A Low Overhead Dynamic Memory Management System for Constrained Memory Embedded Systems

Supratim Das
IIIT-Delhi
New Delhi, INDIA
Email Id:supratim12107@iiitd.ac.in

Amarjeet Singh
IIIT-Delhi
New Delhi, INDIA
Email Id:amarjeet@iiitd.ac.in

Surinder Pal Singh
STMicroelectronics India Pvt. Ltd.
Greater Noida, INDIA
Email Id:surinder-pal.singh@st.com

Amit Kumar
GreaterNoida
INDIA
Email Id:amiit@gmail.com

Abstract - Embedded systems programming often involve choosing the worst case static memory allocation for most applications over a dynamic allocation approach. Such a design decision is rightly justified in terms of reliability, security and real time performance requirements from such low end systems. However with the introduction of public key cryptography and dynamic reconfiguration in IP enabled sensing devices for use in several "Internet of Things" applications, dynamic memory allocation in embedded devices, is becoming more important than ever before. While several embedded operating systems like MantisOS, SOS and Contiki provide dynamic memory allocation support, they usually lack flexibility or have relatively large memory overhead. In this paper we introduce two novel dynamic memory allocation schemes, ST_MEMMGR (without memory compaction) and ST_COMPACT_MEMMGR (with memory compaction), with a close compliance with the libc memory allocation API. Both designs take into account the very limited RAM (1KB - 64KB) in most microcontrollers. Experimental results show that ST_MEMMGR has a 256 - 5376 bytes lesser memory overhead than similar non-compaction based open source allocators like heapLib and memmgr. Similarly, ST_COMPACT_MEMMGR is observed to have 33% smaller memory descriptor as compared to Contiki's managed memory allocator with similar performance in terms of execution speed.

Keywords - Dynamic Memory Management, Memory Fragmentation, Memory Compaction, Microcontrollers, Embedded Systems, WSN

NOMENCLATURE

DTLS – Datagram Transport Layer Protocol
IoT – Internet of Things
IP – Internet Protocol
TLS – Transport Layer Protocol
WSN – Wireless Sensor Networks

I. INTRODUCTION

Embedded devices are proliferating more than ever before in our lives. With Internet of Things (IoTs) being pitched as the future, whereby all things around us will have a computing and a communication interface, demand for such embedded devices is only going to grow. Typical requirements from such devices include low cost and low power consumption. These requirements eventually lead to designs with low end microcontrollers and small memory footprint.

As these devices are connected to the internet, previous means of secure communication employed in WSN are no longer sufficient, and the state of the art standard security protocols must be used. Standard transport layer security protocols like TLS, DTLS and network layer security protocols like IPSEC make use of public key cryptography, that is both compute and memory intensive for low end microcontroller based embedded devices. In addition to the memory and compute constraint, cryptographic algorithms sometimes presents a level of complexity not well understood by many embedded systems programmers. Several of these reasons motivate the need to use proven and existing implementation rather than rolling out an implementation from scratch. However it was observed that majority of the existing implementation heavily use dynamic memory allocation. In embedded systems design and development, dynamic memory management is generally avoided, hence most C runtime for low end embedded platforms omit the support for dynamic memory allocation. Unavailability of dynamic memory allocation support makes it difficult for easy portability of the existing network security stacks and cryptographic libraries for the new embedded platforms, and hence increases the time to market.

Dynamic memory allocations for low end embedded systems require special consideration, as these devices are often constrained in terms of memory, computation and power requirements. In this paper various aspects of dynamic memory management for constrained memory embedded devices is studied and scrutinized in detail. We propose two new dynamic memory allocators – ST_MEMMGR and ST_COMPACT_MEMMGR. ST_MEMMGR is based on the sequential fit algorithm with the first fit allocation scheme. It does not support memory compaction and hence suffers from memory fragmentation. Further, this proposed approach is completely compliant with the standard libc dynamic memory management API thus ensuring easy portability. ST_COMPACT_MEMMGR on the other hand supports memory compaction and hence does not suffer from fragmentation problem. While, in order to support memory compaction, API for ST_COMPACT_MEMMGR deviates slightly from compliance, it is still very closely related.

The rest of the paper is organized in the following eight sections. Related work is discussed in Section II. In Section III, an overview of the existing embedded dynamic memory allocation schemes is presented. The motivation behind the work is presented in Section IV, which is followed by Section V which elaborates the design and implementation of the proposed dynamic memory allocators are described and illustrated in detail. Experimental results are presented in Section VI, followed by future scope in Section VII. The conclusion is presented in Section VIII.

II. RELATED WORK

Dynamic memory management is not a new concept and has been widely explored in the history of computing. In his book “The art of computer programming” [1] Donald E. Knuth has described dynamic storage allocation in considerable detail, with an elaborate discussion on various dynamic storage allocation schemes like sequential fits, segregated free lists, buddy systems and bitmap fits. Further in [2]Puaut et al. explores the real time performance of various dynamic memory allocation algorithms. Considerable research efforts [3], [4], [5], [6], [7]

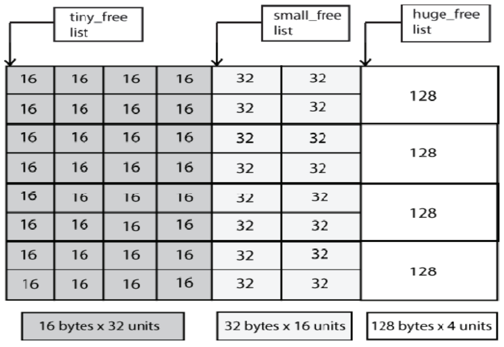

Fig. 1. Fixed sized blocks of 16, 32 and 128 bytes in SOS

have been done in the past on dynamic memory allocation to improve the system in terms of time complexity and fragmentation.

Despite being a matured problem, with a rich research background, dynamic memory allocation in microcontrollers and

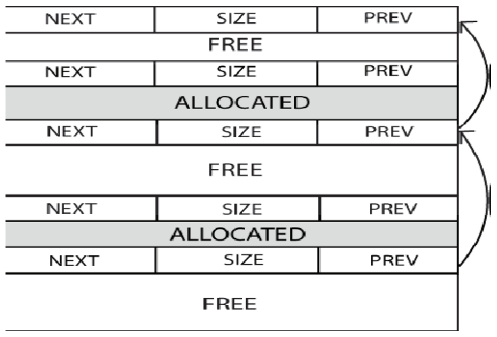

Fig. 2 Doubly linked list of free blocks in MantisOS

similar memory constrained embedded systems have not received much attention. The use of dynamic memory allocation in embedded systems is often avoided by embedded system programmers and a worst case compile time static allocation is a preferred choice. As most low cost microcontrollers have a small RAM (1KB – 64KB), limited computing power and lack advanced hardware features like a MMU and memory protection mechanisms so a compile time static memory allocation is rightly justified in terms of reliability, security and realtime performance requirements. Specifications like ZigBee IP [8] introduced public key cryptography and advanced features like dynamic loading of modules [9], [10], [11], dynamic memory management is becoming increasingly important. The very fact that many modern sensor network OSes like MantisOS [12], SOS [10] and Contiki [13] have support for dynamic memory allocation reinforces this claim.

Although dynamic memory allocation support is available as standalone open source libraries [14], [15] and in many modern embedded and sensor network OSes, they suffer from some limitations that restrict their applicability. One such limitation is a non-libc compliant API, which makes it difficult to port existing code across different platforms. Further in many designs, the descriptors used for managing the memory is often too large, when compared to the limited RAM in microcontrollers.

III. EXISTING SCHEMES

Operating system (OS) support for WSNs plays a central role in building scalable distributed applications. An OS aids in the development of applications by presenting a variety of services and abstractions like process management, network stack, hardware abstraction layer and dynamic memory allocation. Although OSes like TinyOS and RETOS does not implement dynamic memory allocation, many popular OSes like SOS, MantisOS and Contiki has explicit support for dynamic memory allocation. In this section a brief review is made on the dynamic memory allocation schemes employed in SOS, MantisOS and Contiki.

A. Dynamic Memory Allocation in SOS

SOS requires a dynamic memory allocation and uses the power of two free lists [16], [10]. It consists of 16 X 32, 32 X 16, 128 X 4 bytes memory blocks. The total size of memory bock is 1536 bytes. Similar to the segregated free lists, it only requires *O(1)* execution time to find an appropriate block, but suffers from a serious internal fragmentation problem. In addition to this, if consecutive 5 allocation requests of 128 bytes are made then the allocator will fail, despite when enough free space is available. Fig. 1 shows the memory allocation structure of the SOS.

B. Dynamic Memory Allocation in MantisOS

Fig. 2 shows the memory management structure employed in MantisOS and Nano-Qplus [16], [12]. The memory is managed as a doubly linked list and the allocation is performed using a sequential fit algorithm with best fit policy. Correspondingly this memory management scheme has an allocation runtime complexity of *O(n)*. The allocation scheme of MantisOS is similar to open source embedded dynamic memory allocation solutions like heapLib [14] and memmgr [15], with but slight variations like employing a singly linked list of free blocks, rather than a doubly linked list, allocation using multiples of a fixed size blocks and mechanisms for aligned memory access on a word boundary.

C. Dynamic Memory Allocation in Contiki

The managed memory allocator (mmem) provides a dynamic memory allocation service similar to any other allocation schemes. Its main distinction, however, is that it uses a level of indirection to enable automatic defragmentation of the managed memory area [17]. Every managed memory block is represented by an object of type struct mmem, as shown in Fig. 3. The mmem library organizes the struct mmem objects in a list named mmem_list. In the struct mmem object, ptr refers to the size of the allocated chunk in the contiguous memory pool reserved for the mmem library. The (mmem) managed memory

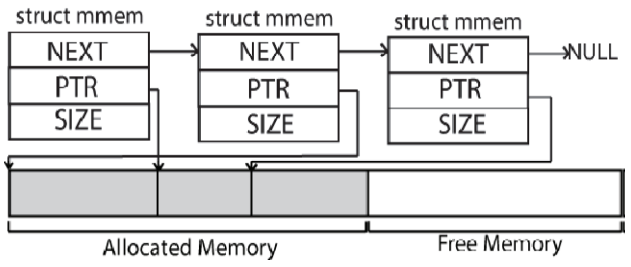

Fig. 3. A singly linked list of struct mmem with PTR refers to the current allocated block, and SIZE denotes the size of the allocated block.

allocator's support for memory compaction however leads to a non-libc compliant API, this makes it difficult to port existing applications.

IV. MOTIVATION

With the advent of IP enabled sensor devices for "Internet of Things" applications, security has taken a place of paramount importance. In order to support state of the art network security protocols and cryptographic suites it is required to have a generic dynamic memory allocation support.

Often there is a tradeoff involved between performance and flexibility while designing memory allocators. For example, the dynamic memory allocator in SOS has a tightly bounded execution time and very low memory overhead, but it's not nearly as flexible as the allocation schemes of MantisOS and Contiki. Similarly with a more general and flexible implementation, MantisOS and Contiki has a relatively large memory overhead and variable execution time.

TABLE I. COMPARISON OF MEMORY DESCRIPTOR SIZE PER ALLOCATION

Allocation Scheme	Descriptor Size	Ext. Fragmentation
MantisOS	12 bytes	YES
Contiki	12 bytes	NO
heapLib	5 bytes	YES
memmgr	8 bytes	YES
ST_MEMMGR	2 bytes	YES
ST_COMPACT_MEMMGR	4 bytes	NO

In this paper the design of dynamic memory allocation is approached strictly from the perspective of a small memory device. The design of the proposed allocation scheme leverages the fact that most wireless sensor nodes have limited RAM, in order to have reduced memory overhead, than existing implementations. A comparison of the per allocation memory descriptor sizes is given in Table I.

V. PROPOSED DESIGN

Wireless sensor nodes are severely resource constrained devices generally featuring a microcontroller with very limited RAM (4KB - 16KB). The standard size of a pointer in C language is of 32 bits capable of addressing 4GB of memory. In case of embedded devices, it is possible to reduce the size of the memory descriptors by replacing the 32 bit absolute pointers with a reduced size offset relative to a base address. The obvious disadvantage of this approach is reduced scalability as it is not possible to address any arbitrary size memory, but it is often practical for small memory embedded devices.

In this section the two memory allocator designs are proposed following the above laid concept. The first design ST_MEMMGR follows a standard sequential fit approach. The second design ST_COMPACT_MEMMGR is a fragmentation less design by employing the technique of memory compaction.

A. ST_MEMMGR

The ST_MEMMGR follows the sequential allocation algorithm with the first fit allocation policy. The entire memory is maintained as a singly linked list of allocated and free blocks. During the allocation operation, the entire list is searched sequentially in order to find the first free block large enough to satisfy the allocation request. It employs split and coalesce operations to best suite an allocation request. The allocation operation takes O(n) execution time since it requires a sequential scan of the memory descriptors while the free operation involves only a reset operation on the allocation bit thus having an execution complexity of O(1).

The size of the memory descriptor is 16 bits: 1 allocation bit and 15 size bits (2 bytes). The allocation bit determines whether a block is flagged as an allocated block or a free block. The 15 size bit is capable of handling a single block of 32K allocation frames and also acts as a pointer to the next block. The

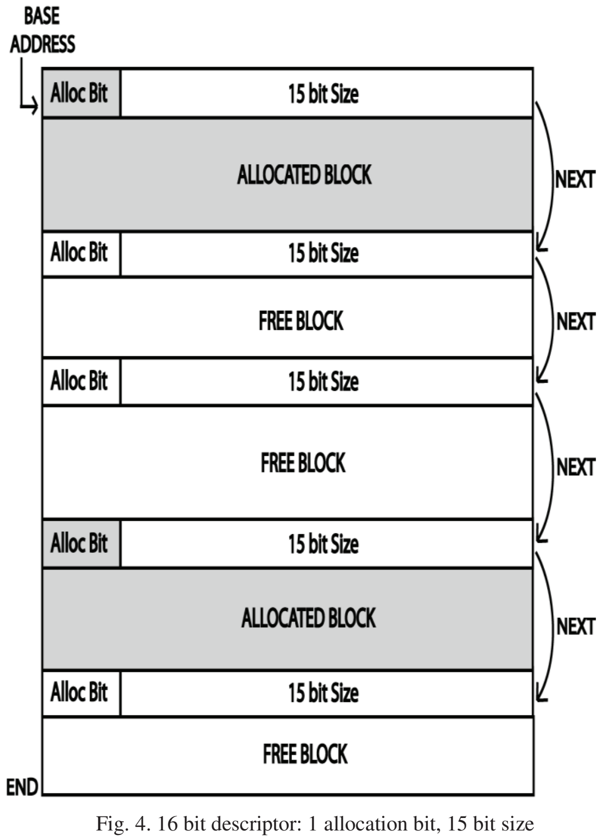

Fig. 4. 16 bit descriptor: 1 allocation bit, 15 bit size

allocation frame size is configurable and can be set as 1, 2, 4 and 8 bytes which can effectively handle a single block of size 32KB, 64KB, 128KB and 256KB respectively. Fig. 4. shows the allocation scheme of ST_MEMMGR.

B. ST_COMPACT_MEMMGR

The ST_COMPACT_MEMMGR utilizes memory compaction in order to eliminate the problem of external fragmentation leading to suboptimal usage of memory. However memory compaction involves reordering of allocated blocks across the memory and thus a mechanism is required to update the allocation references, before any read/write operation is attempted. Contiki handles this problem, by introducing a level of indirection. However in Contiki the programmer is required to get an updated reference each time an attempt is made to access the allocated block. ST_COMPACT_MEMMGR employs a novel scheme to automate the process of updating the allocation references. To automate the process of updating the allocation references, the allocation reference pointers are kept as global variables, and while allocating the reference to the allocation pointer is passed as an argument. Internally the system keeps track of the allocation reference as a 16 bit value which is the relative address of the reference from the base address of the RAM. During every memory compaction process it recalculates the absolute address from the stored 16 bit relative address and the base address and updates the refer-

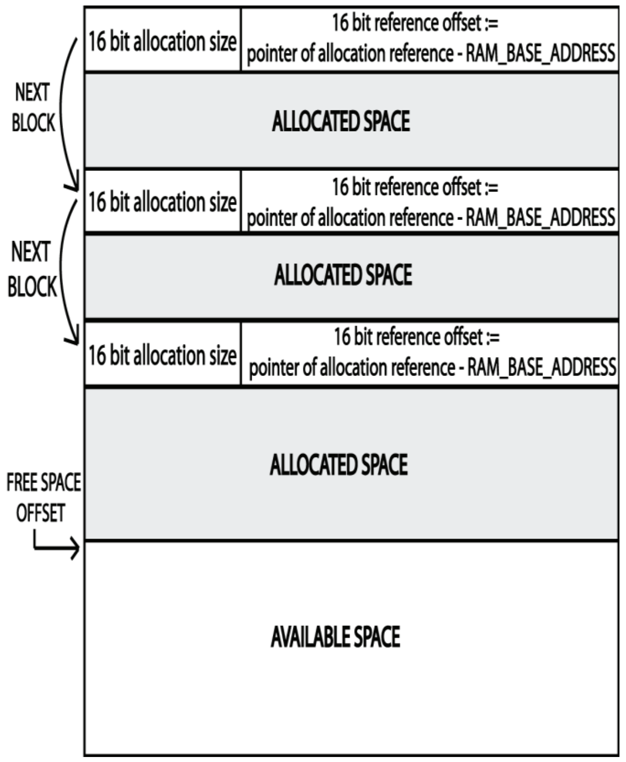
16 bit allocation size
16 bit reference offset := pointer of allocation reference - RAM_BASE_ADDRESS
ALLOCATED SPACE
NEXT BLOCK
16 bit allocation size
16 bit reference offset := pointer of allocation reference - RAM_BASE_ADDRESS
ALLOCATED SPACE
NEXT BLOCK
16 bit allocation size
16 bit reference offset := pointer of allocation reference - RAM_BASE_ADDRESS
ALLOCATED SPACE
FREE SPACE OFFSET
AVAILABLE SPACE

Fig. 5. 32 bit descriptor: 16 bit allocation size, 16 bit allocation reference offset

ences. Table II describes the required modification to be done in the API in order to achieve this.

One limitation of this scheme however is that it has a heavy dependency on the actual amount of RAM available on a particular platform and it will fail to work in a general setting where the available RAM size exceeds 64KB. Due to this apparent problem, special measures were taken while designing the experimental framework to enable the evaluation of this scheme on a general purpose PC.

TABLE II. COMPARISON OF THE TYPE SIGNATURE OF THE STANDARD API WITH THE MODIFIED API FOR ST_COMPACT_MEMMGR

Standard API	**Modified API**
void* malloc(int)	void st_compact_memmgr_alloc(void**,int)
void free(void*)	void st_compact_memmgr_free(void**)

VI. RESULTS & DISCUSSION

The methodology followed for the purpose of experiments is mainly divided in two steps. In the first step allocation and deallocation traces of a set of real applications and a synthetic application were collected using the linux ltrace command. The second step involved performance evaluation of the proposed schemes with respect to some existing allocation schemes.
Three real open source applications were chosen. These applications were treated as black box entities that use dynamic memory allocation. The applications were chosen such that it may find a potential application in the domain of WSN.
AxTLS [18], is an opensource compact ssl implementation. Such a library can find a potential application in implementing a network security protocol like TLS. Huffman Compresssion [19] is a general text compression algorithm and can be used to compress a data payload in order to reduce network bandwidth. SimpleXML [20], is a minimalistic XML parser, and may find application in protocols like SEP2 [21], which uses XML formatted data. Besides these real applications synthetic allocation and de allocation traces were also taken which generates random allocation and de allocation request of varying size, in order to maintain generality.

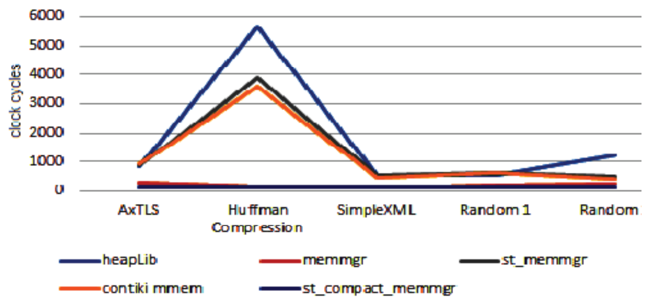

Fig. 6. Comparison of average allocation execution time measured in clock cycles

For the purpose of performance evaluation a PC application was created that reads the allocation and deallocation trace from a file and performs the allocation and deallocation operations in the same order as in the trace file. The dynamic memory manager to be used can be configured along with the size of the fixed pool allocation buffer size. To get the closest possible estimation of the allocation and deallocation execution times the specialized x86 instruction RDTSC (Read timestamp counter) was used. The RDTSC instruction reads a 65 bit counter which increments its value every CPU cycle from the moment the computer is turned on.

A. *Results Based on Average Execution Time*

Fig. 6 and Fig. 7 shows the average allocation and deallocation execution time in CPU cycles respectively. It can be observed that memmgr has the best allocation and deallocation execution times. ST_MEMMGR has a comparable allocation and deallocation execution time with respect to heapLib and Contiki mmem.ST_COMPACT_MEMMGR however shows relatively better performance in terms of both allocation and deallocation time's when compared to Contiki mmem.

B. *Results Based on Cost Metric*

Cost metric is defined as the minimum amount of memory required to successfully fulfill all allocation request of an application [22]. For embedded devices with very limited memory it serves as a very important metric. Fig. 8 describes the relative cost metric of ST_MEMMGR and ST_COMPACT_MEMMGR with respect to heapLib, memmgr and contiki mmem for a set of applications.

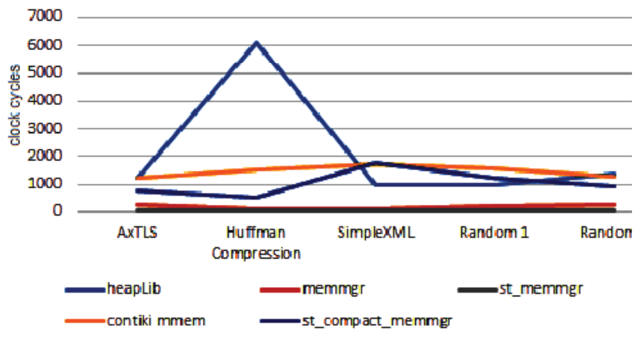

Fig. 7. Comparison of average de allocation execution time measured in clock cycles

From the statistics describes in Figure 6 it is obvious that the memory managers supporting memory compaction i.e. Contiki mmem and ST_COMPACT_MEMMGR has the least cost metric as they don't suffer from any form of memory fragmentation.

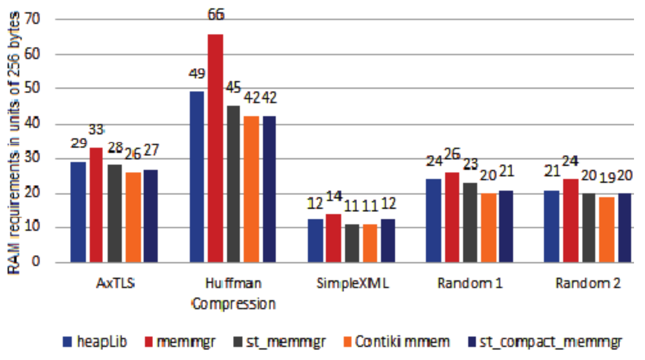

Fig.8. Minimum memory required in multiples of 256 byte units

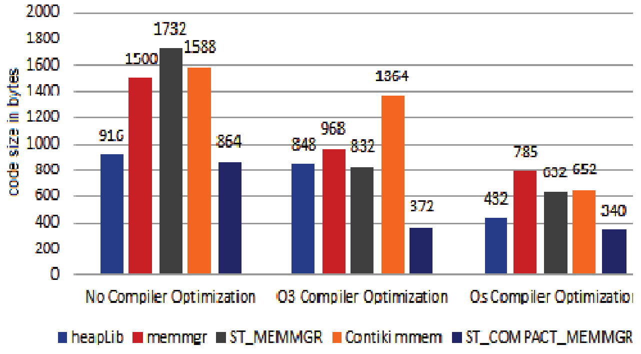

Fig.9. Code size in Flash

Among memory managers that don't support memory compactionST_MEMMGR has the least cost metric and its minimum memory requirements are approximately 256 bytes – 5376 bytes lesser. If compared the following is the order of cost metric for memory managers without memory compaction: memmgr>heapLib> ST_MEMMGR.

Contiki mmem however scores a better cost metric than that of ST_COMPACT_MEMMGR. The specific reason for such an observation is because Contiki mmem does not stores any memory descriptor information in the fixed memory pool. Rather it maintains a linked list of struct mmem outside of the fixed memory pool as shown in Figure 3.

However if the overall memory overhead per allocation is consid-ered it can be seen that each struct mmem is of size 12 bytes, while for ST_COMPACT_MEMMGR the memory overhead per allocation is 4 bytes (Memory Descriptor) + 4 bytes (Pointer serving as allocation handle) = 8 bytes. This accounts for 33% reduction of per allocation memory overhead in ST_COMPACT_MEMMGR over Contiki mmem.

C. *Results Based on Code Footprint*

Fig. 8 shows a comparative program flash space requirements under different compiler optimization configuration. The compiler used for the evaluation of these results is an arm-non-eabi-gcc compiler for Cortex M3 processor. From the statistics in Fig. 9 it can be observed that although ST_MEMMGR has the largest code footprint without any optimization, it considerably reduces with O3 and Os compiler optimization and assumes values which are comparable to others. ST_COMPACT_MEMMGR however consistently shows the least code footprint size.

VII. FUTURE SCOPE

The techniques and methods described in this paper are primarily targeted towards rapid prototyping activities. Limitations in terms of unbounded execution time still make dynamic memory management a poor choice for any mainstream embedded systems development activity. This work can be further explored in terms of assessing the various practical problems and the extent to which it hinders with the normal system functionality when deployed in real field testing.

VIII. CONCLUSION

In this paper the problem of dynamic memory management in constrained embedded systems is revisited. The motivation for this work surfaced during a rapid prototyping development activity of a secure network stack for WSN devices where a viable replacement was required for the dynamic memory management in standard C library. Although various embedded dynamic memory management API are present, they are not compliant with the dynamic memory management of the standard C library. Using one of these non-compliant libraries would have led to refactoring a complex code base, which has a high chance of introducing bugs and also increase the development time. Other opensource embedded dynamic memory managers failed to service all requests within a limited memory.

The paper discusses various problems associated with dynamic memory management in a constrained memory embedded systems. Dynamic memory management schemes present in a few popular sensor network OS like SOS, MantisOS and Contiki are explored and analyzed. It was identified that most of these schemes suffered from a large memory descriptors and thus two new designs ST_MEMMGR and ST_COMPACT _MEMMGR are proposed with reduced size of memory descriptors. Through experimental evaluation it was shown that with reduced size memory descriptors it is possible to have a more optimal use of memory without having significant detrimental effect on performance.

REFERENCES

[1] Knuth, D. E. 1968. Fundamental Algorithms. (The Art of Computer Programming, vol. 1). Addison Wesley.

[2] Puaut, I. 2002. Real-time performance of dynamic memory allocation algorithms. In *proceedings of the 14th Euromicro Conference on Real-Time Systems, 2002.* (pp. 41-49). IEEE.

[3] Lea, D., &Gloger, W. 1996. A memory allocator.

[4] Masmano, M., Ripoll, I., Crespo, A., & Real, J. 2004. TLSF: A new dynamic memory allocator for real-time systems. In *proceedings of the 16th Euromicro Conference on Real-Time Systems, 2004* (pp. 79-88). IEEE.

[5] Ramakrishna, M., Kim, J., Lee, W., & Chung, Y. 2008. Smart dynamic memory allocator for embedded systems. In *proceedings of the 23rd International Symposium on* Computer and Information Sciences, 2008 (pp. 1-6). IEEE.

[6] VO, K. P. 1996. Vmalloc: A general and efficient memory allocator. *Software: Practice and Experience*, *26*(3), 357-374.

[7] Johnstone, M. S., & Wilson, P. R. 1998. The memory frag-mentation problem: solved? In *ACM SIGPLAN Notices* (Vol. 34, No. 3, pp. 26-36).

[8] Alliance, Z. 2010. ZigBee IP Specification. *ZigBee 095023r10, Work in Progress, July*.

[9] Dunkels, A., Finne, N., Eriksson, J., & Voigt, T. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (pp. 15-28). ACM.

[10] Han, C. C., Kumar, R., Shea, R., Kohler, E., & Srivastava, M. 2005. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mo-bile systems, applications, and services* (pp. 163-176). ACM.

[11] Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., & Yoon, C. 2007. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *6th Inter-national Symposium on Information Processing in Sensor Networks, 2007.* (pp. 148-157). IEEE.

[12] Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., & Han, R. 2005. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. In *Mobile Networks and Applications*, 2005. *10*(4), 563-579. Springer.

[13] Dunkels, A., Gronvall, B., & Voigt, T. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks, 2004.* (pp. 455-462). IEEE.

[14] P. Pariani. 2012. heapLib 1.1. http://sourceforge.net/projects/ heaplib/

[15] E. Bendersky. 2008. Memmgr http://eli.thegreenplace.net/2008/10/17/memmgra-fixed-pool-memory-allocator

[16] Min, H., Yi, S., Cho, Y., & Hong, J. (2007, March). An efficient dynamic memory allocator for sensor operating systemsIn *Proceedings of the 2007 symposium on applied co-mputing*(pp. 1159-1164). ACM.

[17] A. Dunkels, "The mmem managed memory allocator,"https://github.com/adamdunkels/contiki-fork/wiki/Mem-ory-Allocation

[18] Forsberg, Dan, et al. "Protocol for carrying authentication for network access (PANA)." http://www.ietf.org/rfc/rfc5191. txt (2008).

[19] Aboba, Bernard, et al. Extensible authentication protocol (EAP). RFC 3748, June, 2004.

[20] Simon, Dan, Bernard Aboba, and Ryan Hurst. "The EAP-
TLS authentication protocol." RFC5216, IETF, March (2008).

[21] Cameron Rich. 2011. axTLS embedded SSL. http://axtls.sourceforge.net/index.html

[22] ShaileshGhimiray. 2011. huffman-textfile-compression. https://code.google.com/p/huffman-textfile-compression/

[23] Bruno Essmann. 2002. Simple XML 1.0. http://simplexml.sourceforge.net/index.html

[24] Alliance, Z., & Alliance, H. P. 2010. Smart Energy Profile
2.0 Technical Requirements Document

[25] Del Rosso, C. 2006. The method, the tools and rationales for assessing dynamic memory efficiency in embedded real-time systems in practice. In *International Conference on Software Engineering Advances, 2006* (pp. 56-56) IEEE.