DYNAMIC MEMORY MANAGEMENT FOR EMBEDDED REAL-TIME SYSTEMS

A. Crespo, I. Ripoll, M. Masmano
Universidad Politecnica de Valencia
46022 Valencia, Spain
{acrespo,iripoll,mmasmano}@disca.upv.es

Abstract Dynamic memory storage has been widely used during years in computer science. However, its use in real-time systems has not been considered as an important issue because the spatial and temporal worst case for allocation and deallocation operations were unbounded or bounded but with a very large bound.

TLSF (Two Level Segregated Fit) is a new allocator has been designed specifically to meet real-time constraints. These constraints are addressed in two axis: Time and Space. While the temporal behaviour of TLSF is excellent, $O(1)$, the spatial behaviour is as the best of the known allocators. The spatial constraint is specially required in embedded systems with mitided resources. An efficient and guaranteed use of memory is needed for these systems. In this paper we compare the temporal and spatial performances of the TLSF allocator comparing it with the most relevant allocators.

Keywords: Dynamic memory allocation, real-time systems, embedded systems

1. INTRODUCTION

Dynamic storage allocation (DSA) algorithms plays an important role in modern software engineering paradigms (object oriented paradigm) and techniques. Additionally, it allows to increase the flexibility and functionalities of the applications. In fact, there exist in the literature a large number of works and references to this particular issue. However, in the real-time community the use of dynamic memory techniques has not been considered as an important issue because the spatial and temporal worst case for allocation and deallocation operations were insufficiently bounded. It is significant the reduced number of papers about this topic in most relevant real-time events.

Nevertheless, it is not wise to ignore the use of dynamic memory in real-time applications just because it is believed that it is not possible to design an allocator that matches the specific needs of a real-time system. To take profit of, these advantages for the developer of real-time systems a deterministic response of

Please use the following format when citing this chapter:

Crespo, A., Ripoll, I., Masamano, M., 2006, in IFIP International Federation for Information Processing, Volume 225, From Model-Driven Design to Resource Management for Distributed Embedded Systems, eds. B. Kleinjohann, Kleinjohann L., Machado R., Pereira C., Thiagarajan P.S., (Boston: Springer), pp. 195–204.

the dynamic memory management is required. This requirement implies the use of dynamic memory mechanisms completely predictable. Allocation and deallocation are the basic mechanisms to handle this issue. Additionally to the temporal cost of these operations is the memory fragmentation incurred by the system when dynamic memory is used. Fragmentation plays an important role in the system efficiency which is more relevant for embedded systems.

TLSF [4] is a new allocator has been designed specifically to meet real-time constraints. These constraints are addressed in two axis: Time and Space. While the temporal behaviour of TLSF is excellent, $O(1)$, the spatial behaviour is as the best of the known allocators. In this paper we compare the temporal and spatial performances of the TLSF allocator comparing it with the most relevant allocators.

1.1 DSA AND REAL-TIME REQUIREMENTS

The requirements of real-time applications regarding dynamic memory can be summarised as:

- **Bounded response time.** The worst-case execution time (WCET) of memory allocation and deallocation has got to be known in advance and be independent of application data. This is the main requirement that must be met.
- **Fast response time.** Besides, having a bounded response time, the response time has got to be short for the DSA algorithm to be usable. A bounded DSA algorithm that is 10 times slower than a conventional one is not useful.
- **Memory requests need to be always satisfied.** No real-time applications can receive a null pointer or just be killed by the OS when the system runs out of memory. Although it is obvious that it is not possible to always grant all the memory requested, the DSA algorithm has got to minimise the chances of exhausting the memory pool by minimising the amount of fragmentation and wasted memory.
- **Efficient use of memory.** The allocator has got to manage memory efficiently, that is, the amount of wasted memory should be as small as possible.

2. DYNAMIC STORAGE ALLOCATION ALGORITHMS

This section presents a categorisation of existing allocators and a brief description of the most representative ones, based on the work of Wilson et al. [13] which provides a complete taxonomy of allocators.

Considering the main mechanism used by an allocator, the following categorisation is proposed [13]. Examples of each category are given. In some cases it is difficult to assign an allocator to a category because it uses more than one mechanism. In that case, tried to determine which is the more relevant mechanism and categorise the allocator accordingly.

Sequential Fits: Sequential Fits algorithms are the most basic mechanisms. They search sequentially free blocks stored in a singly or doubly linked list. Examples are first-fit, next-fit, and best-fit. First-fit and best-fit are two of the most representative sequential fit allocators, both of the are usually implemented with a doubly linked list.

Segregated Free Lists: These algorithms use a set of free lists. Each of these lists store free blocks of a particular predefined size or size range. When a free block is released, it is inserted into the list which corresponds to its size. There are two of these mechanisms: Simple Segregated storage and Segregated

Buddy Systems: Buddy Systems [2] are a particular case of Segregated free lists. Being $\mathcal{H}$ the heap size, there are only $\log_2(\mathcal{H})$ lists since the heap can only be split in powers of two. This restriction yields efficient splitting and merging operations, but it also causes a high memory fragmentation. The Binary-buddy [2] allocator is the most representative of the Buddy Systems allocators, which besides has always been considered as a real-time allocator.

Indexed Fits: This mechanism is based on the use of advanced data structures to index the free blocks using several relevant features. To mention a few examples: algorithms which use Adelson-Velskii and Landin (AVL) trees [10], binary search trees or cartesian trees (Fast-Fit [12]) to store free blocks.

Bitmap Fits: Algorithms in this category use a bitmap to find free blocks rapidly without having to perform an exhaustive search. TLSF and Half-fit [6] are examples of this sort of algorithms. Half-fit groups free blocks in the range $[2^i, 2^{i+1}[$ in a list indexed by i. Bitmaps to keep track of empty lists jointly with bitmap processor instructions are used to speed-up search operations.

Hybrid allocators: Hybrid allocators can use different mechanisms to improve certain characteristics (response time, fragmentation, etc.) The most representative is Doug Lea's allocator [3], which is a combination of several mechanisms. In what follows this allocator will be referred to as DLmalloc.

Table 1 summarises the temporal costs of the worst-case allocation and deallocation of these algorithms

Table 1. Worst-case costs

	Allocation	Deallocation
First-fit/Best-fit	$O\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$	$O(1)$
Binary-buddy	$O(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$	$O(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$
AVL-tree	$O(1.44\ \log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$	$O(3 \cdot 1.44\ \log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$
DLmalloc	$O\left(\frac{\mathcal{H}}{\mathcal{M}}\right)$	$O(1)$
Half-fit	$O(1)$	$O(1)$
TLSF	$O(1)$	$O(1)$

Although the notion of fragmentation seems to be well understood, it is hard to define a single method of measuring or even defining what fragmentation is. In [13] fragmentation is defined as "the inability to reuse memory that is free". Historically, two different sources of fragmentation have been considered: internal and external. Internal fragmentation is caused when the allocator returns to the application a block that is bigger than the one requested (due to block round-up, memory alignment, unablility to handle the remaining memory, etc.). External fragmentation occurs when there is enough free memory but there is not a single block large enough to fulfill the request. Internal fragmentation is caused only by the allocator implementation, while external fragmentation is caused by a combination of the allocation policy and the user request sequence.

Attending to the reason an allocator can not use some parts of the memory, three types of wasted memory, usually called fragmentation can be defined: it Internal fragmentation when the allocator restricts the possible sizes of allocatable blocks because of design constraints or efficiency requirements; *External fragmentation:* The sequence of allocation and deallocation operations may leave some memory areas unused since; and, *wasted memory* which corresponds to the memory managed by the allocator that can not be assigned to the application . In fact, the wasted memory at any time is the difference between the live memory and the memory used by the allocator at that time.

The following table summarises the worst-case memory requirements for several well known allocation policies (see [7–9, 2]).

The table shows the heap size needed when the maximum allocated memory (live memory) is $\mathcal{M}$ and the largest allocated block is m, Robson also showed that an upper bound for the worst-case of any allocator is given by: $\mathcal{M} \times m$.

The other allocation algorithms will have a fragmentation depending on the implemented policy. While AVL-tree, DLmalloc and TLSF tends to a good fit policy, Half-fit implements a binary buddy policy.

Table 2. Fragmentation worst-case

	Heap size
First-fit	$\frac{\mathcal{M}}{\ln 2}\sum_{i=1}^{m}(\frac{1}{i})$
Best-fit	$\mathcal{M}(m-2)$
Binary-buddy	$\mathcal{M}(1+\log_2 m)$

However, several studies [11, 5] based on synthetic workload generated by using well-known distributions (exponential, hyper-exponential, uniform, etc.). The results obtained were not conclusive; these studies show contradictory results with slightly different workload parameters. At that time, it was not clear whether First-fit was better than Best-fit.

Johnstone and Wilson [1] analysed the fragmentation produced by several standard allocators, and concluded that the fragmentation problem is a problem of "poor" allocator implementations rather than an intrinsic characteristic of the allocation problem itself. A

The policy used to search for a suitable free blocks in Half-fit and TLSF introduces a new type of fragmentation or *incomplete memory use* as it is called in [6]: free blocks larger than the base size of the segregated list where they are located, will not be used to serve requests of sizes that are one byte larger than the base size.

TLSF is a bounded-time, Good-fit allocator. The Good-fit policy tries to achieve the same results as Best-fit (which is known to cause a low fragmentation in practice [1]), but introduces implementation optimisations so that it may not find the tightest block, but a block that is close to it. TLSF implements a combination of the Segregated and Bitmap fits mechanisms. The use of bitmaps allow to implement fast, bounded-time mapping and searching functions.

3. TLSF

In a previous paper [4], the authors described a preliminary version of a new algorithm for dynamic memory allocation called TLSF (Two-Level Segregated Fit). We now present a slightly improved version of the algorithm that maintains the original worst-case time bound and efficiency, and reduces the fragmentation problem in order to make TLSF usable for long-lived real-time systems.

TLSF is a bounded-time, Good-fit allocator. The Good-fit policy tries to achieve the same results as Best-fit (which is known to cause a low fragmentation in practice [1]), but introduces implementation optimisations so that it may not find the tightest block, but a block that is close to it. TLSF implements a combination of the Segregated and Bitmap fits mechanisms. The use of

bitmaps allow to implement fast, bounded-time mapping and searching functions.

The TLSF data structure can be represented as a two-dimension array. The first dimension splits free blocks in size-ranges a power of two apart from each other, so that first-level index i refers to free blocks of sizes in the range $[2^i,2^{i+1}[$. The second dimension splits each first-level range linearly in a number of ranges of an equal width. The number of such ranges, $2^{\mathcal{L}}$, should not exceed the number of bits of the underlying architecture, so that a one-word bitmap can represent the availability of free blocks in all the ranges. According to experience, the recommended values for $\mathcal{L}$ are 4 or, at most, 5 for a 32-bit processor. Figure 1 outlines the data structure for $\mathcal{L} = 3$.

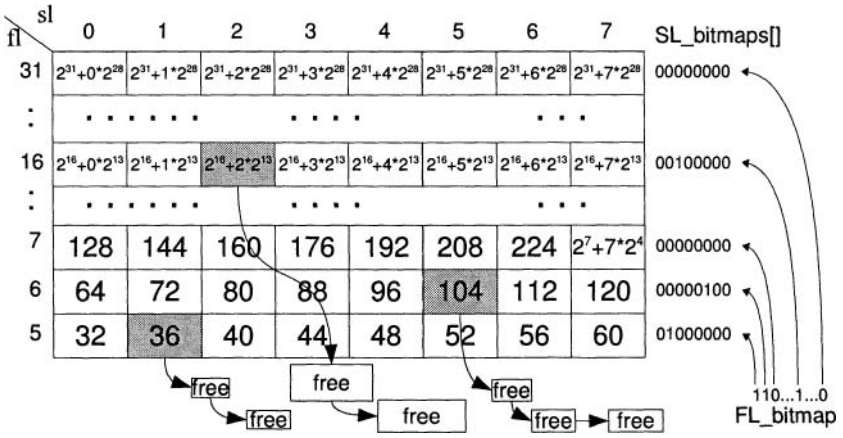

Figure 1. TLSF data structures example.

TLSF uses word-size bitmaps and processor bit instructions to find a suitable list in constant time. For example, using the *ffs* instruction, which returns the position of the first (least significat) bit set to 1, it is possible to find the smaller non-empty list that holds blocks bigger or equal than a given size; and the instruction *fls* (returns the position of the most significant bit set to 1) can be used to compute the $\lfloor \log_2(x) \rfloor$ function. Note that it is not mandatory to have these advanced bit operations implemented in the processor to achieve constant time, since it is possible to implement them by software using less than 6 non-nested conditional blocks (see glibc or Linux implementation).

Given a block of size $r > 0$, the first and second indexes (fl and sl) of the list that holds blocks of its size range are: $fl = \lfloor \log_2(r) \rfloor$ and $sl = \lfloor (r - 2^{fl})/2^{fl-\mathcal{L}} \rfloor$. For efficiency reasons, the actual function used to cal-

cualte sl is $(r/s^{fl-\mathcal{L}}) - 2^{\mathcal{L}}$. The function mapping_insert computes efficiently fl and sl:

4. TEMPORAL ANALYSIS

In this analysis we compare the performance of TLSF with respect to the worst-case scenario. To evaluate the temporal cost we have analysed the worst-case scenario for each allocator and we have measured the cost of all of them under it. A description of the worst-case scenarios of each allocator can be found in [4].

The memory heap size was set to 4Mbytes, and the minimum block size is 16bytes. All testing code is available at http://rtportal.upv.es/rtmalloc/

Table 3. Worst-case (WC) and Bad-case (BC) allocation

Malloc	FF	BF	BB	DL	AVL	HF	TLSF
Processor instructions	81995	98385	1403	721108	3116	164	197
Number of cycles	1613263	1587552	3898	3313253	11739	1690	2448

Table 3 show measurements of a single malloc operation after the worst-case scenario has been constructed. Every allocator has been tested for each worst-case scenario. The result of an allocator when tested in its worst-case scenario is printed in bold face in the tables. The results show that each allocator performs badly in its theoretical worst-case scenarios. This can be easily seen in table 3(b) (instruction count).

As expected, First-fit and Best-fit perform quite badly under their worst-case scenarios. The low data locality produces a high cache miss ratio which makes the temporal response even worse than expected, considering the number of instructions executed. The number of cycles per instruction (CPI) is high: 19 for First-fit and 16 for Best-fit.

The DLmalloc allocator is a good example of an algorithm designed to optimise the average response time, but it has a very long response time in some cases. DLmalloc tries to reduce the time spent coalescing blocks by delaying coalescing as long as possible; and when more space is needed coalescing is done all at once, causing a large overhead on that single request. DLmalloc has the largest response time of all allocators, and therefore it is not advisable to use in real-time systems.

Half-fit, TLSF, Binary-buddy and AVL show a reasonably low allocation cost, Half-fit and TLSF being the ones which show the most uniform response time, both in number of instructions and time. Half-fit shows the best worst-case response; only TLSF is 20% slower. Although Half-fit shows a fast, bounded response time under all tests, it suffers from a considerable theoretical internal fragmentation.

As explained in the fragmentation section, Half-fit provides a very good worst-case response time at the expense of wasting memory. Half-fit is superior in all respects to Binary-buddy. Half-fit is faster than Binary-buddy and handles memory more efficiently.

5. FRAGMENTATION EVALUATION

In order to analyse the fragmentation incurred by the allocators, we have defined a periodic task model which includes the memory requirement of each task. Using this task model, the allocators have been evaluated under different situations.

Let $\tau = \{T_1, ..., T_n\}$ be a periodic task system. Each task $T_i \in \tau$ has the following temporal parameters $T_i = (c_i, p_i, d_i, g_i, h_i)$. Where c_i is the worst execution time, p_i is the period; d_i is the deadline, g_i is the maximum amount of memory that a task T_i can request per period; and h_i is the longest time than task can hold a block after its allocation (holding time).

In this model a task T_i can ask for a maximum of g_i bytes per period which have to be released no later than h_i units of time.

To serve all memory requests, the system provides an area of free memory (also known as heap) of $\mathcal{H}$ bytes. The symbol r denotes the memory size of an allocation request, subindexes will be used to identify the activation that made the request. l_i is the maximum amount of live memory required by task T_i.

$\mathcal{L} = \sum_{i=0}^{n} l_i$ will be the maximum amount of memory need by the periodic task system τ.

The simulation model has been focused on studying the behaviour (spatial response) of the allocators when they are executed in our memory model and how the use of a holding time (h) and a maximum amount of memory per period (g) affects each *real-time* allocator. It is important to note that we have executed 50, 000 u.t. (steps) of simulation in all the simulations of this section, moreover, each test has been repeated *at least* 100 times with different seed for the random generator. That is the reason that we have written up our results as an average, followed, when required by a standard deviation.

To measure fragmentation we have considered the factor $\mathcal{F}$, which is calculated as the point of the maximum memory used by the allocator relative to the point of the maximum amount of memory used by the load (live memory).

In order to evaluate the fragmentation generated by each allocator three different tests have been designed. First test (Test 1) consist in 100 periodic tasks with harmonic periods where the allocation size r_i and the maximum memory per period g_i of each task has been generated using uniform distribution in the range of (16, 4096) and (8, 2048), respectively. The holding time h_i of each task has also been calculated with an uniform distribution, $uniform(4 \cdot p_i, 12 \cdot p_i)$. Second test (Test2) uses 200 periodic tasks with non harmonic

periods. with $r_i = uniform(100, 8192)$, $g_i = uniform(200, 16384)$, $h_i = uniform(4 \cdot p_i, 6 \cdot p_i)$, to generate g_i, we have used the same constraint as in Test 1. Third test (Test3) is only conformed by 50 periodic tasks with non harmonic periods. As in the previous tests, we have used an uniform distribution to calculate the parameters of the tasks with the next values: $r_i = uniform(1024, 10240)$, $g_i = uniform(2048, 20480)$, $h_i = uniform(p_i, 16 \cdot p_i)$, to generate g_i, we have used the same constraint as in Test 1.

Table 4 shows the fragmentation obtained by each allocator under these randomly-generated tests.

Table 4. Fragmentation obtained by randomly-generated tests

	FF	BF	BB	HF	TLSF
Test1 Avg.	100.01	4.64	46.37	82.23	**4.33**
Std.Dev.	4.96	0.61	0.74	1.06	**0.55**
Test2 Avg.	85.01	**4.54**	45.00	75.15	4.99
Std.Dev.	4.92	0.67	0.98	1.52	**0.59**
Test3 Avg.	112.51	**7.01**	48.63	99.10	7.69
Std.Dev.	8.53	1.13	1.90	2.61	**0.98**

The results of these tests show that, overall, TLSF is the allocator that requires less memory (less fragmentation) closely followed by Best-Fit. As shown, TLSF behaves better even than Best-Fit in most cases, that can be explained due that TLSF always rounds-up all petitions to fit them to any existing list, allowing TLSF to reuse one block with several petitions with a similar size, independenly of their arrival order. On the other hand, Best-Fit always splits blocks to fit the requested sizes, making impossible to reuse some blocks when these blocks have previously been allocated to slightly smaller petitions.

On the other hand we have Binary-Buddy and Half-Fit, both of them with a fragmentation of up to 99.10% in the case of Half-Fit and 69.59% in Binary-Buddy. As expected, the high fragmentation caused by Binary-buddy is due to the excesive size round up (round up to power of two). All wasted memory of Binary-buddy is caused by internal fragmentation. Half-Fit's fragmentation was also expected because of its *incomplete memory use*. As can be seen, both allocators are quite sensitive request sizes that are not close to power of two, causing a high fragmentation, internal fragmentation in the case of the Binary-Buddy and external one in the Half-Fit case.

6. CONCLUSIONS

TLSF is a dynamic storage allocator designed to meet real-time requirements. This paper has focused on the evaluation of the TLSF in two axis: Time and Space.

The temporal cost of the TLSF is constant which is demonstrated in the experiments. The main conclusion when considering spatial performance is that TLSF algorithm performs as well as Best-fit, which is one of the allocators that better handles and reuses memory. On the other hand Half-Fit handles memory poorly. Half-fit achieves a very fast and constant temporal response time at the expenses of high wasted memory.

We have also extended the already-existing periodic real-time model to include the dynamic memory allocation requirement of the tasks. Based on this model, an experimental framework has been constructed to compare the efficiency, regarding wasted memory, of several dynamic storage allocators. The allocators used in the study were those that meet the requirements needed to be used in real-time systems, i.e., allocation and deallocation is performed in bounded time (constant or logarithmic time).

REFERENCES

[1] M.S. Johnstone and P.R. Wilson. The Memory Fragmentation Problem: Solved? In *Proc. of the Int. Symposium on Memory Management, Vancouver, Canada*. ACM Press, 1998.

[2] D. E. Knuth. *The Art of Computer Programming, volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 1973.

[3] D. Lea. A Memory Allocator. *Unix/Mail*, 6/96, 1996.

[4] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *16th Euromicro Conference on Real-Time Systems*, pages 79–88, Catania, Italy, July 2004. IEEE.

[5] Norman R. Nielsen. Dynamic memory allocation in computer simulation. *Commun. ACM*, 20(11):864–873, 1977.

[6] T. Ogasawara. An algorithm with constant execution time for dynamic storage allocation. *2nd Int. Workshop on Real-Time Computing Systems and Applications*, page 21, 1995.

[7] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the Association for Computing Machinery*, 18(3):416–423, 1971.

[8] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the Association for Computing Machinery*, 21(3):491–499, 1974.

[9] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.*, 20(3):242–244, 1977.

[10] R. Sedgewick. *Algorithms in C. Third Edition*. Addison-Wesley, Reading, Massachusetts, USA, 1998.

[11] J.E. Shore. On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies. *Communications of the ACM*, 18(8):433–440, 1975.

[12] C. J. Stephenson. Fast fits: New methods of dynamic storage allocation. *Operating Systems Review*, 15(5), October 1983. Also in Proceedings of Ninth Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983.

[13] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In H.G. Baker, editor, *Proc. of the Int. Workshop on Memory Management, Kinross, Scotland, UK*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 1995. Vol:986, pp:1–116.