WINTECHCON 2018

September 28, 2018

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

Problem Statement

- *Embedded devices are mostly used in context where amount of available memory is limited.*
- *Embedded devices we don't use virtual memory and demand paging concepts also due to the associated inherent latencies.*
- *Tasks/processes/threads are pre-allocated memory for code and stack area at system boot-up.*
- *With the increase in embedded products especially fueled by Internet of Things (IOT) and wearables, Over the Top (OTT) applications once installed on such embedded systems can very frequently cause memory constraints .*
- *In complex systems like Modem Processors, it is not possible to estimate worst case scenarios of memory use.*

Problem Statement

- *Its very common to have memory crunch issues (allocating from heap/deeper function calls – call stack); which is reported as a system failure.*

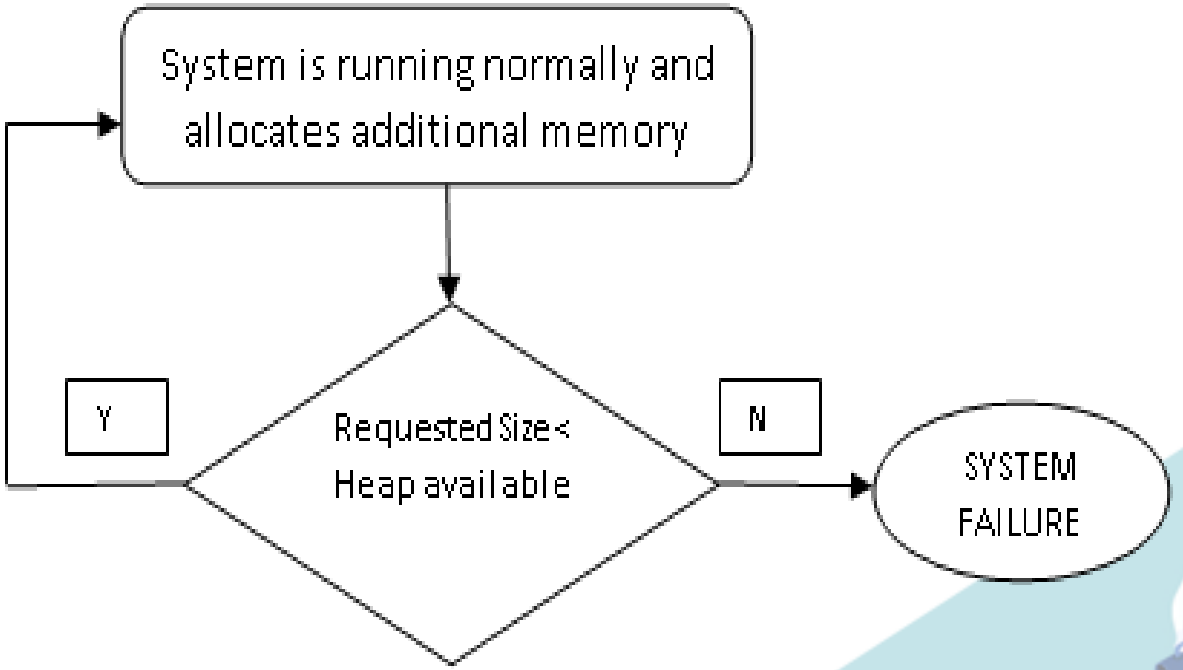
- To handle memory crunch issues in a graceful manner, typical solutions of increasing or reassigning stack or heap could turn out to be costly, from development and/or production cost perspective. Leading to practice to over budget the stack usage by ~ 20%.

- In most of the scenarios, only a specific part of the code is run, and most of the programmed logic lies idle.

- For example in a Multi-RAT Mobile Modem processor, code/stack area for one RAT could only be getting used (depending on the signal conditions).

Problem Statement

The problem is explained in the figure below:



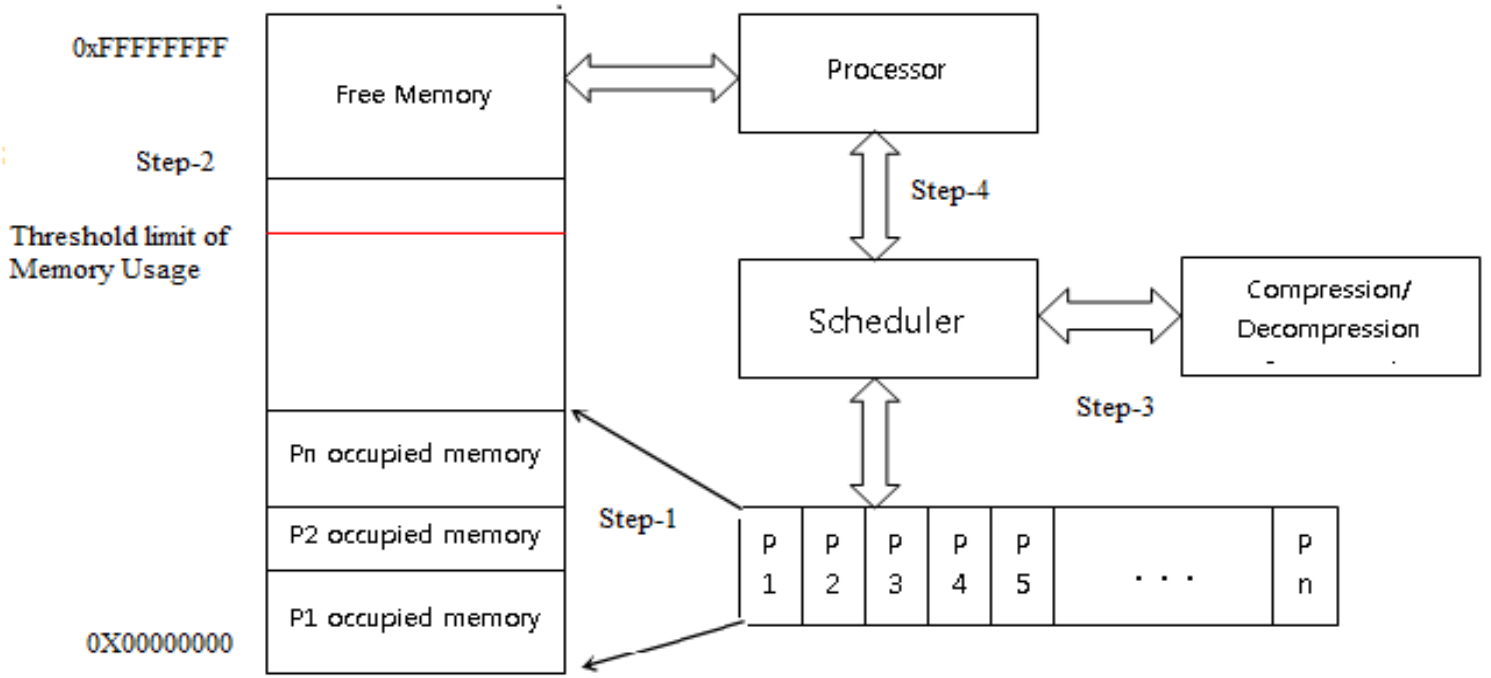
Solution approach

- As Discussed above, in most scenarios, any programmed code is in static state of mobility ~ 50% of its lifecycle.
- This intelligence could be utilized to design a software/hardware approach to Regenerate memory in an embedded system. Our disclosure is centered on this idea.
- In our disclosure, we discuss a software based approach which can be deployed in a RTOS based embedded processor, which doesn’t have a demand paged and virtual-physical address translation architecture (MMU-Less).
- 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.

Solution approach

- *Step 1*: At start up, scheduler will allocate the required amount of memory to tasks/Processes like P1, P2, P3… Pn, 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 repeat until the system memory usage drops below the designed threshold limit.

Solution approach



Solution approach

- The symbols TCi, TDi and TSi represent the code section, data section and stack for the task Ti.
- SFi’ is scheduling frequency or no. of times the task is scheduled of task Ti.
- SMC is System Memory Capacity which is basically the heap memory size remaining at any point of time during program execution.
- The threshold SMC’ will trigger the compression algorithm, which will determine the task to be compressed based on SFi’. The compression is performed sequentially.
- The SFi’s are reset once the system reaches the SMC’. The TSi of task Ti is compressed completely except the scheduling information (uncompressed) which is essential for scheduler.

Solution Algorithm

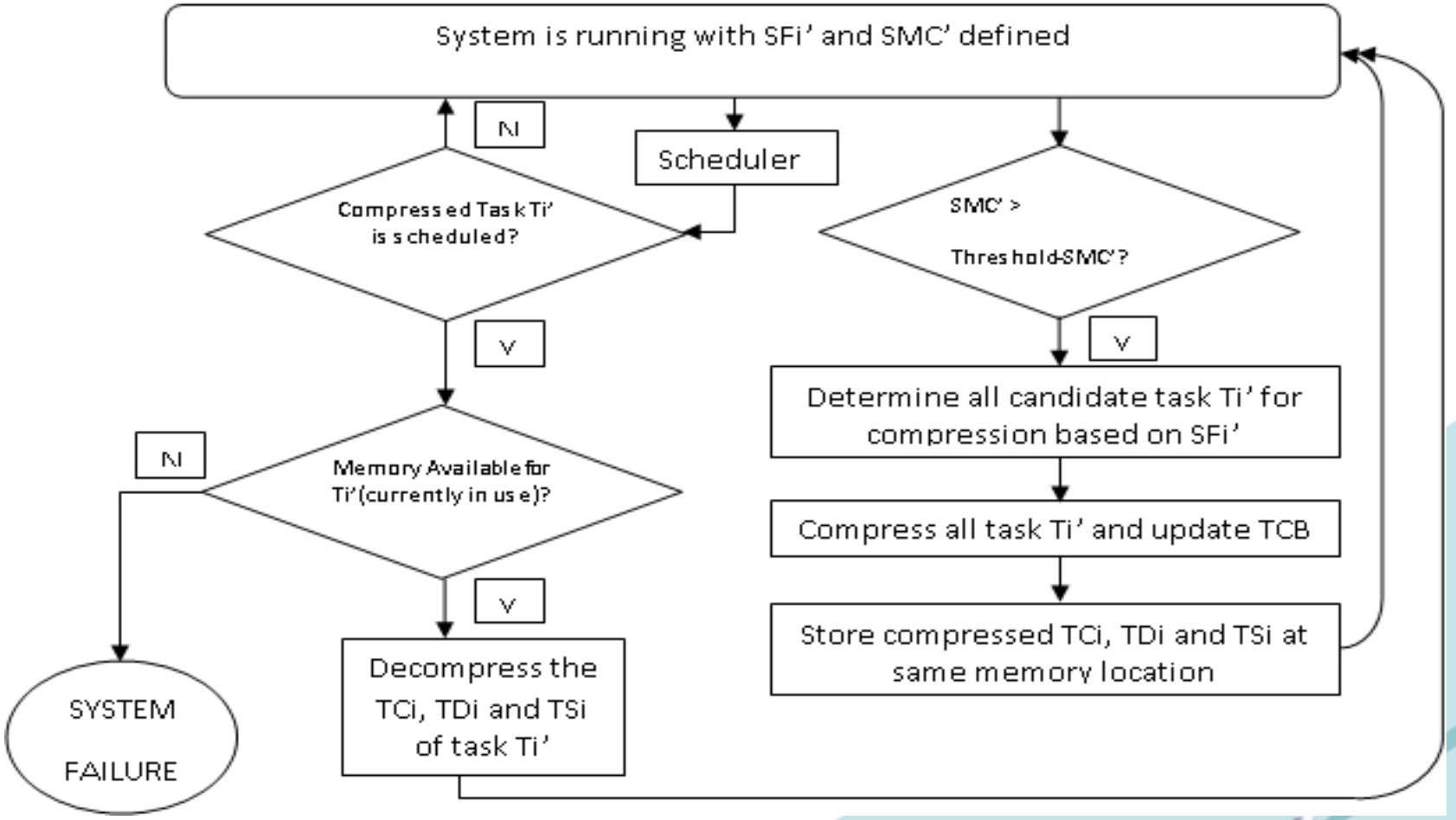
Input: SFi’ and SMC’

Output: Success or Error

Assumption: System has reached the threshold SMC’.

```
WHILE (1)
        IF (SMC’ has reached threshold)
                Determine the candidate task Ti’ for compression based on SFi’
                Compress the task Ti’ and update TCB
                Store the corresponding 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
END WHILE
```

Solution approach



Memory generation, Memory Map with our approach

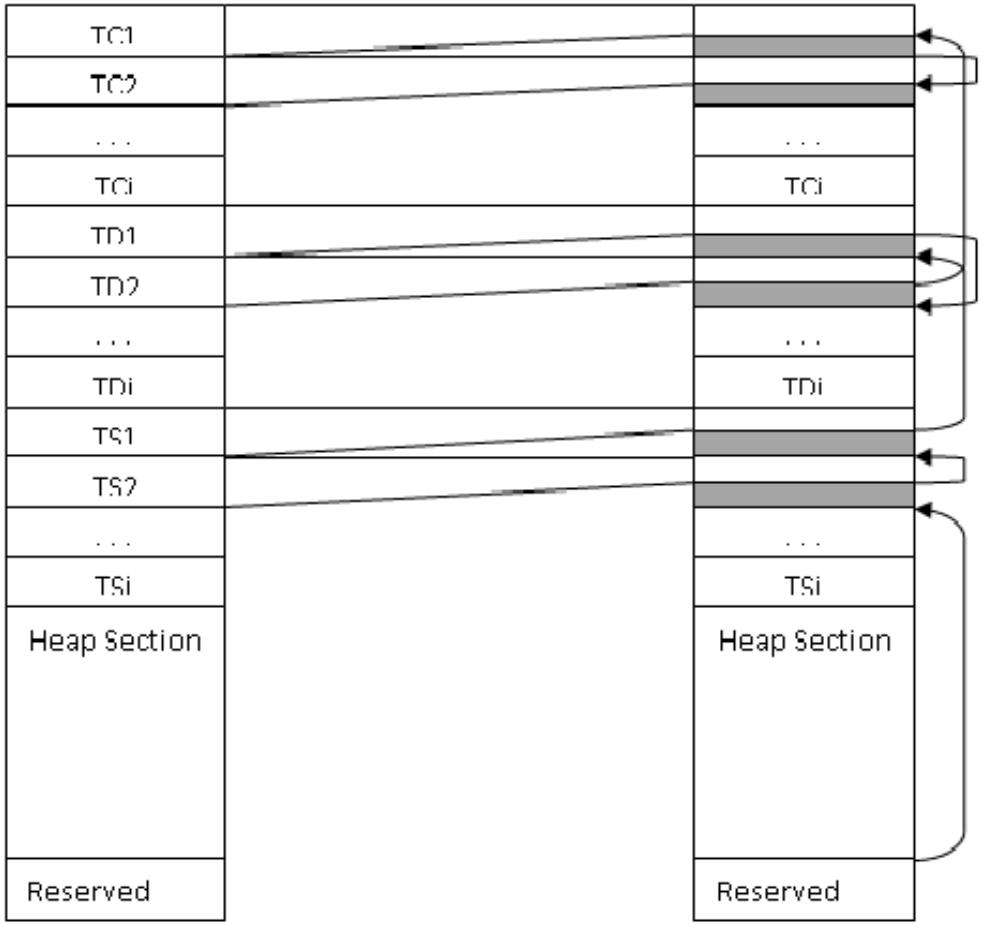


Fig: 1 – Memory map for approach

Experimental Results /Mathematical Model

- 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, if there are *N* tasks, in the system out which *M* tasks fall in the category of SFi' <= Threshold of being called in-frequently scheduled. Where we define that any Task scheduled for less than *m* number of times, could belong to the finite set {SFi'-Infrequent}, where (*M* <= *N*).
- Further, a task can get scheduled when there is an event for that Task and that event's probability = *1/S* (*S* is total number of events in the system). Thus, the probability of task getting scheduled in = *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.

Experimental Results

- 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

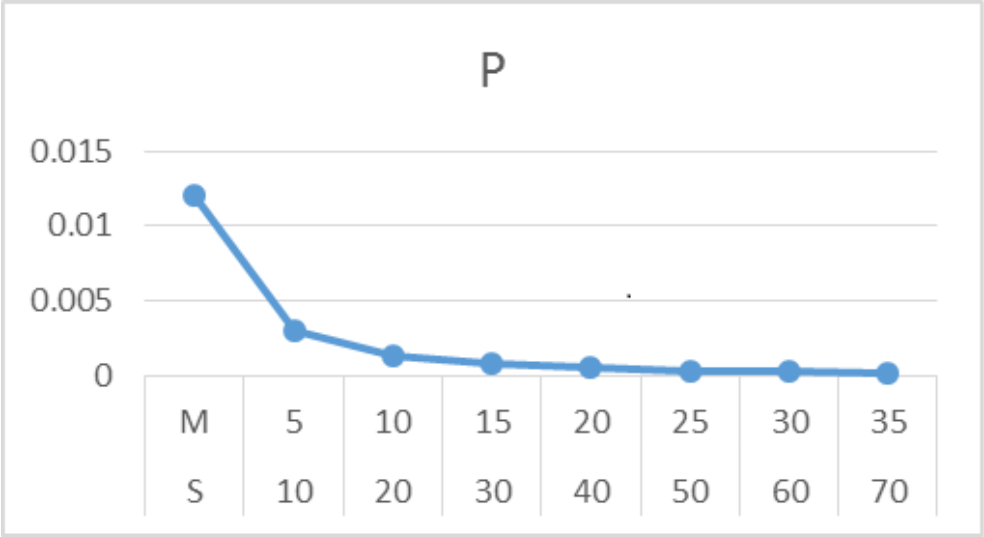


Fig. 3: Probability of system failure vs Number of Events and Idle tasks

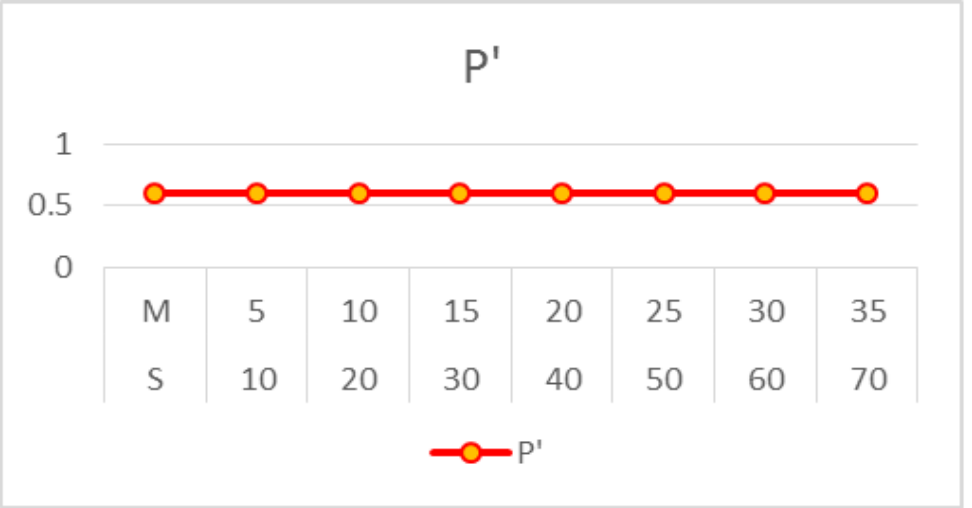


Fig. 4: Probability of system failure without proposed approach

Conclusions

- In embedded system, depending on the scenarios some of the task (and associated) stack memory is not used for a long time during the lifecycle.
- We discuss 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 = 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.