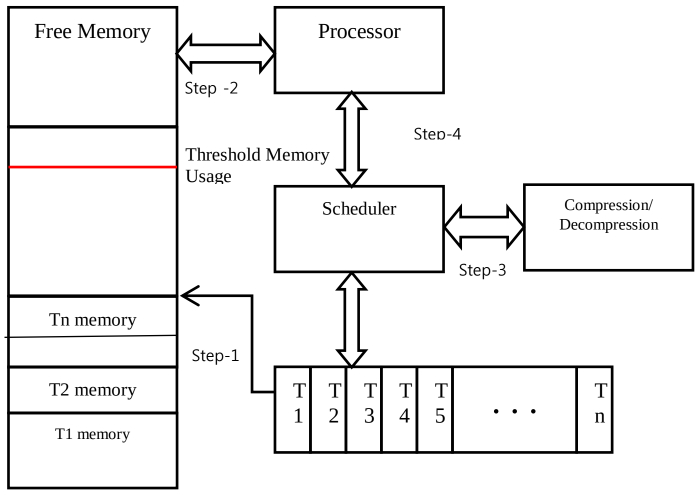

Fig. 1. Memory Regeneration Overview

Step 1: At start up, scheduler will allocate the required amount of memory to tasks like T1, T2, T3… Tn, and scheduling frequencies of tasks and memory availability of system is continuously monitored. System maintains a threshold limit for availability of dynamic memory (denoted by red line).

Step 2: Once the system has used more memory than defined by threshold limit, it initiates Step 3.

Step 3: Tasks to be compressed are identified; stack, dynamic memory and code area is compressed for each of such task. The remaining area is linked to the memory pool which can then be used for dynamic usage. Scheduler uses compression/decompression while scheduling the tasks from here onwards (until system memory usage drops below threshold limit)

Step 4: Once a task, which was compressed, is to be scheduled, scheduler decompress the memory region of compressed task while assigning to the processor.

Step 3 and Step 4 will be repeated until the system memory usage drops below the designed threshold limit.

The proposed mechanism will find the candidate task for compression based on metrics as scheduling frequency SFi' (no. of times the task is scheduled) of task Ti and System Memory Capacity (SMC'). SMC' is basically the heap memory size remaining at any point of time during execution. Thus, let us assume SFi' is the scheduling frequency for task Ti and SMC' is the threshold of system memory capacity. Compression algorithm is triggered to determine the candidate task Ti for compression based of their respective SFi'. The Task Control Block (TCB) for respective task is updated with the compression information. As shown in Fig.2, the compressed sections of the task Tci denoting Code Area, TDi, denoting Data Area and TSi, denoting Stack Area are stored at the same memory location after compression. The saved areas are linked to a usable available memory pool from where any memory allocation can be met. Whenever the compressed task is scheduled, the decompression algorithm will decompress the corresponding sections of the task Ti. If the same memory location was currently not in use, then decompression algorithm will simply decompress those sections in the respective areas. If the grey block near the compressed block is allocated, then algorithm will fail and a system failure would finally be triggered.

Algorithm 1: Approach 1

```
Input: SFi' and SMC'
Output: Success or Error
WHILE (1)
        IF (SMC' has reached threshold)
             Determine the candidate task Ti' based on SFi'
             Compress the task Ti' and update Task Control Block
             Store the compressed sections at same memory location
        END IF
        IF (Compressed Task Ti' is scheduled)
            /* decompress the corresponding sections of task Ti'*/
             IF (Memory was not currently in use)
                  Store decompressed task at same memory location
         ELSE
             Announce System Failure
             ENDIF
END IF
```

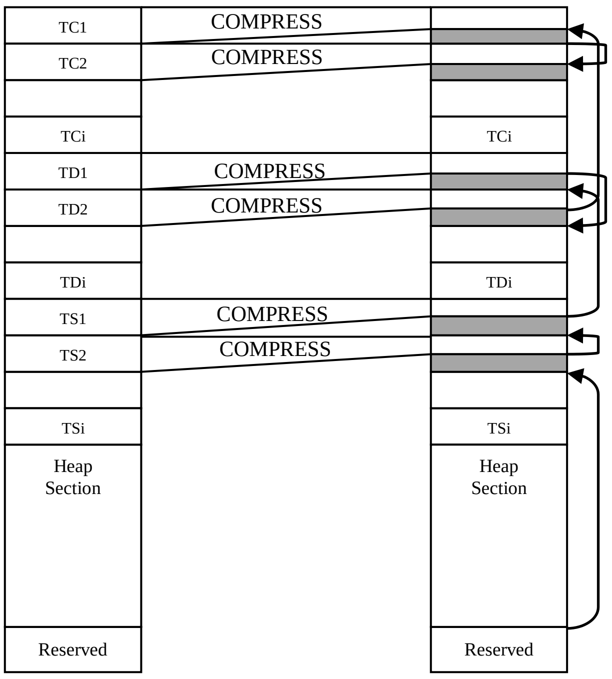

Fig. 2: Memory generation, Memory Map

III. Abbreviations and Acronyms

N = No. of total tasks in system

SFi = No. of times each Task is scheduled

SMC (System Memory Capacity) = Heap memory size remaining during program execution

TH = Threshold of being called in-frequently scheduled

S = Total number of events in the system

Ti = iTh number of Task in System

Pi = iTh number of Process in System

TCi = Code section for task Ti

TDi = Data section for task Ti

TSi = Stack for task Ti

IV. Mathematical Model

Through mathematical modelling, we will try to show theoretically how system failure is decreased by half when we increase the available dynamic memory by double using our proposed method in the system.

For the Improvement Analysis, we use the Probability of System Failure as measure to show the impact of our proposed solution.

In the Base system (without our proposed solution): If p is the probability of a requested block of size > available heap, then Probability of system failure = p.

In a system with our proposed solution,

Let us assume,

For M tasks SFi <= TH, where (M <= N).

Thus, we define that any Task scheduled for less than m number of times, could belong to the finite set {SFi'-Infrequent},

Further, a task can get scheduled when there is an event for that Task and that event's probability = 1/S.

Thus, the probability of task getting scheduled = 1/S

Probability of a task getting allocated from set {SFi'-Infrequent} = 1/M

Probability of a block getting allocated from set {SFi'-Infrequent} finite set of tasks, and the same task getting scheduled = (1/M) * (1/S)

Thus the probability of failure of the new System = p * (1/ (M*S))Thus we see that as Number of idle Tasks (M), and Number of Events (S) increase in the system, the probability of failure decreases drastically.

For computational purposes, we have chosen the system in which *p* is 0.6, Number of events *S* to be varying from 10 - 70 and number of tasks which are not frequently scheduled, *M* varying from 5-35

In the graphs shown below, we see that the probability of system failure drastically reduces as the number of idle tasks and number of events increase in the system

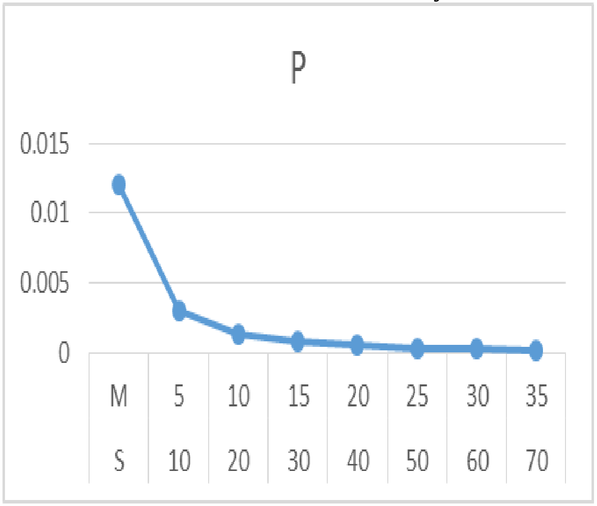

Fig. 3: Probability of system failure vs Number of Events and Idle tasks

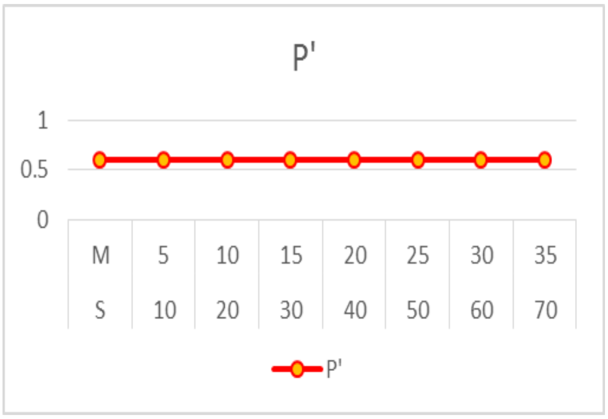

Fig. 4: Probability of system failure without proposed approach

V. CONCLUSION

In embedded system design we seldom use complex OS. Code is statically linked and stack space are pre-allocated. Due to this stack and memory are generally over budgeted. We have seen, in embedded designs, depending on the scenarios some of the task (and associated) stack memory is not used for a long time during the lifecycle. We have new scheme in Task Scheduler to maintain 'Set' of in-frequently scheduled Tasks, at any given Point of Time When System Available Memory goes below a Threshold (SMC'), then task/data/stack for all Tasks in the 'Set' is compressed and freed-space is used as a New Memory Pool If we compress these stacks/code area, and reuse at run time for dynamic allocation, then the probability of system failure decreases from p $\rightarrow$ p * (1/(M*S)). We have seen that during the life cycle of software, transient memory allocations are very high, and it leads to system failures due to memory allocation failure at run time. Through the optimizations in this paper, we can regenerate Memory (theoretically to almost double) and system failure rate can be improved based on above probability improvement

REFERENCES

[1] Sicheng Tang; Huailiang Tan; Lun Li, "A novel memory compression technique for embedded system"
Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference, pp - 287 - 292.

[2] O. Ozturk, M. Kandemir, and M. J. Irwin, "Using Data Compression for Increasing Memory System Utilization" in Volume: 28, Issue: 6, , 2009, pp. 901 - 914.

[3] J.-J. Hwang, Y.-C. chow, P. D. Anger, and C.-Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. SIAM Journal of Computing 18(2):244-257 1989. [7] B. Keinhuis, E. Depretter, K. V~sek and P. van de; Wolf. An

[4] C. D. Benveniste, P. A. Franaszek, and J. T. Robinson, "Cache-memory interfaces in compressed memory systems," IEEE Trans. Comput., vol. 50, no. 11, pp. 1106–1116, Nov. 2001.

[5] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, “Dynamic management of scratch-pad memory space,” in Proc. 38th Conf. Des. Autom., 2001, pp. 690–695.

[6] O. Ozturk, M. Kandemir, and M. J. Irwin, “Using data compression in an MPSoC architecture for improving performance,” in Proc. 15th ACM GLSVLSI, 2005, pp. 353–356.

[7] L. Benini, A. Macii, E. Macii, and M. Poncino, “Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation,” IEEE Des. Test Comput., vol. 17, no. 2, pp. 74–85, Apr.–Jun. 2000.

[8] S. Coleman and K. S. McKinley, “Tile size selection using cache organization and data layout,” in Proc. ACM SIGPLAN Conf. PLDI, 1995, pp. 279–290.

[9] O. Ozturk, H. Saputra, and M. Kandemir, "Access Pattern-Based Code Compression for Memory-Constrained Embedded Systems"- Design, Automation and Test in Europe (INSPEC Accession Number: 8394771).