SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2008; **38**:995–1026
Published online 26 October 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.858

Implementation of a constant-time dynamic storage allocator

M. Masmano[1], I. Ripoll[1], J. Real[1], A. Crespo[1,*,†] and A. J. Wellings[2]

[1]*Department of Computer Engineering, Universidad Politécnica de Valencia, Valencia, Spain*
[2]*Department of Computer Science, University of York, York, U.K.*

SUMMARY

This paper describes the design criteria and implementation details of a dynamic storage allocator for real-time systems. The main requirements that have to be considered when designing a new allocator are concerned with temporal and spatial constraints. The proposed algorithm, called TLSF (two-level segregated fit), has an asymptotic constant cost, *O*(1), maintaining a fast response time (less than 200 processor instructions on a x86 processor) and a low level of memory usage (low fragmentation). TLSF uses two levels of segregated lists to arrange free memory blocks and an *incomplete search* policy. This policy is implemented with word-size bitmaps and logical processor instructions. Therefore, TLSF can be categorized as a good-fit allocator. The incomplete search policy is shown also to be a good policy in terms of fragmentation. The fragmentation caused by TLSF is slightly smaller (better) than that caused by best fit (which is one of the best allocators regarding memory fragmentation). In order to evaluate the proposed allocator, three analyses are presented in this paper. The first one is based on worst-case scenarios. The second one provides a detailed consideration of the execution cost of the internal operations of the allocator and its fragmentation. The third analysis is a comparison with other well-known allocators from the temporal (number of cycles and processor instructions) and spatial (fragmentation) points of view. In order to compare them, a task model has been presented. Copyright © 2007 John Wiley & Sons, Ltd.

Received 25 January 2007; Revised 4 July 2007; Accepted 16 September 2007

KEY WORDS: dynamic storage management; real-time systems; operating systems

*Correspondence to: A. Crespo, Department of Computer Engineering, Universidad Politécnica de Valencia, Camino de Vera, 14, E-46071 Valencia, Spain.
†E-mail: acrespo@disca.upv.es

Contract/grant sponsor: FRESCOR; contract/grant number: IST/5-034026
Contract/grant sponsor: ARTIST2; contract/grant number: IST NoE 004527
Contract/grant sponsor: Thread; contract/grant number: TIC2005-08665

1. INTRODUCTION

Although dynamic storage allocation (DSA) has been extensively studied, it has not been widely used in real-time systems due to the commonly accepted idea that, because of the intrinsic nature of the problem, it is difficult or even impossible to design an efficient, time-bounded algorithm. An application can request and release blocks of different sizes in a sequence that is, *a priori*, unknown to the allocator. It is no wonder that the name *DSA* suggests the idea of unpredictable behaviour.

An allocator must keep track of released blocks in order to reuse them to serve new allocation requests, otherwise memory will eventually be exhausted. A key factor in an allocator is the data structure that it uses to store information about free blocks. Although not explicitly stated, it seems that it has been accepted that even when using a very efficient and smart data structure the allocation algorithm, in some cases, has to perform some form of linear or logarithmic search to find a suitable free block; otherwise, significant fragmentation[‡] may occur.

There are two general approaches to DSA management:

1. *Explicit allocation and deallocation*—where the application has to explicitly call the primitives of the DSA algorithm to allocate memory (e.g. malloc) and to release it (e.g. free).
2. *Implicit memory deallocation* (also known as garbage collection)—where allocation is explicit but the DSA is in charge of collecting the blocks of memory that have been previously requested but are not needed any more.

This paper is focused on the design and implementation of an explicit low-level allocation and deallocation primitives. Garbage collection is not addressed in this work.

Explicit allocation and deallocation are usually performed at the individual object level. An alternative to this fine-grained approach is *region-based* memory allocation, which has been used as an implementation technique under a variety of names (zones or groups). Regions have also recently attracted research attention as a target for static inference of memory management [1]. In a region-based memory allocation scheme, each allocated object is placed in a program-specific region. Memory is reclaimed by destroying a region, freeing all the objects allocated therein. Region management has a low overhead. The programmer can control how long allocated values will live. Whole regions can be deallocated at once. This can often result in a speed-up over manual memory management. The main drawbacks of this technique are that the programmer has to group objects into appropriate regions according to their lifetimes, run-time checks are required to avoid dangling pointers and memory fragmentation can occur when regions are destroyed [2].

There are several DSA strategies that have been proposed and analysed under different real or synthetic loads. In [3], a detailed survey of DSA was presented, which has been considered the main reference since then. The authors presented a comprehensive description, as well as the most important results, of all problems related to memory allocation: the basic problem statement, fragmentation, taxonomy of allocators, coalescing, etc. The paper also contains an outstanding chronological review of all related research starting from four decades ago.

[‡] Although the term 'wasted memory' describes better the inability to use some parts of the memory, historically the term 'fragmentation' has been used.

DOI: 10.1002/spe

Several efficient implementations of dynamic memory allocators exist [4–11]. Also, there exists worst-case analysis of these algorithms from the temporal or spatial point of view [12–14]. The assumed distribution of block sizes, block lifetimes, and details of the testing procedure are relevant factors that can modify the results when comparing these algorithms. However, it is clear that if we try to use any of them for real-time applications, the following important aspects are not acceptable:

- Temporal cost: The worst-case execution of most allocators is very far from the average case. Binary buddy presents a bounded temporal cost that has been considered the most appropriate for real-time until now. Half-fit [15] was the first constant-time allocator, but the memory usage is very poor.
- Memory usage: Fragmentation is still an unsolved issue. From the point of view of real-time systems that have to operate over very long periods, fragmentation plays an important role in system degradation. Whereas some of the allocators achieve low fragmentation levels, others such as binary buddy have a very high fragmentation.

In a previous paper [16], the authors presented a new allocator called TLSF (two-level segregated fit) and compared it with other well-known allocators under different loads generated by well-known programs. These programs (e.g. gfrac, perl, gawc, etc.) were used by other authors for comparison purposes. In this paper, we focus on the criteria used to design and implement the TLSF allocator for real-time systems. The goal was to obtain an algorithm that allocates and deallocates in constant-time cost and maintains efficiency in time and space (low fragmentation). These two characteristics are essential for the analysis and implementation of real-time systems. In particular, the requirements for our algorithm are as follows.

Temporal requirements: The primary requirement for any real-time allocator is to provide allocator operations with constant 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. Although allocation operation involves a search for an appropriated block of memory, this search has to be bounded.

A second temporal requirement is efficiency: the algorithm has to be fast enough to compete with other well-known efficient allocators.

Spatial requirements: Traditionally, real-time systems run for long periods of time and some (embedded applications) have strong constraints of memory size. Fragmentation can have a significant impact on such systems. It can increase dramatically and degrade the system performance. The main requirement of our allocator is reduce fragmentation to a minimum.

Another spatial requirement is, given a memory pool, to guarantee that memory requests will always be satisfied. In order to guarantee this, the memory pool size has to consider the following aspects: the maximum amount of live memory used by the application, the data structures of the algorithm and the additional memory generated by memory fragmentation. The first of these is application dependent and beyond the scope of this paper. Static analysis or run-time profiling of the application code are possible techniques that can be used here.

The remainder of this paper is structured as follows. Section 2 briefly reviews the basic dynamic storage concepts and terms used in the paper and presents a review of the available allocators. Section 3 then summarizes the lessons that can be learned from the long history of dynamic memory research that allows us to focus our design for a real-time allocator. That design is discussed in

DOI: 10.1002/spe

Section 4, followed by a discussion of the implementation from a temporal and spatial point of view (in Section 5). Section 6 then presents an evaluation of the allocator operations. Section 7 details a workload model for real-time systems and performs a comparison of TLSF with other significant allocators. Finally, we conclude summarizing the major points of the paper and outlining future research directions.

2. DYNAMIC MEMORY ALLOCATION OVERVIEW

In 1995, Wilson *et al.* [3] presented a detailed survey of DSA that has been considered the main reference since then. The authors gave a comprehensive description, as well as the most important results, of all the problems related to memory allocation: the basic problem statement, fragmentation, taxonomy of allocators, coalescing, etc. In this section we extract the main issues that are relevant to understand the rest of the paper.

An *allocator* is an online algorithm that keep tracks of the memory status (areas of memory in use and free) by maintaining a pool of free blocks to satisfy the application memory requests. The allocator must respond to memory requests in strict sequence, immediately, and its decisions are irrevocable. The allocator does not know when the blocks will be allocated or freed. Once the allocator allocates a block, it is maintained in its place until it is freed.

After successive block allocations and deallocations, the memory space contains holes (*free blocks*). The effect of these free blocks is known as *fragmentation*. The goal of an allocator design is to find the appropriated free block for a request in a minimum time, minimizing wasted space (minimal fragmentation).

Allocators record the location and sizes of free blocks of memory using a data structure that may be a linear list ordered by physical addresses (address ordered (AO)) or block sizes, a totally or partially ordered tree, a bitmap, a segregated list, or some hybrid data structure.

The problem of finding a block to allocate the memory request is one of the major issues in the allocator. It depends on the global strategy (for instance, 'limit the number of operations involved maintaining low fragmentation'), the policy used (for instance, 'use the smallest block that satisfies the request') and the mechanism (for instance, 'segregated lists'). The different policies used are as follows.

First fit: The block used to serve the memory request is the first one found that satisfies the memory size required.

Next fit: Next-fit allocation differs from first fit in that a first-fit allocator commences its search for free space at a fixed end of memory, whereas a next-fit allocator commences its search wherever it previously stopped searching.

Best fit: This policy performs a detailed search to find the smallest block that satisfies the request. As several equally good fits can be found, the second criterion could be, for example, the one whose address is lowest.

Good fit: This policy allows the allocator to select a block that is not the best but is *near* the best. The term 'near' should be specified in terms of maximum block size difference with respect to the best. This kind of policy permits the allocator to apply search techniques that do not perform a complete search of the data structures, for example, using an *incomplete search* (see Section 4.3).

DOI: 10.1002/spe

2.1. Fragmentation and memory coalescing

Although the notion of fragmentation seems to be well understood, it is hard to define a single method of measuring it, or even an agreed definition of what fragmentation is. For the purpose of this paper, the definition given by Wilson *et al.* (in [3]) as 'the inability to reuse memory that is free' will suffice.

Historically, two different sources of fragmentation have been considered: *internal* and *external*. When the allocator serves a memory request, the free block used can be *split* into two blocks, one to serve the request and the other containing the remaining memory, which is added as a new free block. The splitting technique can deliver a block whose size is *exactly* or *greater than* the requested size. In the second case, there is some internal part of the block that will not be used (this is called *internal fragmentation*). In the first case, no internal fragmentation is generated. External fragmentation occurs when there is enough free memory but there is not a single block large enough to fulfil the request. Internal fragmentation is caused only by the allocator's implementation, while external fragmentation is caused by a combination of the allocation policy and the user request sequence.

When a block is freed, the allocator can *coalesce* (merge) it with neighbouring free blocks to maintain the availability of large blocks of memory. In some allocators, the coalescing process is not performed each time a deallocation occurs. The assumption is that a block of the same size will be requested (in the near future) for which this block will be candidate. These allocators avoid the merging and splitting work to increase the performance. However, in order to reduce the fragmentation, they will have to perform some coalescing process on the memory at a later time. This is known as *deferred coalescing*.

2.2. Description of allocators

A review of the most significant allocators is provided in this section. Allocators are organized considering the mechanism used. In some cases it is difficult to assign an allocator to a category because it uses more than one mechanism. In that case, the more relevant mechanism is used.

Sequential fits: Sequential fits algorithms are the most basic mechanisms. They search sequentially free blocks stored in a singly or doubly linked list. Typically, the pointers that implement the list are embedded inside the header of each free block (*boundary tag* technique [6]). Examples are first fit, next fit, and best fit.

First fit and best fit are two of the most representative sequential fit allocators, both are usually implemented with a doubly linked list.

Segregated free lists: These algorithms use a set of free lists. Each of these lists stores 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. It is important to remember that the blocks are logically but not physically segregated. There are two types of these mechanisms: simple segregated storage and segregated fits. An example of an allocator with this mechanism is fast fit [17], which uses an array for small-size free lists and a binary tree for larger sizes.

Buddy systems: Buddy systems [6] are a particular case of segregated free lists. If $\mathscr{H}$ is the heap size, there are only $\log_2(\mathscr{H})$ lists since the heap can be split only into powers of 2. This restriction yields efficient splitting and merging operations, but it also causes a high memory fragmentation.

DOI: 10.1002/spe

There exist several variants of this method [7], such as binary buddy, Fibonacci buddy, weighted buddy and double buddy.

The binary-buddy [6] allocator is the most representative of the buddy systems allocators, which has always been considered as a real-time allocator. The initial heap size has to be a power of 2. If a smaller block is needed, then any available block can be split only into two blocks of the same size, which are called *buddies*. When both buddies are again free, they are coalesced back into a single block. Only buddies are allowed to be coalesced. When a small block is requested and no free block of the requested size is available, a bigger free block is split one or more times until one of a suitable size is obtained.

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 that use Adelson-Velskii and Landin (AVL) trees [11], binary search trees or cartesian trees (fast fit [17]) 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. Half-fit [15] is a good example 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 are used to keep track of empty lists along with bitmap processor instructions to speed up search operations.

When a block of size r is required, the search for a suitable free block starts on i, where $i = \lfloor \log_2(r-1) \rfloor + 1$ (or 0 if $r = 1$). Note that the list i always holds blocks whose sizes are equal to or larger than the requested size. If this list is empty, then the next non-empty free list is used instead.

If the size of the selected free block is larger than the requested one, the block is split into two blocks of sizes r and r'. The remainder block of size r' is re-inserted into the list indexed by $i' = \lfloor \log_2(r') \rfloor$. The cost of this algorithm is constant ($O(1)$).

The process of avoiding an exhaustive search and only considering the sizes of the free blocks as a power of 2 causes what the author calls *incomplete memory use* [15].

Hybrid allocators: Hybrid allocators can use different mechanisms to improve certain characteristics (response time, fragmentation, etc.). The most representative is Doug Lea's allocator [8], which is a combination of several mechanisms. In what follows, this allocator will be referred to as DLmalloc.

DLmalloc implements a good fit jointly with some heuristics to speed up the operations as well as to reduce fragmentation[§].

Depending on the size of the free blocks, two different data structures are used. Blocks of size up to 256 are stored in a vector of 30 segregated lists. Each list contains blocks of the same size. Larger blocks are organized in a vector of 32 trees, which are segregated in power-of-2 ranges, with two equally spaced treebins for each power of 2. For each tree, its power-of-2 range is split into half at each node level, with the strictly smaller value as the left child. Same-sized chunks reside in a first-in first-out (FIFO) doubly linked list within the nodes.

Previous versions of DLmalloc used the delayed coalescing strategy, that is, the deallocation operation does not coalesce blocks. Instead, a massive coalescing was done when the allocator could not serve a request. The current version (2.8.3) performs an immediate coalescence. DLmalloc is considered one of the best and is widely used in many systems (glibc, eCos, etc.).

[§] A detailed description of the algorithm can be found in the comments of the code of the allocator [http://gee.cs.oswego.edu].

DOI: 10.1002/spe

Additionally, several custom allocators have been proposed [5,9,18]. They are designed considering the specific behaviour of the target application and can be tuned to improve time performance or optimize memory footprint. However, in [19] several custom allocators are evaluated and, in general, their performance is worse than DLmalloc's.

2.3. An illustrative simple example

In order to illustrate previous concepts, a naive allocator is described. The allocator is designed to serve memory size requests in the range of $[0 \ldots 8191]$ bytes. The mechanism used is a segregated list composed of an array of eight lists. Each list holds free blocks of a size class in the range of $[(1024*i) \ldots (1024*(i+1)-1)]$, where i is the list index. Figure 1 shows the data structure.

A request of r bytes implies a search in the linked list pointed by the array index computed by $\lfloor r/1024 \rfloor$. If this list is empty, the free block is searched in the next non-empty segregated list. For example, if the request is 4630 bytes, the searching function obtains a pointer position given by $\lfloor 4630/1024 \rfloor = 4$. This segregated list is empty; hence, the next non-empty list (index 5) is used. Several policies can be applied to find a free block in the list:

1. First fit: A sequential search is performed in the list until a block of size greater than or equal to the request is found. In the example, the block found is the first in the list (5120). The block is split into two blocks of 4630 and 1490 bytes. The first one is returned to the application and the second one, the remaining block, is inserted into the appropriated list.
2. Best fit: The sequential search tries to find the best block (the remaining block with smallest size). In this case the best block is 4650, and the remaining block has 20 bytes.

When a block is split to serve a request, the remainder size has to be inserted into the appropriated list. Also, when a block is freed, it has to be included in its segregated list. To insert a free block of size f in the data structure, we need to compute the function $\lfloor f/1024 \rfloor$ to obtain its segregated list. Following the previous example, if the block has 1490 bytes it has to be inserted in the segregated list indexed by $\lfloor 1490/1024 \rfloor = 1$. Blocks can be inserted into the head or tail of the list.

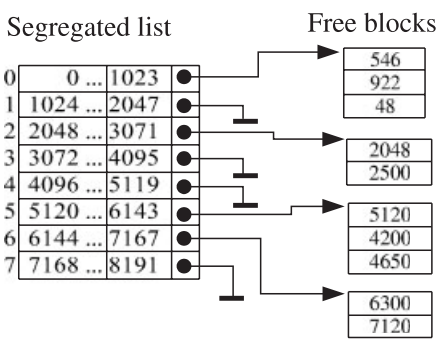

Figure 1. Simple example data structure.

DOI: 10.1002/spe

In this example, we have assumed that lists are not sorted. However, depending on the policy used, sorted lists can improve the performance of the search or insertion. Sort criteria can be block size, block logical address, etc.

Other aspects, such as coalescence, are not considered in this example.

3. DESIGN CRITERIA (LESSONS LEARNED)

In this section we summarize some conclusions from previous work to design a new allocator for real-time applications. We have split these conclusions into two aspects: those related to the allocation policies and the allocation mechanisms.

3.1. Allocation policies

The allocation policy has a strong influence on memory efficiency. The lessons learned from the literature can be used as starting point for the design of the TLSF allocator. In particular, we note the following results.

- AO first-fit policy has less worst-case external fragmentation than AO best fit [13].
- When using real workloads, the fragmentations caused by best-fit policies are slightly better than those produced by first-fit policies [20,21].
- Synthetic workload performance analysis gives no conclusive results about AO best and first policies. Depending on the parameters used in the workload generation, best fit may or may not outperform first fit. In all cases, the difference between allocators is quite small [22].
- Delayed coalescing improves the mean response time but slightly increases fragmentation [21].
- It is better to reallocate blocks that have been freed recently over those freed further in the past. In other words, LIFO policies are better than FIFO ones. Also, any variant of the best-fit policy or AO first-fit policy is much better than the next fit [21].
- The larger the amount of internal fragmentation, the smaller the external fragmentation. For example, on the one hand, the allocator half-fit does not suffer from internal fragmentation but, in the tests, it often has a high level of external fragmentation compared with the other allocators. On the other hand, Binary buddy's internal fragmentation can be up to 50%, but its observed external fragmentation is much smaller than that of other allocators. The worst-case fragmentation scenario for allocators with no internal fragmentation (like AO first fit and AO best fit) is several orders of magnitude the size of the live memory [7].
- With real workload, buddy systems have shown much higher average fragmentation than any basic policy (best, first or next fit) [20,21].
- Applications that allocate data structures tend to allocate only a few sizes (the sizes of the objects used in dynamic data structures like lists, trees, etc.). This is not the case for applications that handle strings or other variable size data objects [21].

In conclusion, an efficient allocator would do immediate coalescing, implement a best fit or close to best-fit policy and round-up memory request to reduce external fragmentation.

DOI: 10.1002/spe

3.2. Allocation mechanisms (data structures)

Whereas the spatial behaviour of an allocator is strongly determined by the allocation/deallocation policies employed, the temporal cost is mainly related to the mechanisms used in the implementation. A description of the mechanisms and data structures that have been used to implement best fit is summarized below:

- The simplest data structure to implement a best-fit policy is to use a doubly linked list. In the worst case, all elements of the list have to be visited on each memory request.
- A LIFO best-fit policy can be implemented using search trees (such as AVL, red–black or 3–4 search trees data structures). Although the asymptotic cost of these algorithms is $O(\log(n))$, where n is the number of sizes, the constant factors are not negligible, mostly because several tree operations (search, remove and/or insert) may be needed for a single allocation or release.
- Segregated list mechanisms consist of a set of free lists, where each list holds free blocks of a size or range of sizes. The more the segregated lists, the faster the search, because only those blocks that are close to the size requested are visited. Most of the best-performance allocators [8] use some variant of this mechanism.
- Buddy systems are a family of implementations that combine a set of segregated lists and a policy to split up and coalesce blocks; this allows it to perform those operations (split and coalesce) in constant time. The worst-case cost of both allocation and free is $O(\log(H))$, where H is the heap size.

Based on the above lessons, TLSF has been implemented using a set of segregated lists that can be directly reached through the size of the blocks. Also, TLSF does not implement a strict best-fit policy, but a good-fit one.

4. TLSF DESIGN

The goals of the TLSF allocator are to obtain a constant, and small, allocation cost (temporal requirement) and very low fragmentation of free memory (spatial requirement). This section discusses how these conflicting goals can be achieved using two levels of segregated lists, where each segregated list has an associated bitmap. The approach achieves a satisfactory compromise between temporal and spatial efficiency, allowing its use in real-time applications.

4.1. Temporal cost issues

The allocation of a requested memory block requires a search of the most appropriated memory-available (free) blocks. In order to implement a best-fit policy, an exhaustive search of the exact free blocks has to be done. Good-fit policies limit the search to find a free block near the best. In our case, we decided to use the good-fit policy to fulfil the constant time requirement.

The TLSF data structures are composed of a set of segregated lists. In general, in order to find a suitable block to serve a memory request, an algorithm based on segregated lists has to:

1. find the segregated list which holds blocks that can fulfil the memory request, and
2. if the segregated list holds a range of block sizes, search the list to find an appropriate fit.

DOI: 10.1002/spe

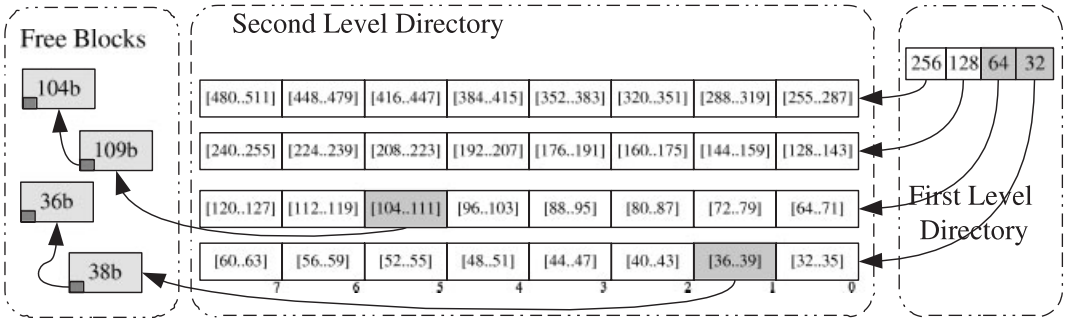

Figure 2. Segregated lists structure.

In TLSF, the range of sizes of the segregated lists has been chosen such that a mapping function can be used to locate the position of the segregated list given the block size, with no sequential or binary search. Also, ranges have been spread along the whole range of possible sizes in such a way that the relative width (the length) of the range is similar for small blocks as it is for large blocks. In other words, there are more lists used for smaller blocks than for larger blocks. Lists are arranged in a two-level tree, where the first-level directory splits sizes in power of 2 ranges, and the second-level sub-splits each first-level range linearly. Figure 2 shows a reduced version of the TLSF data structure with four segregated lists on the second level. This distribution of ranges provides the following benefits:

- Experimental results [21] show that most of the allocated blocks are of small sizes (data structures and strings).
- The fragmentation caused by this range distribution is independent of the requested sizes (see Section 4.3).
- The function which maps the size of a block to the indexes that point to the corresponding list can be computed efficiently (see TLSF functions on Section 5).

The first level is an array of four elements (pointers). Each element points to a segregated list of the second level. The second level is an array of eight elements. Each element points to the first component of a list of free blocks whose ranges match the element range. If a pointer is nil, this implies that there are no free blocks of this range in the system. For instance, in Figure 2, the third element (labelled as 128) in the first level is nil (white coloured). This means that there are no free blocks in the range [128 . . . 255]. However, the second element (64) is not nil (grey coloured). This indicates that there exist one or more free blocks in the range [64, 127]. In the second level, the second array (pointed by the second element (64) in the first level) has free blocks in the range [104, 111]. The corresponding element in this list (grey coloured) points to the first free block whose size is 109. This free block points to the next element in the list (104).

4.2. TLSF functions

To work with the data structure, several functions have been defined.

DOI: 10.1002/spe

The function *mapping_insert*(r) calculates the indexes (i, j) which point to *the list whose range contains blocks of size r*:

$$mapping\_insert(r) = \begin{cases} i = \lfloor \log_2(r) \rfloor \\ j = \left\lfloor \dfrac{(r - 2^i)}{(2^{i-\mathscr{J}})} \right\rfloor \end{cases}$$

where $\mathscr{J}$ is the $\log_2$ of the second level range. Values of $\mathscr{J}$ of 4 or 5, imply 16 or 32 lists, respectively, in the second level.

This mapping function is used by the free and malloc operations whenever a free block needs to be inserted into the TLSF structure. This function returns the indexes of the list where the free block has to be inserted.

The segregated list returned by the `mapping_insert` function may contain blocks that are smaller than r, and a search for a suitable block, from those stored in the given segregated list, has to be carried out. Rezaei and Cytron [23] proposed a segregated binary tree, where each segregated list is a binary tree. This solution will speed up the average search time, but in the worst case (all free blocks are located in the same list/tree and the tree gets unbalanced) the algorithm is almost unbounded.

The solution used in TLSF is to remove completely the search inside a segregated list. TLSF will look for a segregated list that holds blocks whose sizes are equal to or larger than the requested size. In this case, any block of the target segregated list can fulfil the request, and in particular the first block of the list can be used to serve it. This policy achieves constant time by using slightly larger blocks rather than the block that fits best. This is the main aspect that determines that TLSF is a *good-fit* allocator. The wasted memory caused by this policy is analysed later in this paper.

The function that returns the indexes of the list used to serve a memory request is

$$mapping\_search(r) = \begin{cases} i = \lfloor \log_2(r + 2^{\lfloor \log_2(r) \rfloor - \mathscr{J}} - 1) \rfloor \\ j = \left\lfloor \dfrac{(r + 2^{\lfloor \log_2(r) \rfloor - \mathscr{J}} - 1 - 2^i)}{(2^{i-\mathscr{J}})} \right\rfloor \end{cases}$$

The mapping function will always return the address of the smallest segregated list which contains blocks equal to or larger than the size requested. In case this list is empty, a search for the next non-empty segregated list has to be carried out. TLSF relies on a two-level bitmap data structure and special bit instructions to find the next non-empty list in constant time.

Figure 3 shows the complete data structure used. Each bit in a bitmap indicates whether the corresponding segregated list is empty or not. The hierarchical bitmap organization limits the number of bitmaps to be searched to 2. The number of segregated lists on each second-level node is customizable, which, due to implementation constraints, has to be a power of 2. Also, for efficiency reasons, the number of second-level lists, $2^{\mathscr{J}}$, should not exceed the number of bits of the underlying architecture. Therefore, $\mathscr{J}$ has to be no larger than 5.

For efficiency purposes, the tree structure can be directly transformed into a two-dimensional array as will be described in Section 5.

DOI: 10.1002/spe

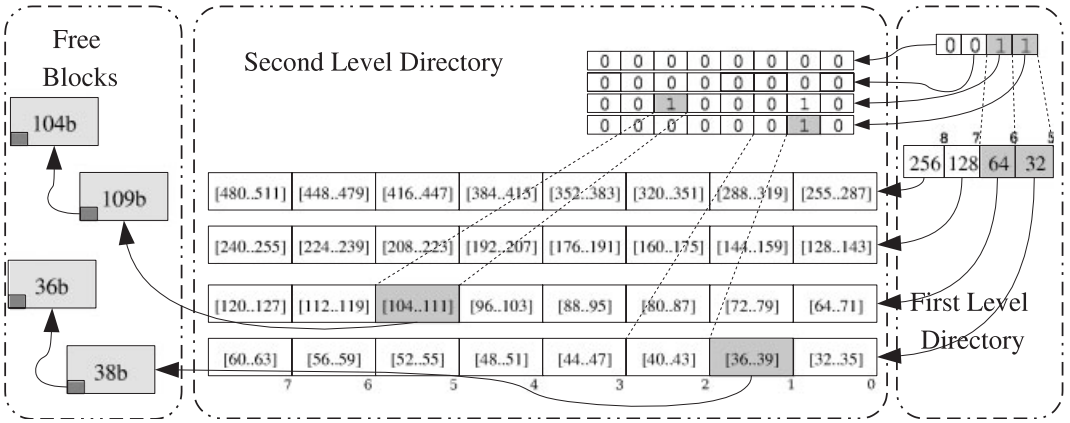

Figure 3. Bitmaps and segregated lists.

4.3. Spatial cost issues

Best-fit policies, or close to best-fit policies, are known to cause low fragmentation [21]. In this section, we analyse how the 'good-fit' allocation policy affects fragmentation and how the policy can be tuned to reduce fragmentation.

To serve an allocation request, TLSF will search for a list of free blocks that holds blocks that are certainly of size equal to or larger than the requested one. Once a target list has been found (position j), the first block of that list is used to serve the request.

It is possible that the predecessor list ($j-1$) contains free blocks that are large enough to serve the request. For example (see Figure 4), suppose that $\mathscr{J}$ is 3 and that the segregated list ($fl=10, sl=3$) (which holds blocks of sizes [1408, 1535]) contains a block of size 1524. TLSF is not able to use the existing free block to serve a request of size 1512. TLSF will use a block located on list $(10, 4)$ (range [1536, 1663]) or above.

Although, at first glance, the wasted memory caused by this *incomplete search* policy can be assumed as an acceptable price to be paid to achieve constant response time, a deeper analysis reveals that *incomplete search* can produce a large amount of wasted memory.

The problem is best illustrated by means of a numerical example. Suppose that the memory is that stated and represented in Figure 4. The application requests a block of size 1512. TLSF will start looking for a suitable block in list ($fl=10, sl=4$). Since that list is not empty, the block at the head of that list (block of size 1630) will be used to serve the request. At this point, there are two possible policies for splitting the block:

1. Assign to the application a block of exactly the same size as the requested one.
2. Round up the assigned size to the starting range of the list where a block of that size will be found. Following the example, a request of size 1512 would be rounded up to 1536, which is the starting size of the list pointed by ($fl=10, sl=4$).

DOI: 10.1002/spe

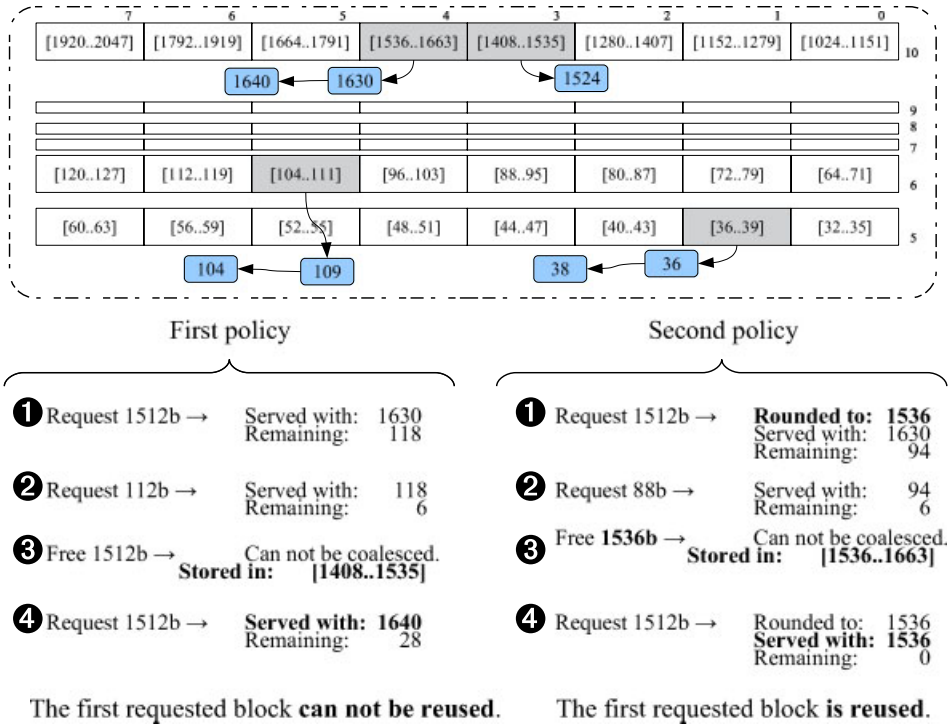

Figure 4. Structural fragmentation example.

An example of the first policy is presented in the left side of Figure 4. A similar trace sequence is sketched in the right side, but using the round-up split policy. The block of size 1630 is used in both cases to serve a request of size 1512. Using the first policy, the 1630 is split into two blocks: 1512 and 118. The first block is returned to the application, and the second one stored back as a free block. The second policy splits the 1660 bytes block differently: 1536 and 94. In step 2, the small remaining block is allocated by the application. Next, the application releases the first allocated block in step 3. Note that it cannot be coalesced because the neighbour was allocated in step 2; therefore, the first policy will store the block in the list pointed by $(10, 3)$ and the second policy will store the block in the list pointed by $(10, 4)$, since the freed block was rounded up to be in the range on the next list. Finally, if the application requests again a block of size 1512, then the exact split policy has to use a different large block to serve the request, but the round-up split policy is able to successfully reuse the first allocated block.

TLSF overcomes the problem caused by the incomplete search using the round-up split policy. This solution replaces the wasted memory caused by the incomplete search policy by internal fragmentation. Also, the more the segregated lists (the bigger the $\mathcal{J}$ parameter), the smaller the distance between ranges, and then, accordingly, the amount of rounded-up memory. In Section 5.3 this internal fragmentation is analysed in detail.

DOI: 10.1002/spe

5. TLSF IMPLEMENTATION

As explained in the previous section, TLSF was designed to be a compromise between constant and fast response time and efficient memory use (low fragmentation).

The TLSF tree structure can be implemented efficiently using a two-dimensional array, where the first dimension (first-level directory) splits free blocks into size ranges that are a power of 2 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 into a number of ranges of equal width. The number of such ranges, $2^{\mathscr{J}}$, 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. A good balance between temporal cost and memory efficiency is obtained for values of $\mathscr{J}=4$, or $\mathscr{J}=5$ for a 32-bit processor. Figure 5 illustrates the data structure for $\mathscr{J}=3$. This figure will be used in the examples of this section.

The mathematical function $\lfloor \log_2(x) \rfloor$ can be computed very fast by finding the index of the most significant bit with the processor instruction *fls*[¶]. Another bitmap function that is commonly available on modern processors is *ffs*[‖]. Note that it is not mandatory to have these advanced bit operations available in the processor to achieve constant time, since it is possible to implement them by software using less than six non-nested conditional blocks (see glibc or Linux implementation in Appendix A).

The function `mapping_insert` computes efficiently fl and sl:

```
1  procedure mapping_insert (r: integer; fl, sl: out integer) is
2  begin
3     fl:=fls(r);
4     sl:=(r right_shift (fl − J)) − 2^J ;
5  end mapping_insert;
```

For example, given the size $r=74$, the first-level index is $fl=6$ and the second-level index is $sl=1$. It is straightforward to obtain these values from the binary representation of the size:

$$\mathtt{r}=74_d=00000000\ 01001010_b=\overset{15\,14\,13\,12\,11\,10\,9\,7\,\mathbf{6}\ \ 5\,4\,3\ \ 2\,1\,0}{0\,0\,0\,0\,0\,0\,0\,0\,1\ \underbrace{0\,0\,1}_{\mathtt{sl=1}}\ 0\,1\,0_b}\ \ \mathtt{fl=6}$$

The list indexed by $fl=6$ and $sl=1$ is where blocks of sizes in the range [72...79] are located.

The function `mapping_search` computes the values of fl and sl used as starting point to search for a free block. Note that this function also rounds up (line 3) the requested size to the next list (see Section 4.1).

[¶] *fls*: Find last set. Returns the position of the most significant bit set to 1.
[‖] *ffs*: Find first set. Returns the position of the first (least significant) bit set to 1.

DOI: 10.1002/spe

```
1  procedure mapping_search (r: in out integer; fl, sl: out integer) is
2  begin
3    r:=r+ (1 left_shift (fls(r) − 𝒻))− 1;
4    fl:=fls(r);
5    sl:=(r right_shift (fl − 𝒻)) − 2^𝒻;
6  end mapping_search;
```

A call to `mapping_search` for a size $r=74$ returns the values of $fl=6$ and $sl=2$, which points to the list that holds blocks in the range [80…87].

The function `find_suitable_block` finds a non-empty list that holds blocks larger than or equal to the one pointed by the indexes fl and sl. This search function traverses the data structure from right to left in the second-level indexes and then upwards in the first level, until it finds the first non-empty list. Again, the use of bit instructions allows implementation of the search in fast and constant time.

```
1  function find_suitable_block (fl, sl: in integer) return address is
2  begin
3    bitmap_tmp:= SL_bitmaps[fl] and (FFFFFFFF#16# left_shift sl);
4    if bitmap_tmp ≠ 0 then
5      non_empty_sl:= ffs(bitmap_tmp);
6      non_empty_fl:= fl;
7    else
8      bitmap_tmp:= FL_bitmap and (FFFFFFFF#16# left_shift (fl+1));
9      non_empty_fl:= ffs(bitmap_tmp);
10     non_empty_sl:= ffs(SL_bitmaps[non_empty_fl]);
11   end if;
12   return head_list(non_empty_fl, non empty_sl);
13 end find_suitable_block;
```

By following the example, the returned free block is the one pointed by the list $(6,5)$ which holds blocks of sizes [104…111].

Allocation procedure

```
1  function malloc (r: in integer) return address is
2  begin
3    mapping_search(r, fl, sl);
4    free_block:= find_suitable_block(r, fl, sl);
5    if not(free_block) then return error; end if;
6    remove_head(free_block);
7    if size(free_block)−r {>} split_size_threshold then
8      remaining_block:= split(free_block, r);
9      mapping_insert(size(remaining_block), fl, sl);
10     insert_block(remaining_block, fl, sl);
11   end if;
12   return free_block;
13 end malloc;
```


DOI: 10.1002/spe

The code of the `malloc` function is almost self-explanatory (numbers at the start of the following list refer to lines in the code):

3. Compute the indexes of the list which holds blocks of size equal to or larger than the requested size.
4. Starting from the obtained list, find a non-empty free list.
6. Extract the block at the head of that list.
7. If the result of the size of the found block minus that of the requested block is larger than a predefined threshold (this threshold can be defined as the minimum block size that is worth being managed by TLSF), then:

    8. Round up and split the block: create a block header on the remaining free memory and update the header of the original block.
    9. Compute the indexes of the list whose range contains the size of the remaining block.
    10. Insert the remaining block in the list pointed by these indexes.

12. Return the block.

The `remove_head` and `insert_block` functions extract and insert an element from the head of a list, respectively, and update the corresponding bitmaps. These operations work in constant time.

Operations for coalescing blocks

```
1   function merge_prev (block) return address is
2    if is_prev_free(block) then
3     prev_block := prev_physical(block);
4     mapping_insert(prev_block, fl, sl);
5     remove_block(prev_block, fl, sl);
6     merge(prev_block, block);
7    end if;
8    return prev_block;
9   end merge_prev;
10
11  function merge_next (block) return address is
12   if is_next_free(block) then
13    next_block := next_physical(block);
14    mapping_insert(next_block, fl, sl);
15    remove_block(next_block, fl, sl);
16    merge(block, next_block);
17   end if;
18   return block;
19  end merge_next;
```

The `free` function always tries to coalesce neighbour blocks. `merge_prev` checks whether the previous physical block is free; if so, it is removed from the segregated list and coalesced with

DOI: 10.1002/spe

the block being freed. `merge_next` does the same operation, but with the next physical block. Physical neighbours are quickly located using the size of the free block (to locate the next block) and a pointer to the previous one, which is stored in the head of the freed block. The `remove_block` function removes a given free block from a doubly linked list. In order to update head pointers and bitmaps, this function needs to know which is the head of the segregated list where the removed block belongs to. Therefore, the `mapping_insert` function has to be called to compute the indexes of the corresponding segregated list. Merging two adjacent free blocks can be accomplished by incrementing the size field of the first block (the block with smaller physical address).

Deallocation procedure

```
1  procedure free (block: in address) is
2  begin
3    merged_block:= merge_prev(block);
4    merged_block:= merge_next(merged_block);
5    mapping_insert(size(merged_block), fl, sl);
6    insert_block(merged_block, fl, sl);
7  end free;
```

In order to implement the doubly linked list of segregated blocks, two pointers are needed for each free block. Since only free blocks are linked in a segregated list (note that allocated blocks are not handled by the allocator), these pointers are located inside the free block itself. Neighbour blocks are reached using the boundary tag mechanism [6]. Therefore, allocated blocks have a header that contains only the size of the block (4 bytes in a 32-bit processor), and free blocks have to be large enough to contain four words (12 bytes): the size at the head and replicated at the bottom (boundary tag mechanism), and two pointers for the doubly linked list.

5.1. A note on the actual implementation of the TLSF

As explained in Section 4, TLSF sub-divides the first level ranges linearly. The number of sub-ranges is customizable. This structure works as expected for all block sizes but small ones. The problem is that there are more free lists than possible ranges for small blocks. For example (suppose $\mathscr{J}=5$): the number of different sizes in the first level for $i=4$ is $2^4=16$, but the number of lists allocated in the second level is 32. Therefore, some list ranges overlap.

To overcome this problem, the actual implementation of the TLSF handles small blocks as a special case. Blocks smaller than 128 bytes are stored in a vector of lists, and lists for larger sizes are arranged using the already described two-level lists structure. This change only affects the mapping functions (`mapping_search` and `mapping_insert`) that have to compute fl and sl differently depending on the size. All the data structures, matrix of lists and bitmaps, remain unchanged.

5.2. Temporal cost analysis

It is straightforward to obtain the asymptotic cost of the TLSF from its implementation.

Although `malloc` has to perform a search of the TLSF data structure (`search_suitable_block`), its asymptotic cost is $O(1)$ due to the use of bitmap search functions. `remove_head`

DOI: 10.1002/spe

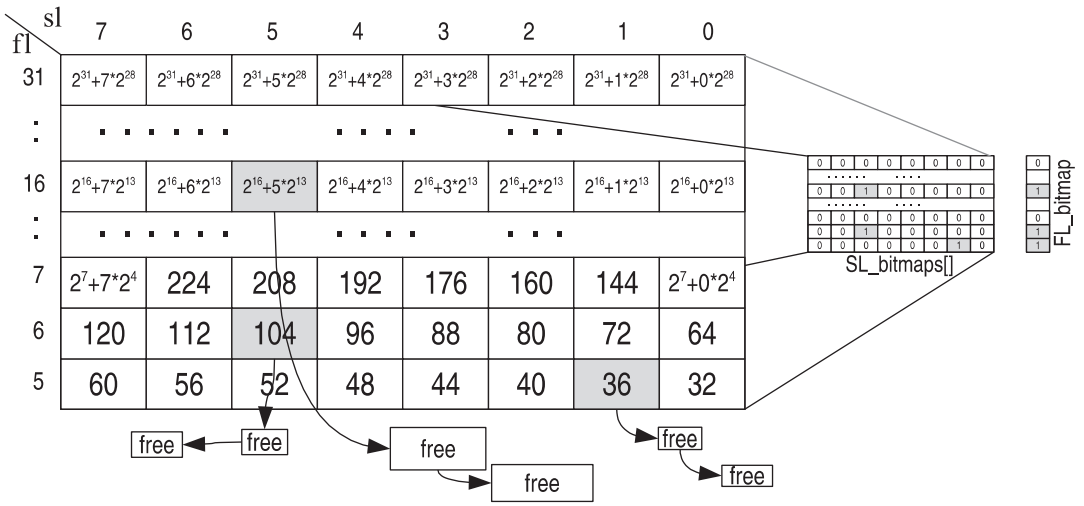

Figure 5. TLSF data structures example.

function simply unlinks the block from the segregated fit list, and `insert_block` function inserts at the head of the corresponding segregated list.

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

All internal operations used from `malloc` and `free` have constant times and there is no loop; therefore, the asymptotic worst-case response time is

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

5.3. Fragmentation cost analysis

There are two sources of wasted memory: internal and external fragmentation.

As analysed in Section 4, the wasted memory caused by the *incomplete search* policy is minimized with both the round-up split policy and a large number of segregated lists. The round-up split policy converts the *incomplete search* wasted memory into internal memory. For example, a request of size 1512 bytes is rounded up to 1536 bytes as shown in Figure 4.

The worst-case internal fragmentation occurs when the requested size is 1 byte bigger than an existing segregated list ($r = 2^i + 1$), and has to be rounded up to the next list ($r_{rounded} = 2^i + 2^{i-\mathscr{J}}$). In the TLSF this can be calculated as follows:

$$\text{Int\_block\_frag}(r) := \overbrace{(2^{\text{fl}(r)} + 2^{\text{fl}(r)-\mathscr{J}} \cdot \text{sl}(r))}^{\text{Allocated}} - \overbrace{r}^{\text{Requested}}$$

DOI: 10.1002/spe

The resulting worst-case internal fragmentation is bounded by

$$\text{internal\_fragmentation}_{\text{TLSF}}^{\max} \leq \frac{1}{1+2^{\mathscr{J}}}$$

This gives a worst-case internal fragmentation that is around 3.1% of the requested size for a value of $\mathscr{J}=5$ and 6.2% for $\mathscr{J}=4$.

Note that wasted memory due to internal fragmentation and incomplete memory usage cannot occur at the same time on the same block. If a block has internal fragmentation, it is because it is already allocated, and a block can cause incomplete memory usage only if it is a free block. Therefore, the overall 'non-external' fragmentation of the TLSF is 3.1%. A more detailed analysis of the fragmentation can be found in [24].

External fragmentation occurs when there is enough free memory but there is no single block large enough to fulfil a 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. Robson [12,13,25] analysed the worst-case memory requirements for several well-known allocation policies. Robson designed allocation sequences that force each policy to exhibit its maximum external fragmentation. If the maximum allocated memory (live memory) is $\mathscr{M}$ and the largest allocated block is m, then the heap size needed for a best-fit policy in the worst case is bounded by $\mathscr{M}(m-2)$.

Most of the initial fragmentation studies [22,26] were 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. Zorn and Grunwald [27] investigated the accuracy of simple synthetic workload models and concluded that synthetic workloads should not be used in the general case because they do not reproduce properly the behaviour of real workloads.

Johnstone and Wilson [21] 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. Among other observations, Johnstone and Wilson pointed out that low-fragmentation allocators are those that perform immediate coalescing, implement a best-fit or good-fit policy and try to relocate blocks that have been released recently over those that were released further in the past.

A comparative analysis of the fragmentation incurred by TLSF and other relevant allocators is provided in Section 7.

6. TLSF EVALUATION

In order to evaluate the proposed allocator, we have performed two temporal analyses:

- *Worst-case scenario test*: The allocator is analysed in a worst-case scenario in order to determine the WCET. This analysis can be summarized in two steps: firstly, building two worst-case synthetic loads (allocation/deallocation) for our allocator; secondly, measuring the executed number of instructions when both synthetic loads are run.

DOI: 10.1002/spe

- *Analysis of the TLSF execution*: The allocator is instrumented and fed with a random load. The goal is to analyse the behaviour of the allocator regarding the complexity of the internal operations.

It is difficult to perform a spatial analysis without a comparison with other allocators. For this reason, the spatial analysis is performed in the next section when comparing temporal and spatial measures with other allocators under real-time workloads.

6.1. Worst-case analysis

The worst-case scenarios for allocation and deallocation are as follows.

Allocation: Since this operation does not depend on the number of free or busy blocks and there are no loops in the implementation, only small variations in the execution time can be observed depending on the conditional code executed. The worst case for allocation occurs when there is only one large free block and the application requests a small block. The asymptotic cost is $O(1)$.

Deallocation: The timing cost of this operation depends on the number of times that the released block has to be merged with other free blocks. There are three possible cases: (1) no free neighbours; (2) one neighbour; and (3) two neighbours. The worst case is when the two neighbours are free so that the allocator has to coalesce them. The cost is $O(1)$.

In these scenarios, the measured number of instructions executed by the `malloc` and `free` operations is 160 and 176, respectively. It is important to note that these results can slightly change depending on the compiler version and the optimization options used.

6.2. Detailed execution analysis

For this test, the code of TLSF has been instrumented to measure the number of machine code instructions required by each step of the algorithm on an x86 box. All the tests have been compiled using GCC with the flag '-O2' set. We have used the Linux *ptrace* system call to trace the process that runs the allocator (traced process). The traced process reads the allocation/deallocation sequence from a file and makes the corresponding allocation/deallocation calls. The allocator is executed in a single-step mode, thus being stopped after each executed instruction. By using a shared memory area, the traced process can inform the tracer (writing a single integer in a shared variable) of which block of code is being executed. It is important to note that the measures obtained by this method differ from the previous one because this introduces an overhead when the tracer marks the code being executed.

Although less intrusive methods could be used (for example, using the address of the instruction being executed to keep track of which part of the allocator is being executed), the optimizations performed by the compiler, i.e. reordering code, make it difficult to determine which 'C' code belongs to which instructions.

Figures 6(a) and (b) show the number of instructions executed by `malloc` and `free` under a random workload.

Several bands, groups of mallocs performing the allocation in the same number of instructions, can be identified. The first two bands of Figure 6(a) (105 and 109 instructions) correspond to

DOI: 10.1002/spe

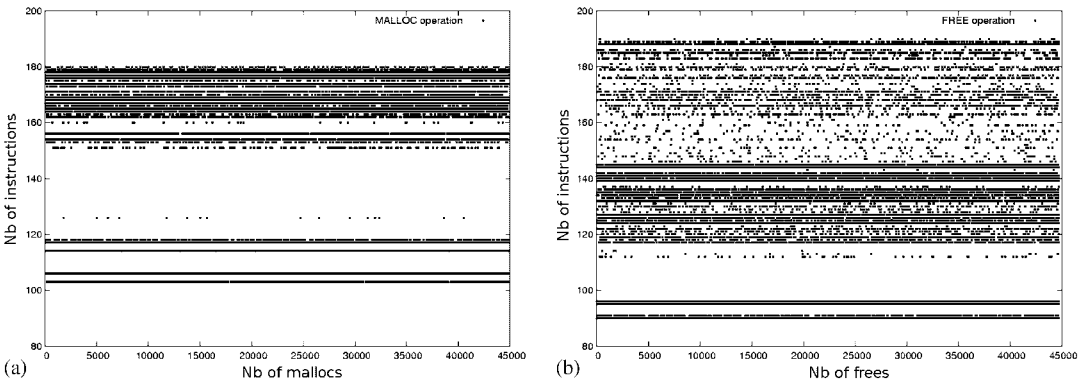

Figure 6. Instruction bands of (a) malloc and (b) free operations.

requests of sizes smaller than 128 bytes served with free blocks of the same size as the requested one. In this case, the function `mapping_search` is faster because it only has to search the first-level bitmap (see Section 5.1), and no division and re-insertion of the remaining block have been carried out.

The difference between the 105 and the 109 bands is due to the cost of `remove_head`, which requires four instructions more for bitmap updates if the removed block is the last block of the segregated list.

Table I summarizes the exact number of instructions required to perform each internal operation.

As can be seen in Figure 6(b), the `free` operation shows three different behaviours: if the freed block cannot be coalesced, it requires 90–96 instructions; if the block has to be coalesced with one neighbour it requires 112 to 146 instructions; and if the freed block has to be coalesced with both neighbours, then it takes between 143 and 190 instructions.

Table II details the number of instructions measured for each internal operation. The sum of the costs of each operation produces the final dispersion of the `free` operation.

7. EXPERIMENTAL EVALUATION

In this section, we compare the performance of TLSF with respect to other allocators. In [16], TLSF** was compared with several well-known allocators such as first fit (sequential fits), best fit (sequential fits), binary buddy (buddy systems), DLmalloc†† (hybrid algorithm using segregated and sequential system) and half-fit under real loads produced by typical programs such as compilers (gcc, perl, etc.) and application programs (cfrac, espresso, etc.). In this paper, we compare TLSF

**The version used in that comparison was TLSF 1.2.

††The version used in that comparison was Dlmaloc 2.7.2.

DOI: 10.1002/spe

Table I. Costs of internal functions for `malloc`.

Operation	Instruction	Comment
`mapping_search`	11	Requested block is smaller than 128 bytes → search on one bitmap
	21	Requested block is larger than 128 bytes → search on two bitmaps
`find_suitable_ block`	20	A free block of the searched range exists
	29	The first suitable segregated list was empty
`remove_head`	17	It was not the last one of its segregated list → no bitmap update
	21	It was the last block of its segregated list → one bitmap update
	24	The last block in its power of two range → two bitmap updates
`split`	6	Constant time
`mapping_insert`	10	Inserted block is smaller than 128 bytes → search on one bitmap
	15	Inserted block is larger than 128 bytes → search on two bitmaps
`insert_block`	22	Insert on non-empty list → update linked list pointers
	24	Insert on empty list → update only the head

Table II. Costs of internal functions for `free` operation.

Operation	Instruction	Comment
`next_physical`	3	Add to the current block address its size
`prev_physical`	1	There is a pointer to the previous block
`mapping_insert`	6	Requested block is smaller than 128 bytes → search on one bitmap
	11	Requested block is larger than 128 bytes → search on two bitmaps
`remove_block`	19, 21, 23, 27, 29, 36, 38, 39	There are several issues that affect the cost of this operation: (1) if it is not the last block of the list → update previous pointer; (2) if it is in the head → update the head pointer; (3) if it is the last block of the list → update the bitmap; and (4) if it is the last block of its power of two range → update the first-level bitmap. The combination of previous conditions produces the eight different observed values
`merge`	7	Increment the size field of a block
`insert_block`	22	Insert on non-empty list → update linked list pointers
	23	Insert on empty list → update only the head

DOI: 10.1002/spe

with the most efficient reported allocators. The compared allocators are as follows:

- *Binary buddy*: This allocator has been used in some real-time applications.
- *DLmalloc*: Currently, this allocator is considered to be one of the best existing allocators. It is widely used in general-purpose systems.
- *Half-fit*: This was the first published allocator with constant timing cost.

Whereas the codes of binary buddy and half-fit have been written from scratch, DLmalloc has been freely downloaded from the author's web site. The version used in this evaluation is 2.8.3. With respect to TLFS, the version used in this evaluation is 2.2, which corresponds to the description given in this paper.

7.1. Load model

To the best of our knowledge, there is not a memory model for real-time periodic threads except the model proposed in the real-time specification for Java [28]. In this model, a real-time thread can define its memory requirements specifying a limit on the amount of memory a thread may allocate in its initial memory area, a limit on the amount of memory a thread may allocate in the immortal area and a limit on the allocation rate in the heap. The real-time Java memory model has been designed around the needs of the garbage collector. That is, to say, the activation period of the garbage collector can be calculated from the task parameters.

In order to perform an evaluation of the considered allocators under the proposed model, a synthetic load generator has been designed. Although synthetic workloads should be avoided because they are not able to grasp the fundamental behaviour of real applications (see Section 5.3), this kind of workload is widely used by the real-time community [29]. Also, considering the lack of real-time applications that use dynamic memory, we were forced to use synthetic workloads in our experiments. The definition of a workload benchmark for real-time applications remains an open issue that deserves further research.

The proposed load model [30] considers that each periodic task T_i defines g_i as the maximum amount of memory a task can request per period, and h_i as the longest time a task can hold a block after its allocation (holding time). In other words, a task T_i can ask for a maximum of g_i bytes per period, which have to be released not later than h_i units of time. In order to serve all memory requests, the system has an area of free memory (also known as heap) of $\mathscr{H}$ bytes.

7.2. Load generator

The proposed workload model generates task sets under the following premises:

1. The number of tasks is randomly generated from a uniform distribution range.
2. The periods (p_i) are randomly generated using a uniform distribution that is defined by minimum and maximum admissible period values.
3. The maximum number of memory requests (N_r) per period follows a uniform distribution in a specified range.
4. The maximum amount of memory requested by period (g_i) is randomly generated using a uniform distribution specified by minimum and maximum block sizes.

DOI: 10.1002/spe

Table III. A task set example of profile 1.

	p_i	g_i	G_{avg}	G_{sdv}	H_{max}	H_{min}
T_1	60	23 657	18 198	1000	35	22
T_2	96	54 126	41 636	1375	26	25
T_3	98	44 996	34 613	2500	25	18
T_4	120	55 995	13 998	1125	29	21
T_5	150	34 457	8614	2000	32	14

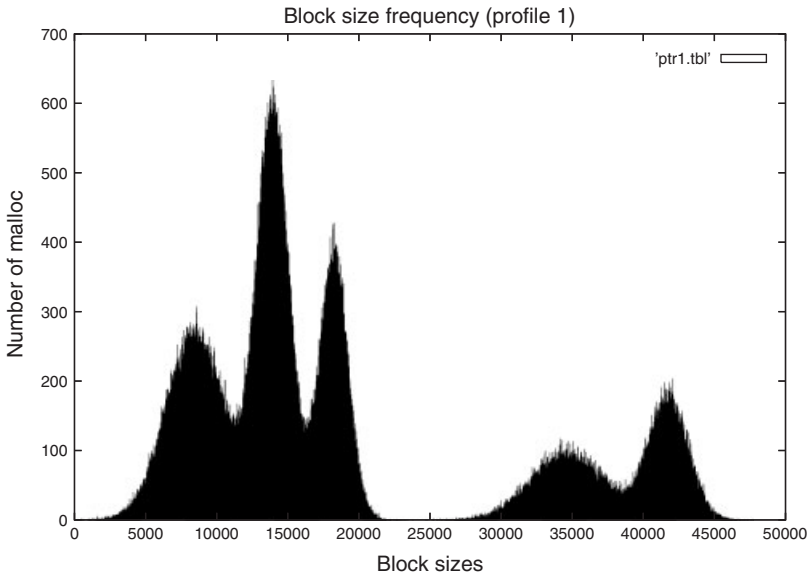

Figure 7. Block size histogram of the requests generated by the task set of Table III.

5. Each memory request follows a normal distribution with an average value ($G_{avg} = g_i / N_r$) and a standard deviation calculated as a constant factor of the block size.
6. The holding time (h_i) is determined using a uniform distribution with minimum (H_{min}) and maximum (H_{max}) admissible holding time values.

Three different application profiles (task sets) have been considered:

Profile 1: A set of tasks that allocate high amount of memory per period by requesting large blocks. The load consists of a set containing between 3 and 10 tasks with periods in the range [20...150]. The maximum amount of memory, g_i, requested per period by each task is in the range [8...64 kb]. We assume that the number of requests N_r performed by a task during its activation period is in the range [2...5]. The holding time, h_i, of each request has also been calculated with a uniform distribution defined in the range [30...50]. An example of a task set generated from this profile is detailed in Table III, which generates the block size requests shown in the histogram plotted in Figure 7.

DOI: 10.1002/spe

Profile 2: This profile is configured to evaluate the behaviour of the allocators when only small blocks are requested. All the parameters of the profile are the same as in the previous one, except the range considered for the g_i parameter, which here is set to [64 b . . . 8 kb].

Profile 3: The third load pattern tries to cover both previous cases. Now the maximum amount of memory g_i requested during a period can vary in the range [64 b . . . 48 kb].

7.3. Experimental evaluation of the allocators

In this section we present the outcomes obtained by the selected allocators under the workloads generated by the proposed profiles.

For each profile, 100 different task sets have been generated. Each task set performs until 10^6 mallocs (the number of free operations is a bit lower due to the holding time effect). Considering all the tests carried out, the total number of malloc operations executed is 3×10^8.

Depending on the characteristic to be compared, the code of the simulator has been instrumented to measure: (1) the execution time required to perform each malloc and free, which is measured in processor cycles; (2) processor instructions, to obtain a measure of the cost of the algorithm, which is independent of the system interferences; and (3) the amount of fragmentation.

For each test, the average, the standard deviation, the maximum and minimum number of processor cycles or instructions are obtained. Tables IV–VI present a statistical summary of all tests performed for each profile.

In both the first tables (cycles and instructions), we have measured individual malloc and free operations. Consequently, *Avg* represents the average of all malloc and free operations carried out in all tests in a profile (100 tests, 10^6 mallocs or free operations of each test). *Std* shows the standard deviation of these measures. *Max* and *Min* indicate the absolute maximum and minimum measures observed, respectively.

In the fragmentation table, we are interested in the fragmentation at the end of a test. In this case, we have 100 fragmentation measures. *Avg* shows the average fragmentation in these 100 tests. *Std* shows the standard deviation of these measures. *Max* and *Min* indicate, respectively, the absolute maximum and minimum fragmentation observed in a profile evaluation.

Table IV. Temporal cost of the allocator operations measured in processor cycles: (a) malloc and (b) free.

	Profile 1				Profile 2				Profile 3			
Allocator	Avg	Std	Max	Min	Avg	Std	Max	Min	Avg	Std	Max	Min
(a)												
BB	165	42	3252	126	145	31	2817	99	167	49	3558	102
DL	316	79	3573	99	265	94	2592	72	365	96	2877	72
HF	210	45	816	126	151	30	783	108	213	47	699	111
TLSF	194	40	651	123	176	34	957	111	210	48	663	111
(b)												
BB	200	116	2490	93	164	89	2256	93	314	155	3087	93
DL	220	97	2295	84	181	108	2184	75	235	116	2247	81
HF	182	57	1401	93	170	45	1461	93	197	63	1266	93
TLSF	192	55	1833	111	179	53	1722	102	204	64	1908	102

DOI: 10.1002/spe

Table V. Temporal cost of the allocator operations in processor instructions: (a) malloc and (b) free.

	Profile 1				Profile 2				Profile 3			
Allocator	Avg	Std	Max	Min	Avg	Std	Max	Min	Avg	Std	Max	Min
(a)												
BB	175	19	653	155	146	28	819	106	171	27	2574	106
DL	278	43	473	98	214	80	453	58	284	57	489	58
HF	114	2	118	112	113	3	118	78	114	2	118	62
TLSF	138	13	151	102	125	20	151	89	135	14	151	89
(b)												
BB	68	21	402	66	68	30	317	66	68	29	321	35
DL	173	56	341	68	144	89	318	68	172	69	347	68
HF	106	19	144	72	106	19	144	72	106	19	144	72
TLSF	109	23	159	83	99	27	159	80	107	24	159	80

Table VI. Fragmentation results (in %): factor $\mathscr{F}$.

	Profile 1				Profile 2				Profile 3			
Allocator	Avg	Std	Max	Min	Avg	Std	Max	Min	Avg	Std	Max	Min
BB	53.8	8.0	76.3	36.7	48.6	7.8	78.4	31.3	52.3	6.3	68.6	39.7
DL	8.9	1.5	13.1	5.8	8.9	1.7	16.6	6.1	8.5	1.9	15.4	4.8
HF	83.4	13.4	120.2	52.7	85.0	12.1	115.0	59.3	89.2	13.9	143.4	62.3
TLSF	9.9	1.5	14.3	5.9	9.7	1.7	16.1	6.3	9.6	1.9	16.4	6.7

When a test is executed, each allocator is initialized with a free block of size 16 Mb. The DLmalloc allocator is initialized by allocating a block of this size and then releasing it. This way, the allocator will have enough memory to run the whole test without requesting more memory from the operating system (via `sbrk` or `mmap` system calls). Note that we are interested in allocators that rely neither on virtual memory nor on operating system facilities.

All measurements were obtained on an Intel® Pentium (Celeron) 1600 MHz with 512 Mb of main memory and 1024 kb cache memory: GCC 3.4.6 compiler with '-O2-fomit-frame-pointer' flags. Processor caches are not disabled, although they were invalidated on the worst-case tests.

7.3.1. Execution time

Execution time is measured as the number of processor cycles needed for both allocation and deallocation operations.

Each test is executed with interrupts disabled. However, there are still some factors (e.g. cache faults, TLB misses, etc.) that can produce significant variations in the execution. To reduce these

DOI: 10.1002/spe

effects, each test has been executed three times (replicas) using the same workload and selecting the minimum number of cycles incurred by each individual malloc or free operation in these replicas (voting system). We have observed that increasing the number of replicas does not decrease significantly the quality of the measurement.

Table IV shows a summary of the processor cycles spent for each allocator for both operations: `malloc` and `free`.

Although the average values of all of the allocators have similar results, with low deviation in all tests, the worst case (maximum value) of half-fit and TLSF are significantly lower than those of others.

Although the half-fit's data structure and its algorithm are simpler than those of TLSF (note that half-fit can be considered as a reduced version of TLSF, with just one level of segregated lists), the average response of both allocators is fairly similar. This is due to the round-up policy of the TLSF, which reduces the chances of having to split free blocks. Splitting a block is a costly operation because it involves an insertion in the data structure of the remaining memory. This is observed in profile 2, where most of the blocks are small and no split operations are performed, giving half-fit lower cycles than TLSF.

In general, the behaviour of the four allocators is quite similar. All of them achieve better results for small blocks than for large blocks. When small-size blocks are requested (profile 2) all of them require a lower number of cycles than are required for the other profiles. This is due to the different data structures and policies used for large and small blocks (DLmalloc, TLSF and half-fit). This is not the case with the binary-buddy allocator. Also, the combination of small and large block sizes produces a small increase in the number of cycles with respect to profiles 1 and 2. The main reason of this result is due to the wide range of sizes requested, which reduces the possibility of free blocks reutilization and increases the number of coalesced blocks.

In order to see in more detail these results, Figure 8 shows the histogram of all malloc operations executed in all tests for profile 1. The x-axis plot range is $[50 \ldots 600]$ (minimum value obtained was 99 by DLmalloc allocator) and, although there are measurements higher than 600, these have been accumulated at the end of the histogram ($x = 599$). The y-axis represents multiples of 10^6 number of mallocs.

Analysing these plots and Table IV, we can conclude the following:

- Half-fit's plot shows three peaks that can be associated with the three main execution paths in the algorithm. Small variations around these peaks could correspond to minor system interferences such as clock granularity, TLB misses, etc., not completely filtered by the voting system used for this test.
- Binary buddy presents two different situations. Bottom part (area lower than 4 in the y-axis) corresponds to the tree traverse and block split with the above-mentioned interferences, and the peaks can be associated with block reutilization (neither block merging nor splitting is required).
- Dlmalloc's plot shows the most uniform behaviour. It can be interpreted as the result of performing an exact search in the bintree each time a large (>256) block is required in addition to the heuristic applied and the already-mentioned system interferences.
- TLSF presents a behaviour that matches the detailed analysis carried out in Section 6.2. Bands described in that analysis are displayed as peaks in the plot. Again, system interferences produce a flattened response.

DOI: 10.1002/spe

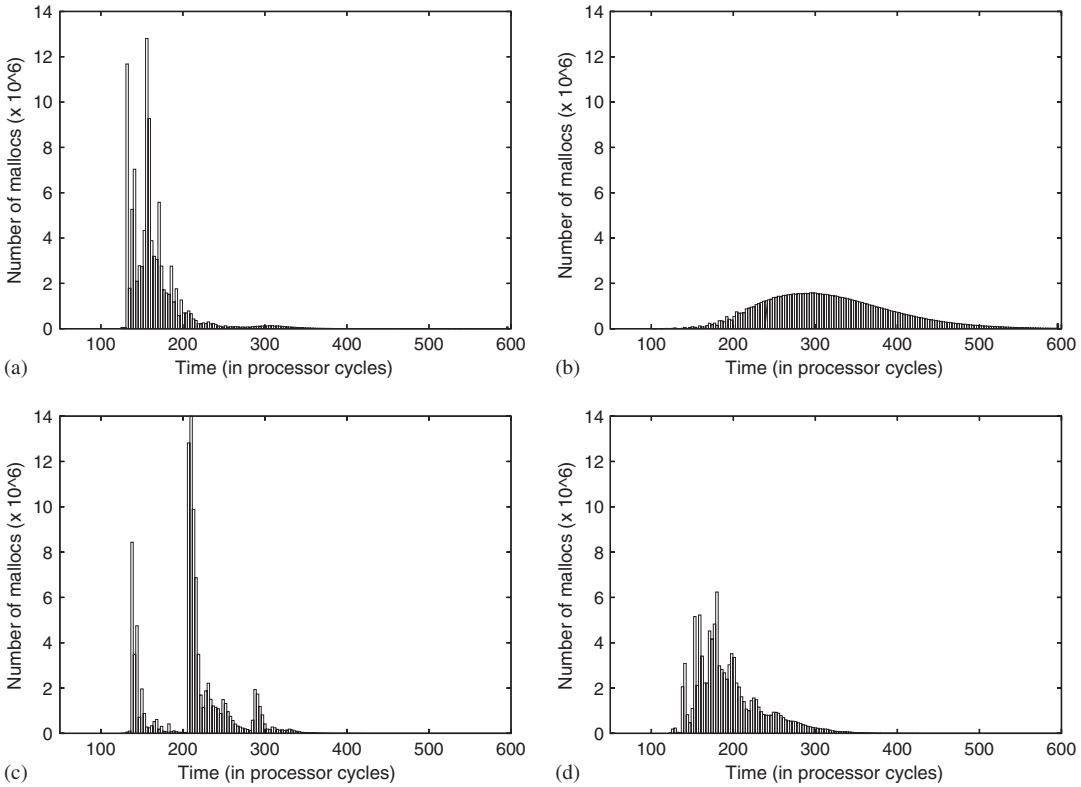

Figure 8. Distribution of the malloc processor cycles: (a) binary buddy; (b) DLmalloc; (c) half-fit; and (d) TLSF.

In the case of the free operation, the results obtained by all allocators are very similar in average, maximum and minimum values. Half-fit and TLSF have lower standard deviations.

7.3.2. Processor instructions

One way to eliminate the interferences caused by cache faults, TLB misses, etc. is to measure the number of instructions that each allocator executes. In order to achieve this, the test program has been instrumented using the *ptrace* system call. This system call allows a parent process to control the execution of its child process (test). The single-step mode permits the parent process to be notified each time a child instruction is executed.

Table V summarizes the results obtained in terms of the number of instructions executed per allocator according to previously designed tests.

The main conclusion from the results presented in this table is the bounded maximum number of processor instructions needed by the malloc operations of half-fit and TLSF, and the very low

DOI: 10.1002/spe

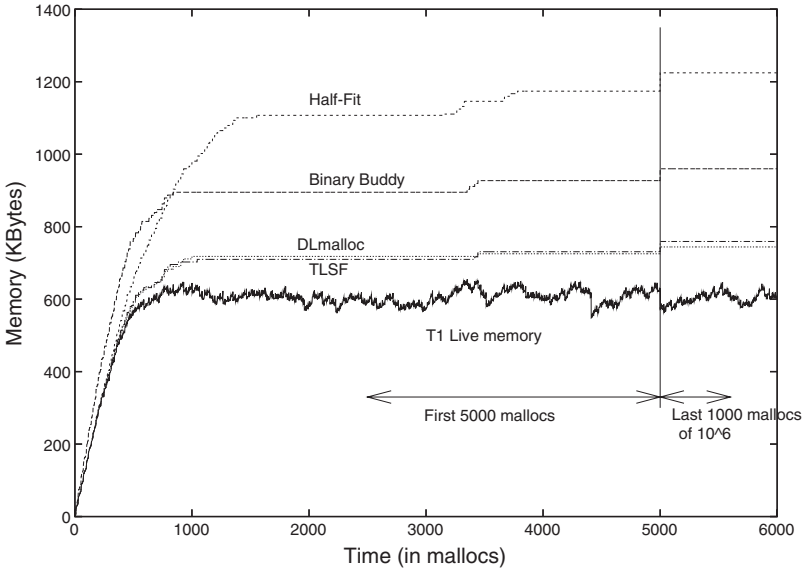

Figure 9. Memory used during one of the simulations of profile 3.

standard deviations. They have a constant cost (117 and 150 instructions observed in all profiles) and the best average values. Similar results can be seen for free operations.

Note that the difference between the instructions required by TLSF (malloc and free) is slightly lower than that shown in Figure 6(b) due to the additional instructions required to instrument the code and detect all paths in the algorithm execution.

7.3.3. Fragmentation

To measure the fragmentation incurred by each allocator, we have calculated the factor $\mathcal{F}$, which is computed 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).

Table VI shows the fragmentation obtained with each allocator. As it was previously described, factor $\mathcal{F}$ has been measured at the end of each scenario. This factor provides information about the percentage of additional memory required to allocate the application requests.

On the whole, results show that TLSF and DLmalloc require less memory (lower fragmentation) to allocate the requested load. Again, both allocators have a very good response in most of the tests (very low standard deviations).

Binary buddy and half-fit produce very high fragmentation (50 and 80%, respectively). As expected, the high fragmentation caused by binary buddy is due to the excessive size round up (round up to power of 2). 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 it can be seen,

DOI: 10.1002/spe

both allocators are quite sensitive to request sizes that are not close to the power of 2, thus causing a high fragmentation (internal fragmentation in the case of the binary buddy and external one in the half-fit case).

Figure 9 shows the variation of fragmentation as a function of the number of `mallocs` requested. The plot labelled as `Live_Memory` corresponds to the amount of memory requested by the application, which is the same for all allocators. Other plots show the memory needed by each allocator during execution. In this figure we have plotted the first 5000 mallocs (in order to observe the initial variation) and the last 1000 mallocs of the test (10^6 mallocs). As the previous table showed, TLSF and DLmalloc have similar variations. Higher fragmentation is obtained with binary buddy and half-fit.

8. CONCLUSIONS

TLSF is a *good-fit* DSA designed to meet real-time requirements. TLSF has been designed as a compromise between constant and fast response time and efficient memory use (low fragmentation). In this paper, we have detailed the design and the implementation criteria to fulfil the initial requirements.

TLSF uses segregated lists and bitmaps of the segregated lists, which can be directly reached through the size of the blocks. The use of both data structures permits constant and fast search operations to access the appropriated lists. The tree structure has been implemented more efficiently with a two-dimensional array, where the first dimension (first-level directory) splits free blocks into size ranges of a power of 2 apart from each other. The second dimension splits each first-level range linearly into a number of ranges of equal width. Low fragmentation is achieved with this second dimension, which allows the definition of small ranges of block size.

TLSF has been evaluated from the perspective of both temporal and spatial costs. A detailed analysis of the TLSF execution permits an in-depth study of the allocator, identifying the different parts of the algorithm and extracting the cost of each internal operation. The analysis has been completed with an exhaustive evaluation of the fragmentation incurred by TLSF. This has included all possible combinations of power of 2 subranges in a wide block-size spectrum. This evaluation shows that worst-case fragmentation is lower than 30%, while average values are around 15%.

When TLSF is compared with other well-known allocators, the results obtained can be considered as good as the best. Whereas all allocators present good results in response time, there is a significant difference in the total amount of memory needed to allocate a workload. Half-fit and binary-buddy fragmentation are not appropriate for embedded systems when memory is limited. On the other hand, DLmalloc and TLFS reach similar results in fragmentation. Considering both measurements (temporal and spatial), we can conclude that TLSF has the best performance of all compared allocators under the synthetic workload generated.

There are still some open issues, and in particular the determination of appropriated workloads for evaluation purposes. It is difficult to find examples of real-time systems using dynamic memory (because of the unpredictability of current allocators). Also, calculating the maximum live memory of a program is difficult, although it can be integrated with WCET analysis techniques which already perform dataflow analysis of programs.

DOI: 10.1002/spe

APPENDIX A: SPECIAL BITMAP FUNCTIONS

If the processor does not provide the operations ffs and fls, they can be implemented with no loops as shown below[‡‡]

```
int generic_ffs(int x) {
    int r = 1;
    if (!x)
        return 0;
    if (!(x & 0xffff)) {
        x >>= 16;
        r += 16;
    };
    if (!(x & 0xff)) {
        x >>= 8;
        r += 8;
    };
    if (!(x & 0xf)) {
        x >>= 4;
        r += 4;
    };
    if (!(x & 3)) {
        x >>= 2;
        r += 2;
    };
    if (!(x & 1)) {
        x >>= 1;
        r += 1;
    };
    return r;
}
```

```
int generic_fls(int x) {
    int r = 32;
    if (!x)
        return 0;
    if (!(x & 0xffff0000u)) {
        x <<= 16;
        r -= 16;
    };
    if (!(x & 0xff000000u)) {
        x <<= 8;
        r -= 8;
    };
    if (!(x & 0xf0000000u)) {
        x <<= 4;
        r -= 4;
    };
    if (!(x & 0xc0000000u)) {
        x <<= 2;
        r -= 2;
    };
    if (!(x & 0x80000000u)) {
        x <<= 1;
        r -= 1;
    };
    return r;
}
```

ACKNOWLEDGEMENTS

This work has been partially supported by the following projects: FRESCOR (IST/5-034026), ARTIST2 (IST NoE 004527) and Thread (TIC2005-08665).

REFERENCES

1. Tofte M, Talpin J-P. Region-based memory management. *Information and Computation* 1997; **132**(2):109–176.
2. Borg A, Wellings A, Gill C, Cytron RK. Real-time memory management: Life and times. *ECRTS*, Dresden, Germany, 2006; 237–250.

[‡‡]More efficient solutions can be found in http://hackersdelight.org/.

DOI: 10.1002/spe

3. Wilson PR, Johnstone MS, Neely M, Boles D. Dynamic storage allocation: A survey and critical review. *Proceedings of the International Workshop on Memory Management*, Kinross, Scotland, U.K. (*Lecture Notes in Computer Science*, vol. 986), Baker HG (ed.). Springer: Berlin, Germany, 1995; 1–116.
4. Brent RP. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems* 1989; **11**(3):388–403.
5. Grunwald D, Zorn B. Customalloc: Efficient synthesized memory allocators. *Software—Practice and Experience* 1993; **23**(8):851–869.
6. Knuth DE. *The Art of Computer Programming*, *Volume 1*: *Fundamental Algorithms*. Addison-Wesley: Reading, MA, U.S.A., 1973.
7. Peterson JL, Norman TA. Buddy systems. *Communications of the ACM* 1977; **20**(6):421–431.
8. Lea D. A memory allocator. *Unix/Mail*, 6/96, 1996.
9. Bonwick J. The slab allocator: An object-caching kernel memory allocator. *USENIX Summer*, Boston, MA, U.S.A., 1994; 87–98.
10. Berger ED, Zorn BG, McKinley KS. Composing high-performance memory allocators. *SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, U.S.A., 2001; 114–124.
11. Sedgewick R. *Algorithms in C* (3rd edn). Addison-Wesley: Reading, MA, U.S.A., 1998.
12. Robson JM. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM* 1974; **21**(3):491–499.
13. Robson JM. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal* 1977; **20**(3):242–244.
14. Garey MR, Graham RL, Ullman JD. Worst case analysis of memory allocation algorithms. *Proceedings of the 4th Annual ACM Symposium on the Theory of Computing* (*STOC'72*). ACM Press: New York, 1972.
15. Ogasawara T. An algorithm with constant execution time for dynamic storage allocation. *Second International Workshop on Real-Time Computing Systems and Applications*, 1995; 21.
16. Masmano M, Ripoll I, Crespo A, Real J. TLSF: A new dynamic memory allocator for real-time systems. *Sixteenth Euromicro Conference on Real-Time Systems*, Catania, Italy, July 2004. IEEE: New York, 2004; 79–88.
17. Stephenson CJ. Fast fits: New methods of dynamic storage allocation. *Operating Systems Review* 1983; **15**(5). Also in *Proceedings of Ninth Symposium on Operating Systems Principles*, Bretton Woods, NH, October 1983.
18. Atienza D, Mamagkakis S, Leeman M, Catthoor F, Mendias JM, Soudris D, Deconinck G. Fast system-level prototyping of power-aware dynamic memory managers for embedded systems. *Workshop on Compilers and Operating Systems for Low Power*, New Orleans, LA, U.S.A., 2003.
19. Berger ED, Zorn BG, McKinley KS. Reconsidering custom memory allocation. *OOPSLA*, Seattle, WA, U.S.A., 2002; 1–12.
20. Neely MS. An analysis of the effects of memory allocation policy on storage fragmentation. *Master's Thesis*, 1996.
21. Johnstone MS, Wilson PR. The memory fragmentation problem: Solved? *Proceedings of the International Symposium on Memory Management* (*ISMM'98*), Vancouver, Canada. ACM Press: New York, 1998.
22. Shore JE. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM* 1975; **18**(8):433–440.
23. Rezaei M, Cytron RK. Segregated binary trees: Decoupling memory manager. *MEDEA'00*. ACM Press: New York, 2000.
24. Masmano M. Gestion de memoria dinamica en sistemas de tiempo real. *Technical Report*, *PhD Thesis*, Real Time Research Group, Universidad Politecnica de Valencia, 2006. http://rtportal.upv.es/rtmalloc [May 2006].
25. Robson JM. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM* 1971; **18**(2):416–423.
26. Nielsen NR. Dynamic memory allocation in computer simulation. *Communications of the ACM* 1977; **20**(11):864–873.
27. Zorn B, Grunwald D. Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation* 1994; 107–131.
28. Bollella G, Gosling J. The real-time specification for java. *IEEE Computer* 2000; **33**(6):47–54.
29. Weiderman NH, Kamenoff NI. Hartstone uniprocessor benchmark: Definitions and experiments for real-time systems. *Real-Time Systems* 1992; **4**(4):353–382.
30. Marchand A, Balbastre P, Ripoll I, Masmano M, Crespo A. Memory resource management for real-time systems. *ECRTS*, Pisa, Italy, July 2007; 201–210.

DOI: 10.1002/spe