Fast Constant Time Memory Allocator for Inter Task Communication in Ultra Low Energy Embedded Systems

Gregor Rebel* (*Author*), Francisco J. Estevez*, Ingo Schulz**, Peter Glösekötter*

* Dept. of Electrical Engineering and Computer Science, University of Applied Sciences Munster, Germany
** Dept. of Computer Science Chair IV, TU Dortmund University, Germany

***Abstract*—Modern microcontrollers provide enough processing power to benefit from the advantages of multitasking schedulers or operating systems even in the area of small, battery based or energy self-sustaining devices. Many of these devices communicate with other devices via different interfaces. For a multitasking operating system, communication means to collect individual bytes in memory blocks and to transport these blocks between tasks. This paper describes how to use a combination of memory pools and memory headers to provide a fast, constant time memory allocator with low internal fragmentation. The proposed memory allocator is fast enough and has so few internal fragmentation, that it is applicable even in ultra low energy embedded systems with few kilobytes of ram. It can provide memory blocks of equal size at high frequency.**

Keywords—Dynamic, Memory, Allocator, Software, Embedded, Low Power

I. Introduction

A. Ultra Low Energy Embedded Realtime Systems (ULERS)

Today's embedded systems range from smart, self-sustaining sensors with own data pre-processing up to line powered, number crunching multi-core architectures. This paper focuses mainly on software optimization for embedded systems at the low energy end of this spectrum. These devices typically offer one processing core and a single level memory interface. Fore these devices, the amount of available RAM typically ranges from 128 bytes (E.g. MSP430G2x31) to 66 KB (E.g. MSP430F5xx). Most dynamic memory allocators are optimized to manage much larger amounts of ram. This is why many software developers, in this field, implement their own memory management instead of using an external dynamic memory allocator.

B. Software Techniques of Data Transport

Whenever a microcontroller communicates, data has to be transported between a low-level interface driver and the main application. In multitasking operating systems, data also has to be transported between different tasks or threads. Two main types of software data transport are known: Transport by value or transport by reference. Transport by value means to copy all bytes of transported data between each adjacent layers of the software stack. This transport mechanism slows down with each additional software layer. This disadvantage can be overcome if data is transported by reference. Once the low-level driver has received a data packet, only a packet reference is passed to higher layers.

C. Conventional Dynamic Memory Allocators (CDMA)

In the context of this paper, a conventional dynamic memory allocator is a software scheme that allows to dynamically manage a fixed amount of memory at runtime. An alloc() operation provides reference to a memory block of requested size. The inverse operation free() returns the memory block for later reallocation to the memory allocator.

Many algorithms and implementations have been demonstrated for CDMAs that are optimized for different hardware architectures. Some examples of CDMAs especially for embedded systems are described in [1], [2] and [3].

Operating systems for desktop computers are mostly optimized for maximum data throughput. For these systems, the average case runtime of alloc()- and free()-operations should be minimized. This optimization typically increases memory usage or worst case runtime. On the contrast, operating systems for realtime systems often have to fulfill hard time constrains. These systems require low worst case runtimes for alloc() and free(). When it comes to ULERS, both worst-case and average-case runtime plus memory overhead should be as small as possible. In practice, the requirements of such systems often forbid the use of a CDMA.

According to [1], most of the CDMAs described in literature cannot be used in embedded systems. Either the runtime of alloc() or free() are unbounded or their memory overhead is not acceptable. The *Smart CDMA* described in [1] claims to fulfill the requirements of small embedded systems. In fact every CDMA requires its own data structures in memory. The occupied memory cannot be used by the application. This overhead is called internal fragmentation. The initialization and use of these structures increases runtime.

The biggest problem of all CDMAs is that their block management algorithms are heuristics. For every heuristic, a scenario can be constructed in which the optimization fails . In the case of CDMAs this means an increase of external fragmentation (non allocatable bytes of memory due to small holes between allocated blocks) and increased runtime for alloc() and free(). Some algorithms state O(1) runtime requirement for their operations. For small embedded systems, not only the asymptotic but also the effective runtime is important. In practice, even function calls can have a measurable impact on overall performance.

According to [1], the main aspects of a CDMA are response time, fragmentation, cache pollution, mutual exclusion and synchronization. The aspects handled in this paper are response time and fragmentation. The microcontrollers in the intended area of applications typically provide very simple architectures when compared to desktop computers. Multi level caches and multi core devices are not typical for ULERS.

D. Main Aspects of the proposed Algorithm

The algorithm proposed in this paper is not a complete CDMA in the sense of the cited sources. It is an extension that can base on any CDMA. The lowest management overhead can be achieved by using a minimal DMA that can only allocate memory. The main idea is to use memory pools of equal sized memory blocks and memory headers to provide a general, flexible, fast, robust and realtime memory allocator with very little internal and without external fragmentation. Internal fragmentation is very little when compared to CDMAs. Allocate() and free() operations can take place in small, constant time. The algorithms have been designed for use in multi- and single-tasking environments. References to allocated memory blocks can be freely passed around in an application. The Algorithm is optimized for small sized memory blocks of a few 100 bytes but can be used for larger block sizes too.

E. Simple Dynamic Memory Allocator (SimpleDMA)

The implementation of proposed memory allocator can be based on any CDMA. The most simple DMA (SimpleDMA) can only allocate memory. This requires

1. A static array of individual bytes Heap[HEAP_SIZE]
2. A byte pointer NextFree at start points to Heap[0]
3. A function void* alloc(unsigned int Size) That checks if enough space remains in Heap[] for required amount of bytes. If enough space is available, the function increases NextFree by Size and returns its old previous value. Otherwise, NULL is returned.

The further description is based on SimpleDMA. SimpleDMA can be easily replaced by any other CDMA implementation.

II. Design of a Fast Constant Time Memory Allocator based on memory pools and headers

A. Main Components of FCTMA

1. Operation **allocBuffer()** returns pointer P to allocated memory. P can be directly casted for use as any structure. Prototype: struct memory_s* allocBuffer(Bytes)
2. Operation **freeBuffer()** pushes buffer at front of list of unused memory blocks in corresponding memory pool. Prototype: void freeBuffer(struct memory_s* Buffer)
3. Operation **createPool()** allocates and intializes a new pool_s structure. If MaxBlocks is given, all required blocks can be allocated immediately to ensure fast constant response times for all further memory operations. Prototype: struct pool_s* createPool(unsigned int BlockSize, unsigned char MaxBlocks)
4. A data structure **struct memory_s** at address P-1 of each memory block stores at least two entries.
   1. struct memory_s* Next
      Memory blocks can form a single linked list.
   2. struct pool_s* Pool
      Points to the memory pool that manages this block
5. A data structure **struct pool_s** manages all memory blocks of same size by storing four entries.
   1. mutex_t PoolLock
      A mutual exclusion lock protects concurrent accesses to the pool.
   2. semaphore_t BlocksAvailable
      A semaphore initialized by createPool() with the value of MaxBlocks. AllocBuffer() takes 1 from this semaphore before obtaining the mutex PoolLock. freeBuffer() gives 1 to it after releasing PoolLock.
   3. unsigned int BlockSize
      All memory blocks of same pool have same size.
   4. struct memory_s* FirstFree
      Points to start of a single linked list of unused memory blocks in this pool.

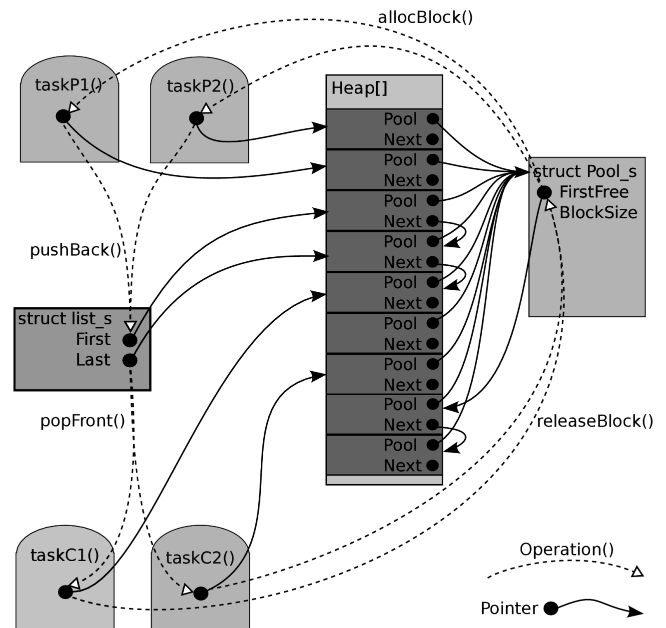

Fig. 1: Dataflow and Memory Structures

III. Performance Comparison

A. Performance Metric

The response times of a CDMA are further defined as the minimum (Tmin), maximum (Tmax) and average (Tavg) runtimes of alloc() and free() operations. The amount of internal fragmentation is measured as ratio of allocated user bytes and total bytes. User bytes means the amount of bytes requested by user application and total bytes being the total amount of bytes allocated from heap plus static allocations like global and local static variables in RAM required by the individual implementation.

The described algorithm has been implemented and tested in an artificial test application. The multitasking scheduler used was *FreeRTOS* [4]. This scheduler is available as open source and has been already described by other papers like [5].

Fig. 1: Dataflow and Memory Structures shows the data flow operations between m Producer tasks taskP1(), ... taskPm() and n Consumer tasks taskC1(), ... taskCn(). It also shows the pointers that build up the complex data structure. At start, the memory allocator under test is initialized. The list is initialized too. Then 10 producer and 10 consumer tasks are spawned. Each producer task allocates twenty memory blocks of 127 bytes (the maximum size of an IEEE 802.15.4 data packet). The task issues pushBack() operations on the list to send all memory blocks to consumer tasks. After push back and if the total amount of allocated memory has reach 16 KB, the producer task sleeps for some random milliseconds. Each consumer task issues a pop_front() operation on the list to retrieve up to ten memory blocks. The retrieved memory blocks are given back to the pool by free() operations. The consumer task then waits for a pseudo random time.

A simple pseudo random number generator is used to calculate waiting times. Fig. 2 shows its implementation.

```
static unsigned long sr_Seed = 1;
sr_Seed = sr_Seed * 1103515245 + 12345;
WaitingTime = ((sr_Seed>>16)) % MaxTime;
```

Fig. 2: Pseudo Random Number Generator

The random waiting time simulates a random processing time to produce and consume each memory block. Fine tuning MaxTime value of producers and consumers lets converge the total amount of allocated buffers around 100 blocks after some minutes.

Comparing different implementations needs exact measure of runtime for each alloc() and free() operation. This is achieved by inserting extra probe-commands at begin and end of these functions to set and clear an individual GPIO pin. Pin ALLOC is set at begin of alloc() and cleared at its end. Pin FREE is controlled by free(). The extra probe-commands add constant runtime of $<1\mu s$ to each operation. The minimum, average and maximum runtime can then be measured by use of a digital oscilloscope.

Multitasking and interrupt requests are disabled during alloc() and free() operations for more precise measures. Task synchronization, task switching and interrupt service routines add extra uncertainty to the runtime of any implemented algorithm and would blur runtime measures.

B. Candidates in Comparison

1) Algorithm FCTMA

The algorithm described above has been implemented in a self developed software toolchain "The ToolChain" [6] for STM32 based microcontrollers. The ToolChain makes use of the *FreeRTOS* [4] multitasking scheduler as a SimpleDMA. Task synchronization is provided by special, 32-bit optimized mutexes, semaphores, queues and lists. FCTMA is used for memory pools in The ToolChain beginning at revision 1.0.52.

2) FreeRTOS Heap4

The *FreeRTOS* multitasking scheduler provides different implementations of DMAs. The most advanced DMA implementation is called *Heap4*. It consolidates adjacent memory blocks as they are freed to reduce memory fragmentation. The benchmark was run against *FreeRTOS* revision 7.3.0. The extra probe-commands have been inserted directly into pvPortMalloc() and vPortFree() inside the vTaskSuspendAll() block.

3) Algorithm SDMA

The *Smart Dynamic Memory Allocator* described in [2] has been implemented by use of The ToolChain [6]. At initialization, a chunk of 32 KB is allocated using *FreeRTOS Heap4*. All further memory allocations are then served by the SDMA implementation from this memory chunk. The amount of free list classes has been reduced to 15 to avoid wasting memory. 15 classes are enough to represent memory allocation of up to 32 KB. The lookup tables LTB1[] and LTB2[] have been implemented as constant char arrays which places them into flash memory and saves an additional 512 bytes of ram. The BlkPredMask has been initialized as 255 to set the first eight classes as short lived. This allows to treat all blocks allocated by the benchmark as short lived and avoids block split and merge operations. The pseudocode implementation of second-level-index calculation was not used as it seemed to be incomplete. Instead an implementation based on bit-shifting was used according to the textual description.

C. Test Setup

The software implementation was tested on a CortexM3 based STM32F103ZET6 microcontroller manufactured by ST Microelectronics Corp. The chip provides a 32-bit architecture with a uniform 4 GB address space. This uniform address space allows to create C-pointers to every memory region or register. The microcontroller is equipped with 64 KB of RAM and 512 KB of FLASH memory. It is clocked at 72 MHz. When compared to older 8- and 16-bit microcontrollers, the CortexM3 provides more computing power, a linear address space and more ram. This allows to run more tasks to stress test the implementation than on smaller architectures. Despite of its computing power, the chosen microcontroller is still a better representation of a ULERS than a desktop computer.

Runtime analysis is done by attaching a Tektronix MSO4104 Digital Oscilloscope to Pins ALLOC and FREE. Its signal analyzer allows to measure minimum, maximum and average high-time of signals on both pins. Measures have been taken after 1 minute of continuos simulation. All sources have been compiled by gcc (GNU for ARM Tools) v4.7.1 without any optimizations and with enabled debugging support.

D. Performance results

TABLE I. DURATION OF ALLOC() OPERATION FOR DIFFERENT MEMORY ALLOCATOR IMPLEMENTATIONS

Algorithm	T_{avg}	T_{min}	T_{max}	**Std Deviation**	**1000 * Std Deviation / $T_{average}$**
FCTMA	2196ns	2174ns	2211ns	14ns	6.38 ‰
FreeRTOS Heap4	3300ns	2706ns	7111ns	1070ns	324.24 ‰
SDMA	171600ns	171600	171600	13.5ns	0.08 ‰

TABLE II. DURATION OF FREE() OPERATION FOR DIFFERENT MEMORY ALLOCATOR IMPLEMENTATIONS

Algorithm	T_{avg}	T_{min}	T_{max}	**Std Deviation**	**1000 * Std Deviation / $T_{average}$**
FCTMA	957ns	955ns	958ns	666.3ps	0.7 ‰
FreeRTOS Heap4	2847ns	1943ns	6874ns	1095ns	384.62 ‰
SDMA	165800ns	165800ns	166100ns	26ns	0.1 ‰

TABLE III. INTERNAL FRAGMENTATION FOR DIFFERENT MEMORY ALLOCATOR IMPLEMENTATIONS

Algorithm	**User Bytes allocated**	**Total Bytes allocated**	**Internal Fragmentation (%)**
FCTMA	20558 (21198)	21878	6.03 % (3.11 %)
FreeRTOS Heap4	26114	26939	3.06 %
SDMA	13152	15936	17.47 %

Table 1 shows average, minimum and maximum runtime measures of different memory allocator implementations. It also shows the standard deviation as absolute and per mill value. The FCTMA implementation is in average 50% faster than *FreeRTOS Heap4*. Still *Heap4* provides a very fast alloc() operation. But the standard deviation of *Heap4* is more than 50 times bigger than FCTMA. This effect is caused by the block splitting algorithm of *Heap4*. The SDMA algorithm is 78 times slower than FCTMA. This is due to large computational overhead in calculation of first- and second level index values for each operation. The very low standard deviation of SDMA is caused by the basic implementation done for this comparison. Block splitting and merging have been disabled for this test.

Table 2 basically shows a similar picture as Table 1. The FCTMA free() operation is nearly 3 times faster than *Heap4* because of its minimal management overhead. Its standard deviation is practically zero. The *FreeRTOS Heap4* implementation merges adjacent blocks which explains its high standard deviation similar to it alloc() performance. The SDMA implementation takes nearly as long for free() as for alloc() because of its disabled block merging. SDMA is now 173 times slower than FCTMA.

Table 3 compares memory usage and internal fragmentation of the three implementations. FCTMA shows 6.03 % of internal fragmentation. This value results from 32 bytes for each memory pool and 8 bytes for each memory header. If the application makes use of the next-pointer in each memory block, then we can add 4 bytes for each memory header to User Bytes and achieve an effective internal fragmentation of 3.11% (numbers in brackets). This benchmark clearly shows, that the SDMA algorithm is optimized to manage larger amount of memory. [2] was benchmarked on a Pentium D 3.4Ghz with 1 GB RAM with up to 4798 MB of allocated memory. But we must also be fair to say that internal fragmentation of SDMA would benefit from block merge like *Heap4* does.

IV. CONCLUSION & OUTLOOK

This paper has compared three different implementations of memory allocators in an artificial, inter task communication benchmark on an up to date microcontroller. The introduced FCTMA algorithm has proven that it can provide dynamic memory of constant size in near constant time and faster than any CDMA implementation. The internal fragmentation of FCTMA is equal or comparable to its competitors. The algorithm makes software development of embedded systems easier and safer by replacing self made memory managers. FCTMA can be used on top of any CDMA and is easy to implement. It is ideally suited to provide memory blocks of equal size in high frequency.

Future research may concentrate on extending FCTMA with support for interrupt service routines. In contrast to a task, an interrupt service routine cannot wait for other tasks to unlock a mutex to gain access to a memory pool.

ACKNOWLEDGEMENT

I wish to thank Oliver Wesch for proof reading this paper.

REFERENCES

[1] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: a New Dynamic Memory Allocator for Real-Time Systems, 2004

[2] Ramakrishna M, Jisung Kim, Woohyong Lee and Youngki Chung. Smart Dynamic Memory Allocator for Embedded Systems, 2008

[3] *David A. Barret, Benjamin Zom. Using Lifetime Predictors to improve Memory Allocation Performance, 1993*

[4] *R. Barry. FreeRTOS. http://www.freertos.org/, 2013*

[5] *Jan Tobias Muhlberg, Leo Freitas.Verifying FreeRTOS: from requirements to binary code. Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AvoCS 2011), 2011*

[6] *Gregor Rebel. The ToolChain – An open source software toolchain for CortexM3 microcontrollers. http://thetoolchain.com, 2013*