TLSF: a New Dynamic Memory Allocator for Real-Time Systems*

M. Masmano, I. Ripoll, A. Crespo, and J. Real
Universidad Politécnica de Valencia, Spain.
{mmasmano,iripoll,alfons,jorge}@disca.upv.es

Abstract

Dynamic storage allocation (DSA) algorithms play an important role in the modern software engineering paradigms and techniques (such as object oriented programming). Using DSA increases the flexibility and functionalities of applications. There exists a large number of references to this particular issue in the literature. However, the use of DSA has been considered a source of indeterminism in the real-time domain, due to the unconstrained response time of DSA algorithms and the fragmentation problem.

Nowadays, new real-time applications require more flexibility: the ability to adjust system configuration in response to workload changes and application reconfiguration. This aspect adds new value to the definition and implementation of dynamic storage allocation algorithms.

Considering these reasons, new DSA algorithms with a bounded and acceptable timing behaviour must be developed to be used by Real-Time Operating Systems (RTOSs). In this paper a new DSA algorithm called Two-Level Segregated Fit memory allocator (TLSF), developed specifically to be used by RTOS, is introduced. The TLSF algorithm provides explicit allocation and deallocation of memory blocks with a temporal cost $\Theta(1)$.

Keywords: Real-time operating systems, memory allocation.

1 Introduction

The study of dynamic storage allocation (DSA) algorithms is an important topic in the operating systems research and implementation areas and has been widely analysed. Jointly with other basic computing problems, like searching and sorting, memory management is one of the most studied and analysed problems. Wilson et al. [14] wrote an excellent survey/tutorial summarising the research carried out between 1961 and 1995 about storage allocation. Due to the large amount of existing DSA algorithms, the reader can get to think that the problem of dynamic memory allocation has been already solved. This can be true for most application types, but the situation for real-time applications is quite different.

In real-time systems, it is needed to know in advance the operation time bounds in order to analyse the system schedulability. The goal of DSA algorithms is to provide dynamically to the applications the amount of memory required at run time. In general, these algorithms are designed to provide *good average response times*, whereas real-time applications will instead require the response times of the memory allocation and deallocation primitives to be *bounded*.

Additionally, dynamic memory management can introduce an important amount of memory fragmentation, that may result in an unreliable service when the application runs for large periods of time.

It is important to point out that a fast response time and high thoughtput are characteristics that need to be taken into account in any kind of system, and in particular in real-time systems. But the main requirement that defines these systems is to guarantee the timing constraints. Most DSA algorithms have been designed to provide fast response time for the most probable case, achieving a good overall performance, although their worst-case response time can be high or even unbounded.

For these reasons, most RTOS developers and researchers avoid the use of dynamic memory at all, or use it in a restricted way, for example, only during the system startup: *"Developers of real-time systems avoid the use of dynamic memory management because they fear that the worst-case execution time of dynamic memory allocation routines is not bounded or is bounded with a too important bound"*. [10]

This paper introduces a new algorithm, called TLSF, for dynamic memory allocation that presents a bounded worst-case response time, while keeping the efficiency of the allocation and deallocation operations with a temporal cost of $\Theta(1)$. In addition, the fragmentation problem has higher impact in system performance with long time run-

*This work has been supported by the European Commission project number IST-2001-35102 (OCERA).

ning applications. A small and bounded fragmentation is also achieved by the proposed algorithm.

The following section reviews the set of real-time requirements that should be fulfilled by the DSA algorithms for real-time applications. Section 3 describes the fragmentation problem that appears in DSA. Section 4 gives an overview of the kinds of dynamic storage allocators and their strategies. Section 5 presents the general operational model of memory allocators. Section 6 details the main criteria used to design the TLSF allocator. The structures and characteristics of the TLSF are presented in section 7. Section 8 evaluates the complexity of the algorithm. Section 9 compares the results obtained by the proposal versus other approaches. Finally, section 10 outlines our conclusions and future direction of this work.

2 Real-Time Requirements for DSA

One of the key issues in real-time systems is the schedulability analysis to determine whether the system will satisfy the application's timing constraints at run-time or not. Regardless of the analysis and scheduling techniques used, it is essential to determine the worst-case execution time (WCET) of all the running code, including application code, library functions and operating system.

Another characteristic that differentiates real-time systems from other kinds of systems is that real-time applications run for large periods of time. Most non-real-time applications take only a few minutes or hours to complete their work and finalise. Real-time applications are usually executed continuously during the whole life of the system (months, years,...). This behaviour directly affects one of the critical aspects of dynamic memory management: the memory fragmentation problem.

Considering all these aspects, 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 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 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**. Non-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 to minimise the chances of exhausting the memory pool by minimising the amount of fragmentation and wasted memory.

With respect to memory requirements, there exists a large range of RTSs, from small embedded systems with very little memory, no virtual memory support (MMU) and no permanent storage; to large, redundant, multi-processor systems. Our study and the proposed algorithm have focused on small systems where memory is a scarce resource and there is no MMU support.

Regarding the way allocation and deallocation are managed, we may find two general approaches for DSA: (*i*) explicit allocation and deallocation, where the application needs to explicitly call the primitives of the DSA algorithm to allocate memory (e.g., `malloc`) and to release it (e.g., `free`) when it is not needed anymore; and (*ii*) implicit memory deallocation (also known as *Garbage Collection*), where the DSA mechanism is in charge of collecting the blocks of memory that have been previously requested but are not needed anymore, in order to make them available for new allocation requests. This paper is focussed on explicitly low level allocation primitives, garbage collection is not addressed in this work.

3 Fragmentation

Besides timing faults (due to an unbounded time DSA implementation) a memory allocation may fail due to memory exhaustion. Memory exhaustion may occur due to two reasons: (*i*) the application requires more memory that the total memory available in the system; or (*ii*) the DSA algorithm is unable to reuse memory that is free.

The DSA algorithm can do nothing in the first case; the application has to be redesigned or more physical memory has to be added to the system. Historically the second reason has been considered as two different aspects called internal and external fragmentation. Recent analysis [14] considers these two fragmentation categories as a single problem called *wasted memory* or simply *fragmentation*.

Fragmentation has been considered a serious issue. Many initial experimental studies based on synthetic or random workloads yield to the conclusion that the fragmentation problem may limit or even prevent the use of dynamic memory. Johnstone et al. [4] showed that the fragmentation problem with *real* (non synthetic) workload is not that grave and can be efficiently managed by well-designed allocators. Their conclusions state that *"the fragmentation problem is really a problem of poor allocator implementations*". Other interesting conclusions of this work are:

- best-fit or physical first-fit policies produce low fragmentation,

- blocks of memory should be immediately coalesced upon release, and
- preferably reuse the memory that has been recently released over those blocks that has been released further in the past.

The results of Johnstone et al. [4] also showed that Doug Lea's allocator [6], which implements a good-fit policy (a nearly best-fit based on a segregated fit policy), performs very well in terms of fragmentation.

Although Johnstone et al. used non-real-time applications as workload (gcc, ghostscript, perl, etc.) there is no reason to think that real-time applications will show a behaviour different enough to invalidate their results. The main difference between real-time and non-real-time applications is that many real-time applications do not use dynamic memory; memory is statically allocated at start time, which is very safe but inflexible.

4 DSA Algorithms

The goal of DSA algorithms is to grant the application the access to blocks of memory from a pool of free memory blocks. The different algorithm strategies differ in the way they find a free block of the most appropriate size. Historically, DSA algorithms have been be classified according to the following categories [14]:

- **Sequential Fit**: These algorithms are the most basic ones; they are based on a single or double linked list of all free memory blocks. Examples of algorithms based on this strategy are Fast-Fit, First-Fit, Next-Fit, and Worst-Fit. Sequential Fit is not a good strategy for RTSs because it is based on a sequential search, whose cost depends on the number of existing free blocks. This search can be bounded but the bound is not normally acceptable.
- **Segregated Free Lists**: Segregated Free Lists algorithms use an array of lists containing the references to free blocks of memory of a particular size (or inside a particular size range). When a block is deallocated, it is inserted in the free list corresponding to its size. It is important to remark that the blocks are logically but not physically segregated. There are two variants of this strategy: Simple Segregated Storage and Segregated Fits. Examples of this strategy are Standish and Tadman's "Fast-Fit", which uses an array of several, small size free lists and a binary tree for the free lists for larger sizes; and the DSA developed by Douglas Lea [6]. This strategy is acceptable to be used in RTOSs, since the searching cost is not dependent on the number of free blocks.
- **Buddy Systems**: Buddy Systems [5] [9] are a variant of Segregated Free Lists, with efficient split and merge operations[1]. There are several variants of this method, like Binary Buddies, Fibonacci Buddies, Weighted Buddies, and Double Buddies.

  Buddy Systems exhibit good timing behaviour, adequate for RTOSs; but they have the disadvantage of producing large internal fragmentation, of up to 50% [4].
- **Indexed Fit**: This strategy is based on the use of advanced structures to index the free memory blocks, with several interesting features. Examples of algorithms using Indexed Fit are: a Best-Fit algorithm using a balanced tree, Stephenson's "Fast-Fit" allocator, which is based on a Cartesian tree to store free blocks with two indexes [13], size and memory address, etc. The Indexed Fit strategy can perform better than Segregated Free Lists if the search time of a free block does not depend on the number of free blocks.
- **Bitmap Fit**: Bitmap Fit methods are a variant of Indexed Fit methods that use a bitmap to know which blocks are busy or free. An example of this method is the Half-Fit algorithm [8]. This approach has the advantage with respect to the previous ones, that all the information needed to make the search is stored in a small piece of memory, typically 32 bits, therefore reducing the probability of cache misses and improving the response time [12].

5 DSA Operational Model

A DSA is an abstract data type that keeps track of which blocks of memory are in use and which are free. It must provide at least two operations to allocate and release memory. The performance of a DSA is mostly determined by the type of data structure used to manage free blocks. In order to better understand the impact of the internal data structure, it is interesting to analyse how most dynamic storage allocators manage free memory blocks:

- Initially, the memory that will be managed by the DSA is a single, large block of free memory, usually called the *initial pool*. It is also possible that the allocator relies on the underlying OS (or the hardware MMU) to request new memory areas to manage.
- First allocation requests are fulfilled by taking blocks of memory from the initial pool. The initial pool is shrinked accordingly.

[1] Split and merge are required to rearrange lists after servicing allocation and deallocation requests.

- When a previously allocated block is released two things may occur depending on the physical position of the released block:

  1. the released block can be merged with the initial pool or with one or more free adjacent blocks; or
  2. the free block is surrounded by allocated blocks and can not be merged.

  In the first case, the DSA algorithm may coalesce the block or return it to the initial pool. In the second case, the DSA algorithm inserts the free block in the data structure of free blocks.

- If there are free blocks, then new allocation requests can be fulfilled by using the initial pool or by searching for a free block large enough to fit the requested block size. The internal data structure where free blocks are stored and the method used to search a suitable block are the heart of the DSA, and determine its performance. If the free block used to fulfill the allocation request is bigger than the requested size, then the block has to be split and the non-allocated part has to be returned to the data structure of free blocks.

Programs using dynamic memory invoke two operations: `malloc()` to dynamically obtain a block of memory, and `free()` to release the previously requested block. These operations are managed by the DSA algorithm which requires to perform basic operations to manage free blocks (inserting, removing or searching). These operations are now described in more detail:

Insert a free block : This operation may be required by both `free` and `malloc`. In the first case, the inserted block is the one released by the application or a larger one if it was coalesced. In the case of `malloc`, the inserted block is the remaining part of a memory block that is larger than the amount of memory requested.

Search for a free block of a given size or larger : It finds a free block large enough to serve the application's request. As said above, the criteria used to find the block (First-Fit, Best-Fit, etc.) and how the DSA data structure design is what mostly determines the performance of the DSA.

Search for a block adjacent to another : In order to coalesce a free block, it is necessary to find the physically adjacent free blocks, if any.

Remove a free block : This operation may also be needed both for `free` and `malloc`. The first case occurs when the block can be coalesced, in which case the free adjacent block is removed and merged with the new free block. The second case occurs when `malloc` finds a suitable block in the free data structure.

6 Design Criteria of TLSF

There is not a single DSA algorithm suitable for all application types. As already stated, real-time applications requirements are quite different from conventional, general-purpose applications. In this section we present the design of a new algorithm called Two-Level Segregated Fit (TLSF, for short) to fulfill the most important real-time requirements: a bounded and short response time, and a bounded and low fragmentation.

Moreover, the constraints that should be considered for *embedded* real-time systems are:

- Trusted environment: programmers that have access to the system are not malicious, that is, they will not try to intentionally steal or corrupt application data. The protection is done at the end-user interface level, not at the programming level.

- Small amount of physical memory available.

- No special hardware (MMU) available to support virtual memory.

In order to meet these constraints and requirements, TLSF has been designed with the following guidelines:

Immediate coalescing : As soon as a block of memory is released, TLSF will immediately merge it with adjacent free blocks, if any, to build up a larger free block. Other DSA algorithms may defer coalescing, or even not coalesce at all. Deferred coalescing is a useful strategy in systems where applications repeatedly use blocks of the same size. In such cases, deferred coalescing removes the overhead of continuously merging and splitting the same block.

Although deferred coalescing can improve the performance of the allocator, it adds unpredictability (a block request may require to merge an unbounded number of free, but not merged blocks) and it also increases fragmentation. Therefore, deferred coalescing should not be used in real-time.

Splitting threshold : The smallest block of allocatable memory is 16 bytes. Most applications, and real-time applications are not an exception, do not allocate simple data types like integers, pointers or floating point numbers, but more complex data structures which contains at least one pointer and a set of data.

By limiting the minimum block size to 16 bytes, it is possible to store, inside the free blocks, the information needed to manage them, including the list of free blocks pointers. This approach optimises the memory usage.

Good-fit strategy : TLSF will *try to* return the smallest chunk of memory big enough to hold the requested block. Since most applications only use blocks of memory within a small range of sizes, a Best-Fit strategy tends to produce the lowest fragmentation on real workloads, compared to other approaches such as first-fit [14, 4]. A Best-Fit (or almost Best-Fit, also called *Good-Fit*) strategy can be implemented in an efficient and predictable manner by using segregated free lists.

On the other hand, other strategies like First-Fit or Next-Fit are difficult to implement with a predictable algorithm. Depending on the sequence of requests, a First-Fit strategy can degenerate in a long sequential search in a linked list.

TLSF implements a Good-Fit strategy, that is, it uses a large set of free lists, where each list is a **non-ordered** list of free blocks whose size is between the size class (a range of sizes) and the next size class. Each segregated list contains blocks of the same class.

No reallocation : We assume that the original memory pool is a single large block of free memory, and no `sbrk()`[2] function is available. Since general purpose OSs (like UNIX®) provide virtual memory, most DSA algorithms were designed to take advantage of it. By using this capability, it is possible to design a DSA optimised to manage relatively small memory blocks, delegating the management of large blocks to the underlying OS by dynamically enlarging the memory pool.

TLSF has been designed to provide complete support for dynamic memory, that is, it can be used by the applications and also by the OS, using no virtual memory hardware.

Same strategy for all block sizes : The same allocation strategy is used for any requested size. One of the most efficient and widely used DSA, Douglas Lea's allocator [6], uses four different strategies depending on the size of the request: First-Fit, cached blocks, segregated list and a system managing facility to enlarge the pool. This kind of algorithms do not provide a uniform behaviour, so their worst-case execution time is usually high or even difficult to know.

Memory is not cleaned-up : Neither the initial pool nor the free blocks are zeroed. DSA algorithms used in multi-user environments have to clean the memory (usually filling it with zeros) in order to avoid security problems. Not cleaning the memory before it is allocated to the user is considered a serious security flaw, because a malicious program could obtain confidential information.

Nevertheless, we assume that TLSF will be used in a trusted environment, where applications are written by well intended programmers. Therefore, initialising the memory is a useless feature that introduces a considerable overhead. The programmer should not use uninitialised memory, as good programming practice recommends.

[2]The `sbrk()` system call increments the program's data space.

7 TLSF Data Structures

The data structures used by the dynamic memory allocator proposed are described in this section. TLSF algorithm uses a segregated fit mechanism to implement a good-fit policy.

The basic segregated fit mechanism uses an array of free lists, with each array holding free blocks within a size class. In order to speedup the access to the free blocks and also to manage a large set of segregated lists (to reduce fragmentation) the array of lists has been organised as a two-level array. The first-level array divides free blocks in classes that are a power of two apart (16, 32, 64, 128, etc.); and the second-level sub-divides each first-level class linearly, where the number of divisions (referred to as the *Second Level Index*, SLI) is a user configurable parameter (this parameter is further explained in section 7.3). Each array of lists has an associated bitmap used to mark which lists are empty and which ones contain free blocks. Information regarding each block is stored inside the block itself.

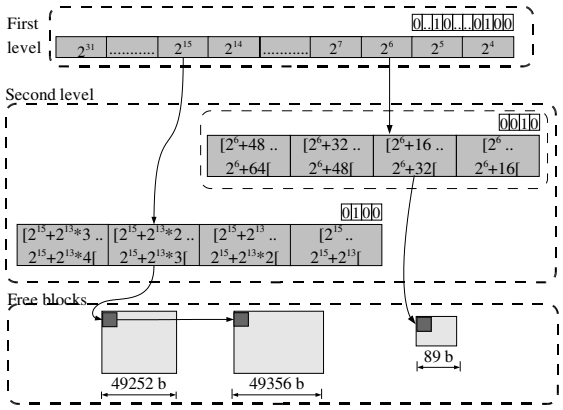

Figure 1. TLSF free data structure overview.

Figure1 shows the two levels in the data structure. The first level is an array of pointers which point to the second level lists of free blocks. In this case, considering the bitmap of the first level, there are only free blocks of size ranges $[2^6, 2^7[$ and $[2^{15}, 2^{16}[$. The bitmap of the second level splits each first level range in four segregated lists.

In order to easily coalesce free blocks, the TLSF employs the *boundary tag* technique proposed by D. Knuth in [5] that originally consisted in adding a footer field to each free or used block, which is a pointer to the beginning of the same block. When a block is released, the footer of the previous block (which is located one word before the released block) is used to access the head of the previous physical block to check whether it is free or not and merge both blocks accordingly. It is possible to simplify the boundary tag technique by allocating the pointer to the head of the block not at the end of each block but inside the header of the next block.

Therefore, each free block is linked in two different double linked lists: 1) the segregated list, holding the blocks belonging to the same size class, and 2) a list ordered by physical address.

7.1 TLSF Block Header

The TLSF embeds into each block the information needed to manage the block (whether the block is free or not) and the pointers to link it into the two lists: the list of blocks of similar sizes and the list ordered by physical addresses. This data structure is called the *block header*.

Since busy (used) blocks are not linked in any segregated list, their block header is smaller than the headers for free blocks.

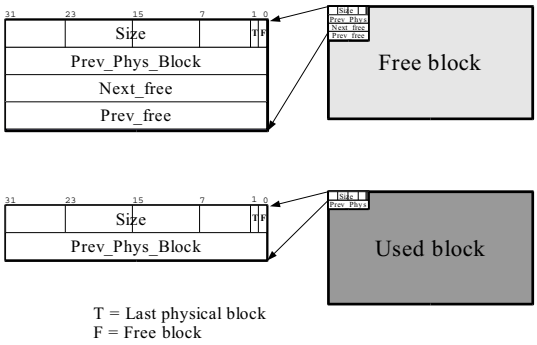

Figure 2. Free and allocated block headers.

The block header of free blocks contains the following data: i) The size of the block, required to free the block and to link this block with the next one in the physical link list; ii) Boundary tag, a pointer to the head of the previous physical block; iii) Two pointers, to link this block into the corresponding segregated list (double linked list).

The header of a busy block contains only the size and the boundary tag pointer.

Since block sizes are always a multiple of four (the allocation unit is the word, 4 bytes) the two least significant bits of the size field are used to store the block status: block is busy or free (bit F), and whether the block is the last one of the pool or not (bit T).

7.2 TLSF Structure Functions

Most TLSF operations rely on the `segregate_list()` mapping function. Given the size of a block, the mapping function calculates the indexes of the two arrays that point to the corresponding segregated list that holds the requested block.

$$\texttt{mapping}\,(size) \rightarrow (f, s)$$
$$\texttt{mapping}\,(size) = \begin{cases} f := \lfloor log_2\,(size) \rfloor \\ s := \left(size - 2^f\right) \frac{2^{SLI}}{2^f} \end{cases}$$

This function can be efficiently implemented using bit search instructions (available on most modern processors) and exploiting some numerical properties. The first level index ($\lfloor log_2\,(size) \rfloor$) can be computed as the position of the most significant bit of the size that is set to one. The second level index can be obtained by extracting the following SLI bits of the size (right-ward). For example, supposing a Second Level Index SLI=4 and given the size 460, the first level index is `f=8` and the second level index is `s=12`:

$$\texttt{size} = 460_d = \overset{15\,14\,13\,12\,11\,10\,9\,8}{0\,0\,0\,0\,0\,0\,0\,1}\ \underbrace{\overset{7\,6\,5\,4}{1\,1\,0\,0}}_{\texttt{s=12}}\ \overset{3\,2\,1\,0}{1\,1\,0\,0}_b \quad \texttt{f=8}$$

The operations provided by TLSF structure are:

- **Initialise TLSF structure**: this function will create the TLSF data structure at the beginning of the pool (the remaining part of the pool will be the initial free memory). It accepts three parameters: a pointer to the memory pool, the size of the pool, and the second level index. It is possible to create several independent TLSF pools.
- **Destroy TLSF structure**: this function marks the given pool as non-usable.
- **Get a free block**: returns a pointer to a free area of memory of the requested size or bigger. The requested size is rounded up to the nearest free list. This operation is implemented as follows:

  First step: calculate the "f" and "s" indexes which are used to get the head of the free list holding the closer class (segregated) list. If this list is not empty then the block at the head of the list is removed from the list (marked as busy) and returned to the user; otherwise,

  Second step: search the next (bigger than the requested size) non empty segregated list in the TSLF

data structure. This search is done in constant time using the bitmap masks and the find first set (`ffs`) bitmap instruction.

If a list is found, then the block at the head of the list will be used to fulfill the request. Since this block is bigger than requested, it has to be split and the remaining is inserted into the corresponding segregated free list. On the other hand, if a free list is not found, then the request fails.

- **Insert a free block**: this function inserts a free block in the TLSF structure. The mapping function is used to calculate the "f" and "s" indexes to find the segregated list where the block has to be inserted.
- **Coalesce blocks**: using the boundary tag technique, the head of the previous block is checked to see whether it is free or not. If the block is free then the block is removed from the segregated list and merged with the current block. The same operation is carried out with the next physical block. Once the block has been merged with its free neighbour blocks, the new big block is inserted in the appropriate segregated list.

7.3 TLSF Structure Parameters

The TLSF structure is characterised by three basic parameters:

- **First Level Index** (FLI): it is the number of first-level segregated classes. Classes are a power of two apart from each other. The FLI is dynamically calculated at the TLSF initialization by the following expression: $FLI = \min\left(\log_2(memory\_pool\_size), 31\right)$
- **Second Level Index** (SLI): this index subdivides linearly the first-level ranges. For efficiency reasons, SLI has to be a power of two, and should be in the range $[1, 32]$ so that simple word bitmap processor instructions can be used. For convenience, SLI is expressed as the $\log_2$ of the number of second-level division (e.g. SLI=4 subdivides each fist-level segregated class into 16 segregated lists).

  Since the number of different segregated lists is determined by the value of SLI, the bigger SLI, the smaller the fragmentation (see section 8.1). This is a user-defined parameter. Reasonable values are 4 or 5.
- **Minimum block size** (MBS): This parameter is defined to limit the minimum block size. Considering implementation reasons, the MBS constant is set to 16 bytes.

From the initial parameters FLI, SLI and MBS, some other parameters can be determined:

- The existing number of lists: $2^{SLI}\left(FLI - log_2\left(MBS\right)\right)$.
- The range of sizes accepted by each list:

$$\left]2^i + \left(\frac{2^i}{2^{SLI}} \cdot j\right), next\_size\right]$$

$$\forall i, j \in \mathbb{Z}^+ | \left(log_2\left(MBS\right) \leq i < FLI\right) \wedge \left(0 \leq j < 2^{SLI}\right)$$

and where

$$next\_size = \begin{cases} 2^{i+1} - 1 & \text{if } j = 2^{SLI} \\ 2^i + \left(\frac{2^i}{2}^{SLI}\left(j+1\right)\right) & \text{otherwise} \end{cases}$$

- The size of the data structures:

$$TFH + \left(PS \cdot 2^{SLI} \cdot \left(FLI - \log_2(MBS)\right)\right)$$

where TFH is the fixed header of the data structure (40 bytes) and PS is the pointer size (4 bytes). For a maximum pool of 4 GB (FLI=32) and the maximum SLI (SLI=5) the size of the data structures required is 3624 bytes. A pool of 32 MB (FLI=25) and SLI=5 needs 2856 bytes.

7.4 TLSF Structure Optimisations

Some additional optimisations are possible to make the TLSF data structures and the search operations more efficient. This is achieved by using several strategies:

- **Cache and TLB Locality Strategies**:
  - The TLSF data structure has been organised as a two level array and not as a bi-dimensional array. This allows to take advantage of cache locality. Several tests showed that this approach is more efficient than using a bi-dimensional array.
  - In addition, the TLSF data structure is created dynamically at the beginning of the memory pool. This approach uses some free memory, but it allows to have the TLSF data structure and the memory pool in the same memory page.
- **Other useful strategies**:
  - Bitmaps are used to track non-empty lists, this is the reason why FLI and SLI must be smaller or equal to 32. This allows to reduce the number of memory accesses and to use few processor instructions to perform the search.
  - The mapping function is calculated as follows:

```
mapping (size_t size, unsigned *fl,
                      unsigned *sl){
  // fls() => Find_Last_Set bitmap function
  *fl = fls(size);
  *sl = ((size ^ (1<<fl)) >> (*fl - SLI));
}
```

8 TLSF Analysis

It is possible to obtain the time complexity for the `malloc` and `free` operations from the TLSF pseudo-code. The `malloc` pseudo-code is the following:

```
void *malloc(size){
  int fl, sl, fl2, sl2;
  void *found_block, *remaining_block;
  mapping (size, &fl, &sl);                          // O(1)
  found_block=search_suitable_block(size,fl,sl);// O(1)
  remove (found_block);                              // O(1)
  if (sizeof(found_block)>size) {
    remaining_block = split (found_block, size);
    mapping (sizeof(remaining_block),&fl2,&sl2);
    insert (remaining_block, fl2, sl2);              // O(1)
  }
  remove (found_block);                              // O(1)
  return found_block;
}
```

Although malloc has to do a search in the TLSF structure (`search_suitable_block`), its asymptotic cost is O(1) because to do a search in the TLSF structure has always a constant worst-case response time due to the existence of bitmap search processor instructions. The `remove` function simply unlinks the block from the segregated fit list; and the `insert` function inserts in the head of the corresponding segregated list.

The pseudo-code of the `free` function is shown below:

```
void free(block){
  int fl, sl;
  void *big_free_block;
  big_free_block = merge(block);                  // O(1)
  mapping (sizeof(big_free_block), &fl, &sl);
  insert (big_free_block, fl, sl);                // O(1)
}
```

The function `merge` checks whether the previous and next physical blocks are free and tries to merge them with the released block. No search has to be done since previous and next physical blocks are linked with the released block.

All internal operations of `malloc` and `free` have a constant time cost and there is no loop nor recursion, therefore their asymptotic worst case response time is:

malloc()	free()
O(1)	O(1)

8.1 Fragmentation

The fragmentation caused by TLSF is due to the fact that it does not perform an exhaustive search in the corresponding segregated list to find a suitable block to fulfill the request. TLSF uses the mapping function and bitmaps to directly find the non-empty smallest segregated list that contains blocks of the same size or bigger than the requested one. Once the corresponding segregated list has been found, the first block of the list is used to serve the request. Therefore, it may happen that there exist free blocks big enough to fulfill a request but stored in the previous segregated list (one segregated list before the one returned by the mapping function).

The worst-case fragmentation occurs when the biggest free block has the largest size of its segregated list (the size of the free block is one word less than the minimum size of the next segregated list), and the application requests a block whose size is one word larger than the start size of the segregated list. TLSF will try to find a suitable block to serve the request starting from the next segregated list that holds the free block, therefore, the request will fail.

The fragmentation can be calculated with the following function: $fragmentation = \frac{2^{fli(pool\_size)}}{SLI} - 2$

The binary representation of the block size gives a more intuitive view of the source of fragmentation in TLSF. For example, assuming $SLI = 5$, the largest free block that will be in the segregated list defined by $f = 12$ and $s = 13$ is 5887:

$$5887_d = \overset{15\,14\,13\,12}{0\,0\,0\,1}\ \underbrace{\overset{11\,10}{0\,1}}_{\text{f=12}}\ \underbrace{\overset{9\,8\,7}{1\,0\,1}}_{\text{s=13}}\ \overbrace{\overset{6\,5\,4\,3\,2\,1\,0}{1\,1\,1\,1\,1\,1\,1}}^{\text{class}}{}_b$$

But the search for a 5761-byte block ($5761_{10} = 0001011010000001_2$) will start in the segregated list indexed by ($f = 12$ and $s = 14$), which is empty. Therefore in this example, the fragmentation will be 126.

It is possible to modify the TLSF algorithm to perform exact search (by looping in the previous segregated list) when the initial fixed time search fails. Although this modification will reduce the worst-case fragmentation, the worst-case response time will be no longer bounded.

9 Experimental Analysis

In the literature, we can find two different approaches to evaluate the performance of DSA. One of them uses standard applications (gcc, espreso, cfrag, etc.) or synthetic workload models [16] based on real application traces. The second approach consists in the design of synthetic workload that reproduces worst-case scenarios [11].

Since the requirements of real-time applications are quite different from the rest, the results obtained using standard workload do not show how the algorithm behaves in worst-case situations. Therefore, in order to obtain experimental results that really show the performance parameters relevant to real-time applications, synthetic workload has been used. Each test workload has been designed trying to produce the worst-case scenario for each allocator. The set of allocators analysed were: First-Fit, Best-Fit, Douglas Lea's malloc, Binary Buddy, and TLSF.

malloc()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	worst	mean	worst	mean	worst	mean	worst	mean	worst	mean
Test 1	**25636208**	**11256641**	**17007384**	**10322776**	168	81	4140	1239	155	148
Test 2	2124	1971	2124	1971	**16216**	**15974**	244	220	172	148
Test 3	568	201	592	197	128	76	**5660**	**5448**	120	115
Test 4	792	235	536	193	124	75	5460	5309	**189**	**168**

free()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	worst	mean	worst	mean	worst	mean	worst	mean	worst	mean
Test 1	528	103	2012	899	460	67	2728	345	140	97
Test 2	428	150	428	150	**96**	**71**	1976	1448	152	96
Test 3	340	107	324	113	560	131	**6512**	**3504**	124	93
Test 5	348	157	372	204	136	68	3728	2881	**188**	**164**

Table 1. Test summary for the malloc and free operations. Results are given in number of processor cycles.

9.1 Workload Generation

The worst case scenario for the First-Fit and the Best-Fit is the same [11]. It occurs when the list of free blocks is full of blocks (the pool alternates between busy and free blocks of minimum size) and the application requests a block that can only be served using the last free block.

Douglas Lee's malloc uses several strategies depending on the size of the block: it uses a block cache strategy for small blocks (<64 B), segregated lists (<512 B), First-Fit (<128 KB) and relies on system allocator (`sbrk`) for very large requests. Therefore, the worst-case scenario is the same as the one for First-Fist but with sizes greater than 512.

The Binary Buddy worst-case is given when all the memory is free (there is only one big free block) and the application requests a block of the smallest size. The allocator has to split the initial block $\log_2(pool\_size)$ times. The release of this small block is also the worst case for the free operation.

Since the TSLF operation does not depend on the number of free or busy blocks and there are no loops in the code, only small variation in the execution time can be obtained due to which branch of its conditional code is executed. The worst case for `malloc` occurs when there is only one large free block and the application requests a small block. And for the `free` operation it occurs when the released block has to be coalesced with its two neighbour blocks.

9.2 Experimental Results

Experimental results have been measured on an Intel Pentium P4 2.8 Ghz machine with 512 KB of cache memory and 512 MB of main memory. Timing measures on table 1 are expressed in number of processor cycles. Each test is executed 1024 times to obtain the maximum and mean times. Following is the description of the tests:

Test-1 `malloc/free` worst case for First-Fit. The memory pool is 1 MB. All the memory except the last 32 bytes are allocated in blocks of minimum size (16 bytes), then odd blocks are released. After this startup, the worst case occurs when a block of 32 bytes is requested.

Test-2 `malloc/free` worst case for Douglas Lea's malloc. The memory pool is 256 KB and the same request sequence that in Test-1, but requesting blocks of 512 bytes. And the last block has a size of 530 bytes.

Test-3 `malloc/free` worst case for Binary Buddy. The memory pool is 2 MB. No blocks were allocated initially. The worst case occurs when a 16 bytes block is requested.

Test-4 `malloc` worst case for TLSF. The memory pool is 2 MB. No blocks are allocated initially. Then a 40-byte block is requested.

Test-5 `free` worst case for TLSF. The memory pool is 1 MB. Three 512-byte blocks are allocated, then the first and third blocks are released. The time to release the second block is the worst-case scenario.

As shown in table 1, the worst-case response time of TLSF is always bounded and presents a highly regular behaviour. On the other hand, the designed worst-case scenario for each allocator produces a highly degraded response time. The test sources and a more detailed evaluation with different FLI and SLI parameters can be found in [7].

10 Conclusions and Future Work

The main problems of DSA, such as the lack of bounded and fast response time primitives, and the high fragmentation, have been a limitation for the use of dynamic memory

in real-time systems. Several DSA algorithms, as Buddy systems, have improved some aspects like the response time, but maintaining the fragmentation in non acceptable limits.

In this paper, we have presented a new algorithm called TLSF, based on two levels of segregated lists which provides explicit allocation and deallocation of memory blocks with a temporal cost $\Theta(1)$. The proposed algorithm has shown excellent experimental results with respect to both response time and fragmentation.

The use of bitmaps and segregated lists to access to different free memory blocks of different size in an efficient and predictable way, permits to obtain constant cost for the basic operations *malloc* and *free*. Also, the analysis of the proposed data structures shows a low and bounded amount of internal fragmentation, dependent on the value chosen for the second level index.

The proposal has been compared with other DSA algorithms obtaining very encouraging results.

This proposal has been implemented in the framework of the OCERA [3] project and has been integrated in *RT-LinuxGPL* [15, 2] and *MARTE OS* [1].

Additional studies are required to investigate a wider variety of real-time applications, both to measure their performance on the real applications and to determine the degree to which arbitrary code complies with the stated requirements described in section 2 of this paper. Also, the integration of this algorithm in a memory resource management able to provide quality of service considering the CPU and memory management in a integrated way is future work.

Acknowledgements

We are grateful to Andy Wellings for his advice in the last version of the paper. We also thank to the paper reviewers for their many valuable comments. Finally, we are grateful to Hermann Hrtig for his comments in the OCERA review about memory management which motivated this research.

References

[1] M. Aldea and M. González-Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. *Reliable Software Technologies - Ada Europe, Lecture Notes in Computer Science*, 2043:305–316, 2001.

[2] RT-Linux-GPL distribution. www.rtlinux-gpl.org.

[3] OCERA: Open Components for Embedded Real-Time Applications. 2002. IST 35102 European Research Project. (http://www.ocera.org).

[4] M.S. Johnstone and P.R. Wilson. The Memory Fragmentation Problem: Solved ? In *Proc. of the Int. Symposium on Memory Management (ISMM'98), Vancouver, Canada*. ACM Press, 1998.

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

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

[7] M. Masmano. TLSF Implementation and Evaluation in OCERA. Technical Report OCERA D4.4, Real Time Research Group. Universidad Politecnica de Valencia, 2004. Available at: http://www.ocera.org and http://rtportal.upv.es.

[8] 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.

[9] J.L. Peterson and T.A. Norman. Buddy Systems. *Communications of the ACM*, 20(6):421–431, 1977.

[10] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. *14 th Euromicro Conference on Real-Time Systems*, page 41, 2002.

[11] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. Technical Report 1429, Institut de Recherche en Informatique et Systemes Aleatories, 2002.

[12] Filip Sebek. Cache Memories in Real-Time Systems. Technical Report 01/37, Mälardalen Real-Time Research Centre, Sweden, October 2 2001.

[13] C.J. Stephenson. Fast Fits: New Methods for Dynamic Storage Allocation. In *Proc. of the 9th ACM Symposium on Operating Systems Principles*, volume 17 of *Operating Systems Review*, pages 30–32. ACM Press, 1983.

[14] 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.

[15] V. Yodaiken and M. Barabanov. A real-time linux. Online at http://rtlinux.cs.nmt.edu/rtlinx/u.pdf.

[16] Benjamin Zorn and Dirk Grunwald. Evaluating Models of Memory Allocation. *ACM Transactions on Modeling and Computer Simulation*, pages 107–131, 1994.