A Constant-Time Dynamic Storage Allocator for Real-Time Systems *

Miguel Masmano (mmasmano@ai2.upv.es), Ismael Ripoll (iripoll@disca.upv.es), Patricia Balbastre (patricia@disca.upv.es), Alfons Crespo (alfons@disca.upv.es)
Department of Computer Engineering, Technical University of Valencia, Camino de Vera, 14, E-46071, Valencia, Spain

Abstract. Dynamic memory allocation has been used for decades. However, it has seldom been used in real-time systems since the worst case of spatial and temporal requirements for allocation and deallocation operations is either unbounded or bounded but with a very large bound.

In this paper, a new allocator called TLSF (Two Level Segregated Fit) is presented. TLSF is designed and implemented to accommodate real-time constraints. The proposed allocator exhibits time-bounded behaviour, $O(1)$, and maintains a very good execution time. This paper describes in detail the data structures and functions provided by TLSF. We also compare TLSF with a representative set of allocators regarding their temporal cost and fragmentation.

Although the paper is mainly focused on timing analysis, a brief study and comparative analysis of fragmentation incurred by the allocators has been also included in order to provide a global view of the behavior of the allocators.

The temporal and spatial results showed that TLSF is also a fast allocator and produces a fragmentation close to that caused by the best existing allocators.

Keywords: Dynamic storage allocation, real-time allocators, allocation algorithms evaluation.

1. Introduction

Although dynamic storage allocation 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. Even the name, *dynamic storage allocation*, seems to suggest the idea of dynamic and unpredictable behaviour.

The real source of the uncertainty seems to come from the basic definition of the problem. An application can request and release blocks of different sizes in a sequence that is, a priori, unknown to the allocator. The allocator must keep track of released blocks in order to reuse them to serve new allocation requests, otherwise memory will eventually be

[*] This work was partially supported by FRESCOR (IST/5-034026), ARTIST2 (IST NoE 004527) and Thread (TIC2005-08665)

exhausted. A key factor in an allocator is the data structure it uses to store information about free blocks. Although not explicitly stated, it seems that it has been accepted that even using a very efficient and smart data structure the allocator algorithm, in some cases, has to perform some sort of linear or logarithmic search to find a suitable free block; otherwise, significant fragmentation may occur. For example, buddy systems (Binary-buddy (Knuth, 1973), Fibonacci-buddy, double-buddy, etc.) use a small number of queues and employ a restrictive method to split and coalesce blocks, so that the list of free blocks always contains blocks of the same size. The fragmentation produced by these allocators is very high.

Regarding the way allocation and deallocation are managed, there are two general approaches to dynamic storage allocation (DSA): (i) 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); and (ii) implicit memory deallocation (also known as garbage collection), where the DSA is in charge of collecting the blocks of memory that have been previously requested but are not needed anymore. This paper is focussed explicitly on low level allocation and deallocation primitives. Garbage collection is not addressed in this work.

This work was conceived during the porting of the Ada runtime support to RTLinux in the OCERA project (OCERA, 2002). The execution of Ada programs requires the availability of dynamic memory management which was not supported by the real-time operating system. A preliminary design and implementation of the allocation algorithm was presented in (Masmano et al., 2003; Masmano et al., 2004). During the evaluation process of this allocator we found another allocator proposal, called Half-fit (Ogasawara, 1995), with a similar approach to meeting real-time constraints but with very high fragmentation. The similarities of both works are significant, and will be discussed in the paper. This allocator passed unnoticed by the real-time community until it was rediscovered by the authors of this paper.

The contributions of this paper can be summarised in following points:

- The proposed dynamic memory allocator (TLSF) is the first algorithm that performs the allocation/deallocation in constant time maintaining a very low memory fragmentation. Moreover, its efficiency, in terms of number of processor instructions, is very high.

- The detailed comparative analysis shows that TLSF outperforms all other allocators regarding real-time requirements.

The paper is organised as follows: the following section summarises the dynamic memory requirements of a real-time environment. Section 4 describes the work carried out on real-time allocation algorithms. It includes a categorisation of the allocation policies and a brief description of the selected allocators, detailing its temporal and spatial behavior. The TLSF allocator is presented in Section 5. Section 6 describes the metrics and experimental framework. Section 7 presents the worst-case analysis and the synthetic and real workload used in the experiments. In Section 8, experimental results are presented and discussed. The last section concludes by summarising the results obtained and outlines open issues and the directions of future work.

2. Motivation

Nowadays portable consumer embedded devices have experienced a very fast growth in their variety, complexity and functionality, including multimedia and wireless network applications demanding extensive use of memory. These new applications (e.g., MPEG4 or network protocols) work with an important unpredictability of their input data.

This lack of predictability due to the size of the input data is present in other systems or applications such as:

- Embedded network systems. Routers, switches, and other network systems have to deal with different packet sizes. These devices use dynamic memory storage to handle incoming packets until they are delivered. Major network system manufacturers (CISCO, IBM, Infineon, etc.) report this functionality in their product characteristics.

- Video and voice devices. In these applications if the size of the buffer allocated to the user requests increases, then the latency and memory need also increases . Dynamic buffer allocation schemes permit to adjust the buffer sizes to the available resources in each situation.

- Games and graphic management. Game technology uses dynamic memory for a wide range of features. Functionalities such as 2D graphic drawing, surface rotation and scaling, 3D reconstruction, animation functions, etc., where the number of elements (corners, surfaces, shadow elements, etc.) change completely from one scene to the next one.

- Control of mobile robots. Control of mobile devices (such as robots, vehicles, ..) that have to deal with a high variability in the environment conditions (obstacles, map reconstruction, image processing, sensor fusion, etc.). Multi-agent systems are a concrete technology that perform agent allocation and deallocation depending on the system behaviour.

- Databases. Memory allocation in database systems has been studied extensively to handle query responses that can vary from a few bytes to a large amount of data.
- Web servers. Web servers integrate some of the above mentioned features such as databases, networking, video, etc., to return dynamic contents.
- Object oriented languages. Dynamic memory storage has been deeply studied and used in languages such as RTJava.

Most of these examples have real-time and spatial constraints. Efficient use of memory is a rather important issue mainly because they have to run for long periods of time and the use of memory has to be controlled.

3. Real-Time Requirements for DSA

One of the key issues in real-time systems is the schedulability analysis to determine whether the system can satisfy the timing constraints of the application at run time. Regardless of the analysis and scheduling techniques used, it is essential to determine the WCET of all the running code, including application code, library functions and operating system. Thus, the study and analysis of DSA functions have to be carefully performed to determine the cost in the worst-case situation.

Another characteristic that differentiates real-time systems from other 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.

Considering all these aspects, the requirements of real-time applications regarding dynamic memory allocation can be summarised as:

Bounded execution 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 completion time: Besides having a bounded execution time, it has to be short for the DSA algorithm to be usable. A bounded DSA algorithm 10 times slower than a conventional one may not be useful.

Minimise the memory pool size: The memory requests must always be 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 wasted memory. Therefore, an analytical bound on the maximum fragmentation is also needed.

4. Related work

In 1995 Wilson et al. (Wilson et al., 1995) presented a detailed survey of dynamic storage allocation which has been considered the main reference since then. The authors presented a comprehensive description, as well as the most important results, of all the problems related with 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.

Also in 1995, Ogasawara (Ogasawara, 1995) proposed the Half-fit allocator, which was the first to performs in constant time both to allocate and deallocate. Half-fit employs bitmaps and bit search instructions (available on most current processors) to achieve a worst-case execution time (WCET) of $O(1)$. Experimental analysis showed that Half-fit exhibits better memory usage than Binary-buddy. Half-fit uses a single level of segregated[1] lists, each list containing free blocks of sizes a power of two apart. This structure yields a theoretic wasted memory similar to that of Binary-buddy.

A different approach to achieving real-time performance was presented by R. Ford (Ford, 1996) in 1996. Ford analysed the contention problem, and the associated priority inversion, that arises when several concurrent tasks use a single memory pool. This work does not focus on the policy or mechanism used by the allocator but on the concurrent control of the allocator. Ford proposed a lock-free algorithm based on uncontrolled access followed by a validation and recovery phase.

[1] Segregated free list(Wilson et al., 1995) is an array of lists where each list holds free blocks of a particular range of sizes.

In 1993, Grunwald (Grunwald and Zorn, 1993) proposed an algorithm to implement allocators adapted to the target application. At the design phase the target application is instrumented and tested to collect data; this data is used later to synthesise the allocator. Although most custom allocators do not perform as well as expected (Berger et al., 2002a), we believe that this approach should be investigated further in real-time systems.

Every new allocator has to be analysed, tested and compared with others in order to show and measure the intended improvements. A lot of work has been done on performance evaluation to analyse both spatial and timing performance. B. Zorn (Zorn and Grunwald, 1994) investigated the accuracy of synthetic workload models, and concluded that many real applications have such complex memory request patterns that no synthetic allocator can simulate them properly. The real workload used by Zorn in this study has been used by most researchers since then.

In 2002, Puaut (Puaut, 2002) presented a performance analysis of a set of general purpose allocators with respect to real-time requirements. The paper presented detailed average and worst-case timing analysis. For the experimental analysis, she used three programs: mpg123, Cfrac and Espresso. In our experiments, we also used Cfrac and Espresso, as well as other programs, but not mpg123 because preliminary tests showed that the current version of mpg123 performed so few allocation/deallocation operations that it was not possible to obtain meaningful results.

4.1. Dynamic Storage Allocation Algorithms

This section presents a categorisation of existing allocators and a brief description of the most representative ones, based on the work of Wilson et al.(Wilson et al., 1995), which provides a complete taxonomy of allocators. According to their terminology, the allocators can be considered in the following aspects: strategy, policy and mechanism. Each allocation policy is motivated by an allocation strategy and implemented by an allocation mechanism.

Considering the main mechanism used by an allocator, the following categorisation is proposed (Wilson et al., 1995). Examples of each category are given. In some cases it is difficult to assign an allocator to a category because it uses more than one mechanism. In that case, we tried to determine which is the more relevant mechanism and categorise the allocator accordingly.

Sequential Fits: Sequential Fits algorithms are the most basic mechanisms. They search sequentially free blocks stored in a singly or doubly linked list. Examples are first-fit, next-fit, and best-fit.

First-fit and best-fit are two of the most representative sequential fit allocators, both of the are usually implemented with a doubly linked list. The pointers which implement the list are embedded inside the header of each free block[2]. The first-fit allocator searches the free list and selects the first block whose size is equal or greater than the requested size, whereas the best-fit goes further to select the block which best fits the request.

Segregated Free Lists: These algorithms use a set of free lists. Each of these lists store free blocks of a particular predefined size or size range. When a free block is released, it is inserted into the list which corresponds to its size. It is important to remember that the blocks are logically but not physically segregated. There are two of these mechanisms: simple segregated storage and segregated fits. An example of an allocator with this mechanism is Fast-fit (Stephenson, 1983), which uses an array for small-size free lists and a binary tree for larger sizes.

Buddy Systems: Buddy Systems (Knuth, 1973) are a particular case of Segregated free lists. Being $\mathcal{H}$ the heap size, there are only $\log_2(\mathcal{H})$ lists since the heap can only be split in powers of two. This restriction yields efficient splitting and merging operations, but it also causes a high memory fragmentation. There exist several variants of this method (Peterson and Norman, 1977) such as Binary-buddy, Fibonacci-buddy, Weighted buddy and Double-buddy.

The Binary-buddy (Knuth, 1973) allocator is the most representative of the buddy systems allocators, which besides has always been considered as a real-time allocator. The initial heap size has to be a power of two. If a smaller block is needed, then any available block can only be split 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.

[2] This technique is called *boundary tag*.

To mention a few examples: algorithms which use Adelson-Velskii and Landin (AVL) trees (Sedgewick, 1998), binary search trees or cartesian trees (Fast-Fit (Stephenson, 1983)) to store free blocks.

As mentioned above, the AVL allocator uses an AVL tree instead of a list. Basically, an AVL tree is a binary search tree where the heights of the two child subtrees of any node never differ in more than one, hence it is also known as a height-balanced tree. This interesting property is achieved by re-balancing the tree when it violates this restriction.

All operations on AVL trees (insert, remove and find) have a logarithmic time cost $O(1.44 \log_2(n))$.

Bitmap Fits: Algorithms in this category use a bitmap to find free blocks rapidly without having to perform an exhaustive search. Half-fit (Ogasawara, 1995) 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 to keep track of empty lists jointly with bitmap processor instructions are used 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 in two blocks of sizes r and r'. The remainder block of size r' is re-inserted in the list indexed by $i' = \lfloor \log_2(r') \rfloor$.

The fact of avoiding an exhaustive search and only considering the sizes of the free blocks as a power of two, causes what the author calls *incomplete memory use* (Ogasawara, 1995). The impact of incomplete memory use is discussed and analysed in Section 4.2.

The cost of this algorithm is constant ($O(1)$).

Hybrid allocators: Hybrid allocators use different mechanisms to improve certain characteristics (execution time, fragmentation, etc.) The most representative is Doug Lea's allocator (Lea, 1996), which is a combination of several mechanisms depending on the requested size. 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 [3].

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 organised in a vector of 32 trees, which are segregated in power-of-2 ranges, with two equally spaced treebins for each power of two. For each tree, its power-of-2 range is split in half at each node level with the strictly smaller value as the left child. Same sized chunks reside in a FIFO doubly linked-list within the nodes.This allocator uses a single array of lists, where the first 48 indexes are lists of blocks of an exact size (16 to 64 bytes) called “fast bins” . The remaining part of the array contains lists of segregated lists, called “bins”. These segregated lists are sorted by block size. A mapping function is used to quickly locate a suitable list. DLmalloc uses the delayed coalescing strategy, that is, the deallocation operation does not coalesce blocks. Instead a massive coalescing is done when the allocator cannot serve a request.

DLmalloc is considered one of the best and is widely used in many systems (glibc, eCos, etc.). There have been several releases of this allocator. Current releases have major changes and improvements over previous ones.

Additionally, several custom allocators have been proposed (Grunwald and Zorn, 1993; Bonwick, 1994; Atienza et al., 2003). They are designed considering the specific behaviour of the target application and can be tuned to improve time performance or optimise memory footprint. However, in (Berger et al., 2002b) several custom allocators are evaluated and, in general, their performance is worse than DLmalloc’s.

4.2. Fragmentation

Although this paper is mainly focused on the temporal analysis of the TLSF, we have considered that the fragmentation incurred by this, and others algorithms, should be taken into account. Due to paper limitations, we present a preliminary analysis of the fragmentation. A more detailed analysis will be stated as future work.

The notion of fragmentation seems to be well understood. It is hard to define a single method of measuring or even defining what

[3] A detailed description of the algorithm can be found in the comments of the the code of the allocator [*h*ttp://gee.cs.oswego.edu].

fragmentation is. In (Wilson et al., 1995) fragmentation is defined as "the inability to reuse memory that is free".

Historically, two different sources of fragmentation have been considered: internal and external. Internal fragmentation is caused when the allocator returns to the application a block that is bigger than the one requested (due to block round-up, memory alignment, unablility to handle the remaining memory, etc.). External fragmentation occurs when there is enough free memory but there is not a single block large enough to fulfill the request. Internal fragmentation is caused only by the allocator implementation, while external fragmentation is caused by a combination of the allocation policy and the user request sequence.

In (Robson, 1971; Robson, 1974; Robson, 1977) Robson analysed the worst-case memory requirements for several well known allocation policies. Robson designed allocation sequences that force each policy cause its maximum external fragmentation. If the maximum allocated memory (live memory) is $\mathcal{M}$ and the largest allocated block is m, then the heap size ,$\mathcal{H}$, needed for the First-fit algorithm is: $\frac{\mathcal{M}}{\ln 2}\sum_{i=1}^{m}(\frac{1}{i})$. Best-fit worst-case spatial cost is close to: $\mathcal{M}(m-2)$. According to (Knuth, 1973), the worst case for Binary-buddy[4] is: $\mathcal{M}(1+\log_2 m)$. Robson also showed that an upper bound for the worst-case of any allocator is given by: $\mathcal{M} \times m$.

Most of the initial fragmentation studies (Shore, 1975; Nielsen, 1977) 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 (Zorn and Grunwald, 1994) investigated the accuracy of simple synthetic workload models and concluded that synthetic workload should not be used in the general case because it does not reproduce properly the behaviour of real workload.

Johnstone and Wilson (Johnstone and Wilson, 1998) 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 which have been released recently over those that were released further in the past.

[4] This result is not directly presented in the Knuth book but let as a reader exercise.

There are many ways to measure the spatial efficiency of an allocator: plots of heap size, maps of busy/free blocks, amount of times the allocator calls the *brk* system call[5], etc. In general, all the memory managed by the allocator that cannot be assigned to the application will be called *wasted memory.* In (Johnstone and Wilson, 1998), four ways to measure fragmentation were considered. Although, all of them give an idea of the amount of fragmentation, in the authors opinion, the factor $\mathcal{F} = (\mathcal{H} - \mathcal{M})/\mathcal{M}$ seems to be the most representative. In fact, this $\mathcal{F}$ factor measures the percentage of the maximum live memory with respect to the maximum memory needed at any time.

5. TLSF overview

For completeness, in this section we describe the allocator presented in (Masmano et al., 2003; Masmano et al., 2004), a dynamic memory allocation called TLSF (Two-Level Segregated Fit). A more detailed description of the algorithm internals can be found in (Masmano et al., 2007).

TLSF is a constant-time, good-fit allocator. The good-fit policy tries to achieve the same results as best-fit (which is known to cause a low fragmentation in practice (Johnstone and Wilson, 1998)), but introduces implementation optimisations so that it may not find the tightest block, but a block that is close to it. TLSF implements a combination of the segregated and bitmap fits mechanisms. The use of bitmaps allow to implement fast, bounded-time mapping and searching functions.

5.1. TLSF implementation details

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

TLSF uses word-size bitmaps and processor bit instructions to find a suitable list in constant time. For example, using the *ffs*[6] instruction it

[5] The brk function is used to increase dynamically the amount of memory allocated for the calling process

[6] *ffs*: Find first set. Returns the position of the first (least significat) bit set to 1.

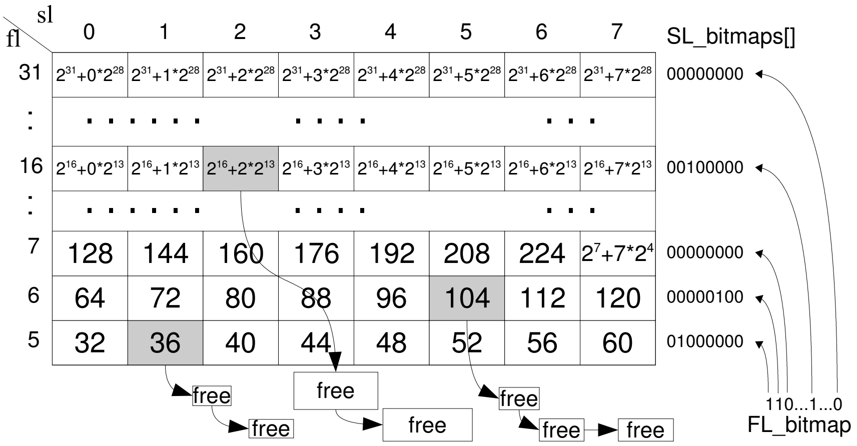

Figure 1. TLSF data structures example.

is possible to find the smaller non-empty list that holds blocks bigger or equal than a given size; and the instruction *fls*[7] can be used to compute the $\lfloor \log_2(x) \rfloor$ function. Note that it is not mandatory to have these advanced bit operations implemented in the processor to achieve constant time, since it is possible to implement them by software using less than 6 non-nested conditional blocks (see glibc or Linux implementation).

Given a block of size $r > 0$, the first and second indexes (fl and sl) of the list that holds blocks of its size range are: $fl = \lfloor \log_2(r) \rfloor$ and $sl = \left\lfloor (r - 2^{fl})/2^{fl-\mathcal{L}} \right\rfloor$. This expression for sl can be rewritten as $\lfloor r/s^{fl-\mathcal{L}} \rfloor - 2^{\mathcal{L}}$ which leads to a more efficient implementation. The function mapping_insert computes efficiently fl and sl:

```
procedure mapping_insert ($r$: integer; $fl$, $sl$: out integer) is
begin
  $fl$ := fls ($r$);
  $sl$ := ($r$ right_shift ($fl - \mathcal{L}$)) $- 2^{\mathcal{L}}$ ;
end mapping_insert;
```

For example, given the size $r = 74$, the first level index is $fl = 6$ and the second level index is $sl = 1$. The binary representation of the

[7] *fls*: Find last set. Returns the position of the most significant bit set to 1.

size gives an intuitive view of the values of fl and sl:

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

The list indexed by $fl = 6$ and $sl = 1$ is where blocks of sizes in the range [72..80[are located. But if the requested size is 74 and we search in this list, then we have to discard blocks of sizes 72 and 73, which will introduce an additional and unpredictable time to the algorithm. Instead of discarding smaller blocks, TLSF will start searching from the list of blocks whose minimum size is at least as large as the requested size. In the case of the example, it will start in $fl = 6$ and $sl = 2$, i.e., the list holding blocks of sizes [80..88[. This decision makes TLSF a Good-fit, rather than a Best-fit policy. The function mapping_search computes the values of fl and sl used as starting point to search a free block. Note that the requested size r is rounded-up to the next list to reduce fragmentation (see Section 4.2).

```
procedure mapping_search (r: in out integer;
                fl, sl: out integer) is
begin
  $r$:= $r$ + (1 left_shift (fls($r$) − $\mathcal{L}$))− 1;
  $fl$:= fls($r$);
  $sl$:= ($r$ right_shift ($fl$ − $\mathcal{L}$)) − $2^{\mathcal{L}}$;
end mapping_search;
```

Now the function search_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 second level indexes and then upwards in first level, until it finds the first non-empty list. Again, the use of bit find instructions allows to implement the search in a very compact manner.

```
function search_suitable_block (fl, sl: in integer)
             return address is
begin
  bitmap_tmp:= SL_bitmaps[$fl$] and (FFFFFFFF#16# left_shift $sl$);
  if bitmap_tmp $\neq$ 0 then
    non_empty_sl:= ffs(bitmap_tmp);
    non_empty_fl:= $fl$;
  else
```

```
      bitmap_tmp:= FL_bitmap and (FFFFFFFF#16# left_shift (fl+1));
      non_empty_fl:= ffs(bitmap_tmp);
      non_empty_sl:= ffs(SL_bitmaps[non_empty_fl]);
    end if;
    return head_list(non_empty_fl, non empty_sl);
  end search_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..112[.

```
function malloc (r: in integer) return address is
begin
  mapping_search(r, fl, sl);
  free_block:= find_suitable_block(r, fl, sl);
  if not(free_block) then return error; end if;
  remove_head(free_block);
  if size(free_block)−r > split_size_threshold then
    remaining_block:= split(free_block, r);
    mapping_insert(size(remaining_block), fl, sl);
    insert_block(remaining_block, fl, sl);
  end if;
  return free_block;
end malloc;
```

The *Free* function always tries to coalesce neighbour blocks. Merge_left checks whether the previous physical block is free, if so, it is removed from the segregated list and coalesced with the block being freed. Merge_right does the same operation but with the next physical block. Physical neighbours are quickly located using the size of the free block (to locate next block) and a pointer to the previous one, which is stored in the head of the freed block. The cost of all operations is constant ($O(1)$).

```
procedure free (block: in address) is
begin
  merged_block:= merge_prev(block);
  merged_block:= merge_next(merged_block);
  mapping_insert(size(merged_block), fl, sl);
  insert_block(merged_block, fl, sl);
end free;
```

5.2. Fragmentation study

The main focus of this paper is the temporal analysis of TLSF, however, considering that there are two proposals of constant time allocators, Half-fit and TLSF, the aim of this section is to present a comparative study of the theoretical fragmentation incurred by them.

Both Half-fit and TLSF implement the same policy: a set of segregated lists store blocks in ranges of sizes; the allocation mechanism just searches the segregated list that contains blocks whose size is equal to or larger than the one requested. Since all the blocks of the target segregated list are equal or larger than that requested, any block (and in particular the first block) of the list can be used to serve the request. Also, both allocators use bitmaps to find the suitable segregated list in constant time. One difference between them is the number of segregated lists. Half-fit uses a one-level array of 32 segregated lists, while TLSF can use up to 1024 segregated lists, arrangeded in a 32×32 matrix, which drastically reduces the amount of memory used.

The policy used to search for a suitable free block in Half-fit and TLSF introduces a new type of fragmentation or *incomplete memory use* as it is called in (Ogasawara, 1995): free blocks larger than the base size of the segregated list where they are located, will not be used to serve requests of sizes that are one byte larger than the base size. For example, in the case of the Half-fit, if the size of the largest free block is $2^{14} - 1 = 16383$, then any request to allocate a block larger than $2^{13} + 1 = 8193$ bytes will fail. The worst case is given by: $(2^{i+1} - 1) - (2^i + 1) \simeq 2^i$, when $r \in [2^i, 2^{i+1}[$, which represents 50% of the requested size.

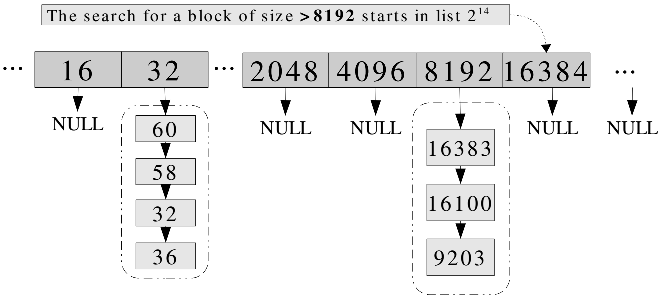

Figure 2. Incomplete memory use problem.

This problem (the existence of free blocks of the requested size that cannot be found due to the search policy) may happen quite often. Suppose the following allocation sequence: $40, 12, 40, 12, 40, 12, 40, 12$; then blocks of size 40 are freed; after that, the next allocation of a block of size 40 cannot reuse any of the already existing 40 byte free blocks, because they are in the “wrong” list. Any free block whose size is not a power of two can not be reused to serve a request of the same size. TLSF solves this problem using two policies:

1. There are more segregated lists (the second level can be configured to hold up to 32 lists, which gives a total of 1024 segregated lists), therefore the gap between segregated lists is much smaller, and so, the fragmentation is $(2^{i+1}/32 - 1) - (2^i + 1) \simeq 2^i/32$ which gives a fragmentation of $\simeq 3\%$.

2. Round-up the requested size to the next segregated list size. This way, when a block becomes free it will be placed on a list where it can be found later for a same size request. This policy converts the original problem (incomplete memory use) into a classical “internal fragmentation” problem. The worst-case fragmentation occurs when a request size is one byte bigger than an existing segregated list, and has to be rounded-up to the next list: $(2^{i+1}/32) - (2^i + 1) \simeq 2^i/32$; which gives a fragmentation of $\simeq 3\%$.

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%.

Since both allocators, Half-fit and TLSF, are based on a Good-fit policy, both exhibit a similar external fragmentation.

In section 8.3 we provide an experimental evaluation of the fragmentation generated by all tested allocators under different loads.

6. Evaluation issues

In this analysis we compare the performance of TLSF with respect to other allocators under different conditions. To evaluate the performance, several approaches may be considered including complexity analysis, simulation, and final implementation[8].

Since it is not possible to compare all the existing DSA algorithms as there are simply too many, a few of the most representative algorithms have been selected. The chosen allocators were: First-fit (sequential

[8] We use the term final implementation to refer to a real and usable implementation of the allocators.

fits), Best-fit (sequential fits), AVL tree (indexed fits), Binary-buddy (buddy systems), DLmalloc (hybrid algorithm using segregated and sequential system)[9] and Half-fit.

The workload that drives a simulation or a final implementation can be obtained in two ways:

- Using real workloads, which allow to evaluate the algorithm in real situations. Most of the research results on dynamic allocation have been obtained by using well known programs such as compilers (gcc, perl, etc.) or application programs (cfrac, espresso, etc.) as real workloads due to their intensive use of dynamic memory. However, real-time requirements have not been considered because of a lack of examples of use (the use of dynamic memory has usually been considered inappropriate for this kind of applications).

- Using synthetic workloads by extracting and modeling the events from a program at run time or by generating events randomly based on a probabilistic model.

In general, real workloads are more accurate than synthetic workloads, which are generated using simplifying assumptions. Several models have been proposed (Zorn and Grunwald, 1994) considering different parameters such as holding time, size or inter-arrival time of dynamic memory requests. Each model defines statistical characterisations of these parameters that attempt to accurately reconstruct the program behaviour. Also, synthetic workloads can be used to reproduce particular situations that require detailed analysis, for example worst-case situations.

A second evaluation issue is the performance metrics used for the evaluation. The following metrics have been used in our analysis:

Execution time: Time measurement of allocation and deallocation operations. For the execution time, three measures can be considered:

1. *Average execution time*: this is the key parameter for non real-time applications.
2. *Worst-case execution time*: this is the most important performance parameter for real-time applications and provides information of the maximum time required to serve a request. This corresponds to the execution time in the worst-case scenario.

[9] The version used in the comparison is 2.7.2.

3. *Standard deviation*: provides information about the execution time variability or stability of the allocator when working under different conditions.

However, exact, or even approximate, time measurement of a piece of code in current processors is not an easy task (Puschner and Burns, 2000). Current processors have been designed to improve throughput (average execution speed) at any cost. The techniques used to speed-up program execution (such as instruction reordering, jump prediction, pipelined units, cache memory levels, etc.) increase the worst-case execution time, and may introduce a large interference in the execution time of single instructions. Also the evaluation environment plays an important role in the measurement process. In order to reduce the uncertainty in the execution time measurement, ad-hoc frameworks, described in section 6.2, will be used.

Number of instructions: An alternative measure of the cost of an algorithm is the number of processor instructions executed. This metric is only sensitive to the processor instruction set and the compiler optimisation flags. The same program, compiled once and run in different processor implementations (e.g. Pentium, K7, Via or C3), will execute the same number of instructions. Also, a given algorithm fed twice with the same input will execute the same number of instructions in both cases.

Counting instructions provides results that should be close to those obtained with an analytical study.

It is important to point out that although on RISC processors the number of instructions executed may be a value close to, or related to, the time required to execute them, this is no longer valid on CISC processors due to the variation in the number of CPU cycles required by different instructions. Also, different algorithms may use a different mixture of instructions or even special instructions not used by other algorithms. For these reasons, instruction counting should be used only to compare how each allocator performs under a different workload. It cannot be meaningfully used to compare different allocators.

Fragmentation: As was described in section 4.2, a way to measure the spatial efficiency of an allocator is to determine the factor $\mathcal{F}$

as: *the maximum amount of memory used by the allocator relative to the maximum amount of allocated memory.* These two points do not necessarily occur at the same point during the program execution. For example, a value of factor $\mathcal{F} = 10\%$ means that the allocator needs 10% more of memory to satisfy the memory requests than the amount of total allocated memory. So, if the total amount of memory requested is $5.600Kb$ the total memory needed to satisfy these requests will be $6.160Kb$.

In this work we have used real workloads to determine the average time, standard deviation and memory fragmentation of the allocators; and synthetic workloads to build worst-case scenarios. For real-time systems, it is mandatory to evaluate the performance in the worst situation. It is also relevant to know how each allocator behaves compared to the others. Analysing each algorithm under its own worst-case scenario is very relevant for real-time systems. Comparing it under the worst-case scenarios of the others could be not necessary because it does not provide relevant information of the allocator. However, we have included it in order to confirm the expected results.

6.1. Evaluation methodology

In order to obtain the foregoing metrics, the following steps have been carried out:

- Worst-case execution time (WCET) and number of instructions measurement:
    - A worst-case scenario for each tested algorithm has been identified (Section 7.1).
    - A synthetic load modelling each worst-case scenario has been implemented.
    - Each allocator has been tested against all the worst-case scenarios.
- Average and standard deviation of execution time and fragmentation:
    - Several programs commonly used for evaluation purposes have been selected (Section 7.2).
    - Several tests using each program under different conditions have been designed.
    - For each test, all allocators are evaluated.

- Average execution times have been measured and standard deviations calculated. The maximum fragmentation incurred by each allocator has also been calculated.

6.2. Evaluation framework

We consider the evaluation framework as the environment (operating system and user programs) used to evaluate the allocators under different conditions.

This framework plays an important role in obtaining reliable and reproducible experiments. Simulation-based frameworks are reproducible, but it is difficult to take all the aspects of real hardware into account (pipelining, caching, etc.). We consider that we can obtain reliable and reproducible results by using *controlled* real environments. Two different frameworks were used to obtain the metrics:

Minimal execution layer . A minimal layer (μlayer) was built using the following off-the-shelf software components to execute the tests on a bare machine for worst-case execution time and number of instruction measurements. The functionality provided by μlayer is: simple string handling and mathematical functions; hardware control operations (interrupt handling, cache flush, time counting); and instruction counting using the processor tracing exception. Each test is executed immediately after a machine boot; therefore the system is in exactly the same initial state for all the tests. Once the scenario for the worst case has been set up, the cache is flushed and invalidated.

Minimal operating system . A minimal environment based on Linux has been used to obtain average execution time and fragmentation. The Linux Knoppix 4.0.2 distribution has been used to run the real workload. To minimise the interference of other processes, tests were performed while the system was at runlevel 1 (single user without network), and physically disconnected from the network to avoid undesired hardware interrupts.Cache blocks were flushed before running each test.

The system call `ptrace()` can be used to count the number of instructions. However, using `ptrace()` introduces a considerable overhead so that experiments using real workloads could last several months. Additionally, the results obtained provide less information about how the allocator behaves compared to others. Instead, we used the evaluation of the number of instructions only for worst-case scenarios of each allocator.

All measurements were obtained on an Intel® PIII(Coppermine) 803Mhz with 256Mb of main memory and 256Kb cache memory. GCC 3.3.6 compiler with "-O2 -fomit-frame-pointer" flags. Processor caches are not disabled, although invalidated on the worst-case tests.

7. Workload selection

Two different workloads have been used: synthetic workload to create worst-case scenarios and fragmentation evaluation, and real workload to compare average execution times.

7.1. Worst-case allocation scenarios

Although our intention was to find the worst-case scenario for each allocator, demonstrating that the described scenario causes the worst execution time is not always possible and is outside the scope of this work. Worst-case (WC) scenarios are used when possible and *bad-case*[10] (BdC) scenarios otherwise. A more detailed description of the worst-case allocation and deallocation of some of these allocators can be found in (Puaut, 2002).

First-fit/Best-fit

Allocation WC: The longest time required to allocate a block occurs when the free list has the longest length and the required block is located at the end of the free list. The list can be constructed by requesting blocks of minimum size until the heap is exhausted, then release two adjacent blocks (which will form a bigger block) and also release the remaining odd allocated blocks (which will be inserted at the head). The length of the list will be $O\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$.

Deallocation WC: The deallocation operation only coalesces with neighbours, if any, and the free block is inserted in the head of the unique free list. Hence, the worst-case is when the released block is surrounded by two free blocks (left and right physical neighbours). The cost is $O(1)$.

Binary-buddy

[10] A bad-case may be the worst-case, but it has not been proved.

Allocation WC: The worst-case scenario for Binary-buddy occurs on the very first request when this is of minimum size. In this case, the allocator has to split the initial free block several times in power of two until a block of the required size is obtained. The number of operations (block splits) required is $O(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$.

Deallocation WC: The worst-case deallocation is symmetrical to the worst-case allocation. When only one single block of minimum size has been allocated and this block is released. All the lists have to be merged to rebuild the original heap block. The cost is $O(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$.

AVL-tree:

Allocation BdC: There are several operations that contribute to the temporal cost of the allocator:

1. Search and remove a suitable block: $O(1.44\ \log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$
2. Insert the remaining block (if the block found is bigger than that requested), and re-balance the tree: $O(1.44\ \log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$

It is not simple to find the worst-case scenario due to the highly dynamic behaviour of the AVL data structure. The proposed bad-case scenario is: build the tallest AVL tree and request a block which can only be served using a leaf block. The requested block size will be such that a split will be needed.

Deallocation WC: The following bad-case scenario was designed and used: in the tallest tree (the tree that stores as many different block sizes as possible), a block is released that has to be coalesced with two neighbours and has to be inserted in the longest branch (the biggest free block). The cost is $O(3 \cdot 1.44\ \log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$.

DLmalloc

Allocation BdC: The proposed bad-case scenario tries to exploit the overhead produced by delayed coalescing. An extreme case occurs when all the heap has been allocated requesting minimum size blocks and then all the blocks have been released. This produces the longest free list that will be coalesced on the next malloc request.

The number of free blocks that are colaesced is $\frac{\mathcal{H}}{\mathcal{M}}$. The resulting asymptotic complexity is: $O\left(\frac{\mathcal{H}}{\mathcal{M}}\right)$.

Deallocation WC: Since DLmalloc delays coalescing, the free operation is quite fast, and has no clear worst-case path. Any free operation has a similar cost. The cost is $O(1)$.

Half-fit & TLSF

Both allocators implement the same strategy, therefore the worst-case scenarios are the same:

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

Deallocation WC: There are only three possible cases: 1) no free neighbours; 2) one neighbour; 3) two neighbours. The worst-case is when the two neighbours are free so that the allocator has to coalesce with both blocks. The cost is $O(1)$.

Table I summarises the costs of the worst-case allocation and deallocation of the algorithms.

Table I. Allocation and deallocation worst-case costs

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

7.2. Real workload

In order to obtain comparable results, we have selected the same workload as previous allocator evaluation studies. In 1994 Zorn (Zorn and Grunwald, 1994) presented an evaluation of synthetic workload used to test allocators based on real applications. They proposed a set of synthetic models and compared their accuracy with respect to the real workload. Since then, many studies have used their synthetic models or even the real workload they used in their experiments.

We have used a subset of the programs used in Zorn and Grunwald's seminal paper[11].

CFRAC: CFRAC is a program to factor integers using the continued fraction method.

Espresso 2.3: Espresso is a program which performs logic circuit simplifications.

GAWK 3.1.3: Gnu AWK is a free implementation of the AWK scripting language for string manipulation.

GS 7.07.1: GhostScript is a free PostScript and PDF language interpreter and previewer.

Perl 5.8.4: Perl is an interpreted programming language known for its power and flexibility.

Zorn and Grunwald's paper defines several tests per program, depending on the input used to feed them. All the tests used in this work are described in the appendix A.

8. Evaluation results

In this section, we compare the results of the selected allocators using the metrics and the frameworks described in section 6. We begin by detailing the results obtained for worst-case execution times and number of instructions. Next, we compare the statistical results when standard programs are used.

[11] All the code can be downloaded from ftp.cs.colorado.edu/pub/cs/misc/malloc-benchmarks

8.1. Results on WCET and number of instructions

WCET and number of instructions are evaluated using specific workloads designed to force each allocator to its worst-case scenario and measuring both metrics in this situation.

The test framework used was the *μlayer*. The memory heap size was set to 4Mbytes, and the minimum block size is 16bytes.

Table II. Worst-case (WC) and Bad-case (BdC) allocation: Processor instructions

Malloc	FF	BF	BB	DL	AVL	HF	TLSF
FF WC	**81995**	**98385**	115	109	699	162	197
BB WC	86	94	**1403**	729	353	162	188
DL BdC	88	96	1113	**721108**	353	164	197
AVL BdC	5085	6093	252	56093	**3116**	162	197
TLSF WC	88	96	1287	729	3053	**164**	**197**

Table III. Worst-case (WC) and Bad-case (BdC) allocation: Processor cycles

Malloc	FF	BF	BB	DL	AVL	HF	TLSF
FF WC	**1613263**	**1587552**	1445	1830	6471	1633	2231
BB WC	1168	1073	**3898**	4070	3580	1425	2388
DL BdC	1203	1227	3208	**3313253**	3844	1651	2251
AVL BdC	105835	101497	1703	132161	**11739**	1629	2149
TLSF WC	1168	1074	3730	4124	3580	**1690**	**2448**

Tables II(processor instructions) and III (processor cycles) show measurements of a single malloc operation after the worst-case scenario has been constructed, as described in section 7.1. Every allocator has been tested for each worst-case scenario. The result of an allocator when tested in its worst-case scenario is printed in bold face in the tables. The results show that each allocator performs badly in its theoretical worst-case scenarios.

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

It is interesting to note that although the TLSF and Half-fit have a high data locality they also have a high CPI ratio (around 12). This is due to two factors: first the use of arithmetic and logical operations to directly find a suitable block; and second, the complex data structure used to organise the segregated lists. The same test executed without invalidating processor caches takes less than 150 CPU cycles. First-fit does not benefit from non-invalidating the memory caches. In any case, the CPI is quite stable and depends mostly on the processor design[12] rather than data distribution.

The AVL bad-case scenario, which generates a large number of blocks of different sizes, is also a bad scenario for most allocators; only Binary-buddy, Half-fit and TLSF perform properly. This is due to the fact that the cost of many allocators depends on the number of free blocks as well as the number of different sizes.

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

Half-fit, TLSF, Binary-buddy and AVL show a reasonably low allocation cost, Half-fit and TLSF being the ones which show the most uniform response time, both in number of instructions and time. Half-fit shows the best worst-case execution time; only TLSF is 20% slower. Although Half-fit shows a fast and bounded execution time under all tests, it wastes a lot of memory due to the incomplete memory use caused by the search policy (see Sections 4.2 and 8.3), which makes this allocator inappropriate for many applications.

On the other hand though the AVL cost bound is not as good as TLSF, the practical bound is acceptable for most applications (logarithmic with respect to the heap size); in addition, since AVL implements a Best-fit policy and allocated block size is always the requested size, it has neither internal fragmentation nor incomplete memory use.

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

[12] Intel® processors show better CPI ratio than AMD® in these experiments.

Although a more detailed fragmentation analysis still has to be performed, TLSF achieves a good compromise between a constant allocation time and efficient memory management.

Table IV. Worst-case (WC) and Bad-case (BdC) deallocation.

Free	FF	BF	BB	DL	AVL	HF	TLSF
Processor Instructions	115	115	1379	51	1496	169	187
Processor cycles	1241	1289	4774	955	7947	1443	2151

Table IV summarises the number of instructions and processor cycles required by the free operation. Almost all allocators show a uniform deallocation cost. For this reason, this table shows only the results of each allocator in its deallocation worst-case.

All allocators except AVL and Binary-buddy do not perform any kind of search (the code does not contain loops), they just update some links if the neighbours are free. Binary-buddy has to coalesce up to $\log_2(\mathcal{H})$ blocks, so its cost is higher. AVL performs poorly due to the fact that the removal operation (the neighbours that have been coalesced) on an AVL tree has a non negligible cost. A free operation may require, in the worst-case, 2 removals and one insertion.

8.2. Average execution time and standard deviation results

In order to determine the behaviour of the TLSF allocator in real conditions and compare it against the others, we evaluated all allocators using real programs under the general-purpose framework described in 6.2. The tests used for each program are detailed in Appendix A.

Tables V to IX show the average time for malloc and free and the standard deviations, measured in processor cycles. The results have been rounded to the nearest integer.

The behaviour of the application (sequence of malloc and free operations) has a great impact on the performance of the allocator. It is not easy to model or describe in a simple way how an application uses dynamic memory. Moreover, a small change in the malloc/free request sequence may have a large impact on the allocator's performance. For example, in the case of segregated fits, if the requested sizes are not the same as at the start of the corresponding segregated lists then the allocator has to do more work to find a suitable block.

Among other factors, the following request patterns are relevant: the amount of different block sizes requested; the size of the blocks;

Table V. CFRAC memory allocation & deallocation (processor cycles)

Malloc/Free	FF	BF	BB	DL	AVL	HF	TLSF
Test 1 avg.	94/65	96/65	1131/114	**90**/**30**	512/335	107/143	231/204
std.dev.	40/38	47/36	7843/49	127/16	703/244	121/210	451/305
Test 2 avg.	131/69	138/69	265/145	120/**24**	750/603	**108**/98	146/116
std.dev.	520/32	511/30	1904/98	442/18	810/385	69/50	87/85
Test 3 avg.	129/72	142/75	254/153	115/**25**	780/762	**114**/116	149/121
std.dev.	505/32	518/32	1705/106	423/8	867/478	68/55	102/87
Test 4 avg.	132/87	144/89	241/182	**108**/**24**	503/801	114/118	158/121
std.dev.	486/41	519/44	1202/107	440/15	931/497	61/43	85/57
Test 5 avg.	139/105	158/106	247/195	**117**/**27**	536/955	126/132	176/134
std.dev.	529/58	594/56	1080/110	510/22	930/573	85/55	106/52

Table VI. Espresso memory allocation & deallocation (processor cycles)

Malloc/Free	FF	BF	BB	DL	AVL	HF	TLSF
Test 1 avg.	**77**/73	132/70	131/132	88/**25**	1132/683	80/104	126/108
std.dev.	124/29	109/32	440/89	319/10	373/468	20/38	36/47
Test 2 avg.	**73**/73	158/69	130/138	85/**25**	1144/709	79/104	123/108
std.dev.	74/30	76/32	288/92	262/9	339/505	15/35	30/47
Test 3 avg.	**74**/69	196/60	128/128	110/**26**	1305/708	78/98	114/95
std.dev.	105/27	111/29	268/84	845/11	448/556	12/32	31/42
Test 4 avg.	**74**/71	201/71	128/136	93/**24**	1242/624	78/102	122/105
std.dev.	56/27	75/30	121/92	905/8	356/505	12/33	20/41

how many requests of different sizes are interleaved (some programs use a large range of block sizes but in bursts of similar size blocks); the number of free operations (some applications only release a small amount of memory during run-time and release all memory on exit); the life time of allocated blocks (that is, the time between requesting and releasing a given block); etc.

Espresso, CFRAC and GAWK behave similarly with respect to the sizes of requested blocks; all of them make intensive use of small blocks. GhostScript (GS) exhibits a different size pattern (see figure 3), it allocates a wider range of block sizes and only a few of them are small blocks. GhostScript tests were the most demanding tests for all the allocators, including TLSF and Half-fit. But while most allocators

Table VII. GAWK memory allocation & deallocation (processor cycles)

Malloc/Free	FF	BF	BB	DL	AVL	HF	TLSF
Test 1 avg.	136/64	160/59	213/112	131/**28**	1081/424	**81**/109	120/92
std.dev.	659/38	609/41	1506/80	641/16	1388/373	38/48	62/58
Test 2 avg.	**77**/80	165/58	103/97	132/**28**	1161/810	88/115	124/101
std.dev.	36/34	65/31	78/55	249/13	422/461	15/36	34/49
Test 3 avg.	**77**/79	161/56	103/98	130/**27**	1148/806	89/115	123/99
std.dev.	25/33	64/30	66/58	247/13	423/455	14/36	35/47

Table VIII. GS (GhostScript) memory allocation & deallocation (processor cycles)

Malloc/Free	FF	BF	BB	DL	AVL	HF	TLSF
Test 1 avg.	1857/170	1834/148	1554/241	2042/**131**	3312/693	**227**/170	685/413
std.dev.	3031/277	2933/214	4827/232	2970/190	3041/965	189/202	964/693
Test 2 avg.	**196**/196	389/286	345/301	801/**143**	2483/1799	343/322	344/302
std.dev.	486/166	512/286	622/217	882/127	1558/979	301/298	245/207
Test 3 avg.	1918/192	1849/184	1130/321	1903/**128**	3788/568	**214**/176	419/257
std.dev.	3278/372	2997/323	3395/377	3206/211	3827/919	182/257	533/454

suffer significant performance loss (DLmalloc required 22 times more time than with other workloads; Binary-buddy 15 times; First-fit 10; AVL 8), the execution time of TLSF and Half-fit is only 2 or 3 times worse than with other workloads, which confirms their stability under very differently behaved workloads.

Regarding deallocation time, DLmalloc is unbeatable since it does not coalesce blocks. On the opposite side, AVL has a large deallocation

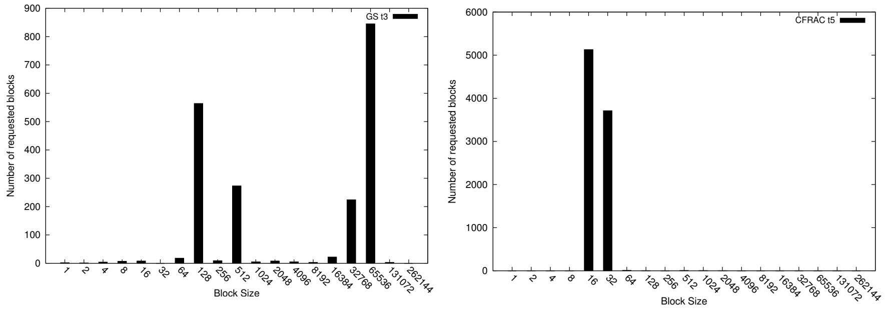

Figure 3. Memory size histogram.

Table IX. Perl memory allocation & deallocation (processor cycles)

Malloc/Free	FF	BF	BB	DL	AVL	HF	TLSF
Test 1 avg.	350/119	263/95	372/156	237/**34**	981/669	**95**/125	149/141
std.dev.	1900/149	1000/61	2376/115	970/34	1381/501	52/59	96/106
Test 2 avg.	126/132	170/105	192/146	**81**/**60**	1593/813	99/126	218/180
std.dev.	363/65	352/54	761/84	403/66	758/485	29/47	164/157
Test 3 avg.	83/82	131/65	111/111	**34**/**24**	1353/647	87/114	109/82
std.dev.	81/26	86/35	167/54	79/6	465/449	19/29	51/52
Test 4 avg.	83/91	135/58	112/112	**33**/**24**	1338/640	76/92	123/100
std.dev.	52/30	56/32	108/55	51/6	487/435	16/27	42/48

cost. Coalescing blocks on AVL, Binary-buddy, Half-fit and TLSF, may require up to two block removals and one insertion if both neighbours are free. DLmalloc just inserts the free block in the head of a list.

When considering the combined cost of allocating and releasing memory, DLmalloc shows the best average execution time, but with the GhostScript workload Half-fit is better.

8.3. Fragmentation results

In order to analyse the fragmentation generated by each allocator in real conditions, and according to the description in Section 7.2, we shall calculate the maximum amount of memory used by each allocator relative to the maximum amount of memory requested by the application (See the description of *Maximum heap size* metric on Section 6).

For the sake of brevity, Table X summarises the results of only the last tests of each tested program. The last tests are the ones that make more allocation/deallocation requests, and so the results are more representative. Note that a distinctive characteristic of real-time applications is that they run over a very long time.

Before we present the results, it is important to remind that First-fit, Best-fit and DLmalloc have shown unappropriate for real-time applications due to their worst-case temporal bound. Therefore, the most interesting results are those of Binary-buddy, AVL, Half-fit and TLSF.

The allocators that show the best results are Best Fit, AVL, DLmalloc and TLSF which are at a far distance from the rest that produce twice as much fragmentation at least. These results confirm that the policies implemented by TSLF (see Section 4.2, page 16) fulfil the low fragmentation requirement.

Table X. Fragmentation results

	FF	BF	BB	DL	AVL	HF	TLSF
Espresso Test 4	67.4%	7.1%	88.7%	7.3%	7.5%	17.4%	8.0%
Cfrac Test 5	92.3%	32.1%	89.8%	34.6%	33.1%	81.5%	38.0%
Gawk Test 3	61.2%	7.5%	44.5%	7.5%	7.9%	26.2%	9.1%
Gs Test 3	12.5%	0.3%	26.8%	0.3%	0.3%	1.1%	0.4%
Perl Test 4	16.5%	8.9%	34.4%	9.3%	9.0%	21.2%	10.3%

It is interesting to note the very large difference between the theoretical and the observed worst-case fragmentation with real workload. The same results have been observed in all fragmentation studies since (Robson, 1977).

9. Case study

In this section we present a case study based on the analysis of a router. Whereas the execution time of the allocators could be analysed in detail in the previous sections, the fragmentation is strongly dependent on the memory requests. In this case study, the analysis is focused on the fragmentation incurred by all allocators when real traces are used.

Roughly speaking, a router is a network device which links together different network segments. When a network receives a packet from one network segment, it extracts its destination, selects the best path to that destination, and forwards the packet to the most suitable network segment.

Current routers present many of the characteristics needed by any embedded real-time system: run for long periods of time and have response time and latencies constraints. Incoming messages (packets) are stored in the main memory waiting to to be sent by the appropriate output channel. Most of the router software uses fair scheduling policies (Ni and Bhuyan, 2002) by allowing the same amount of data to be moved and sent from each internal queue. In fact, variations of this scheduling algorithm are used by Cisco for commercial access points products and by Infineon in its new broadband access devices.

There exist several tools to simulate network systems (NS-2, OPNET, etc). Most of the implementations analyse packet latencies and traffic. However, they can also be used to assess the memory used by a dynamic memory allocator. By using these tools, we have modelled a standard router including additional features to detect the arrival of a packet (malloc operation) and its transfer (free operation) to the output channel.

In this case study, we have assumed that all dynamic memory request are allocated in a unique memory heap. Each output channel has a queue of references to packets allocated in the heap.

In order to perform the analysis, the router is fed with 20 real traces with traffic up to 10 Mbit/sec (LBNLab, 2000). Each trace collects daily internet traffic containing between 280.000 and 1 million of packets.

Figure 4 plots a histogram of one of the traces. Additionally, Fig. 5 displays the evolution of each allocator under this trace. The plot labelled *MemoryAllocated* represents the temporal evolution of the packets (in bytes) allocated in the router memory (live memory). The other plots (TLSF, DLmalloc, ...) show the maximum amount of memory (maximum memory address) needed by each allocator to serve this amount of memory. Fragmentation, as detailed before, is calculated as the percentage of the maximum live memory with respect to the maximum memory needed at any time.

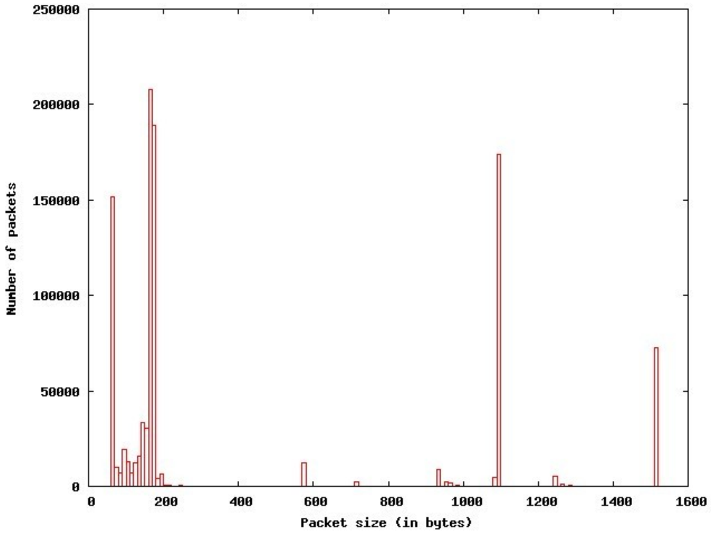

Figure 4. Packet size histogram.

Table XI shows the summary of the results obtained. For each test (trace) we measure the fragmentation of each allocator. *Average* and *Stddev* correspond to the average fragmentation obtained by all tests and the standard deviation. *Maximum* and *Minimum* detail, respectively, the maximum and minimum fragmentation test measured.

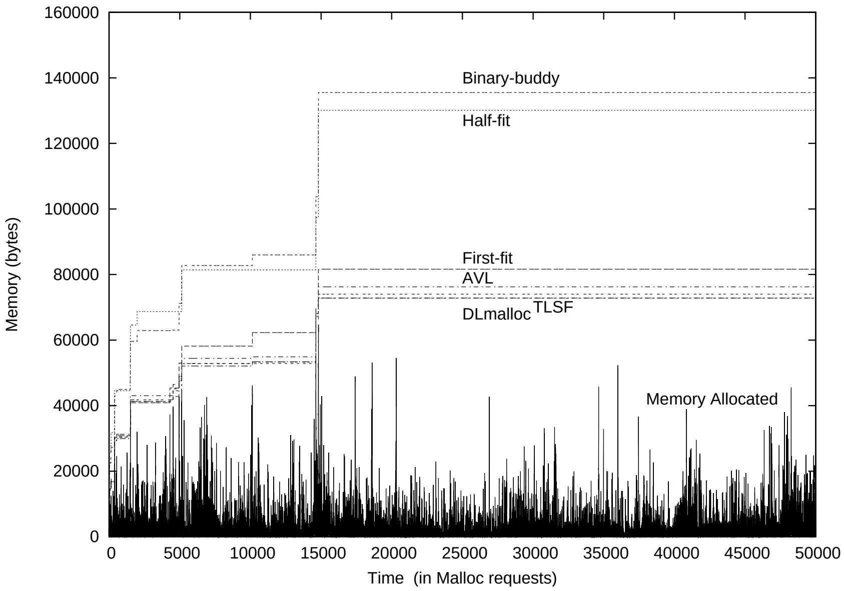

Figure 5. Evolution of the memory required by each allocator.

Table XI. Router case study: Fragmentation results (in %)

	Average	Std dev	Maximum	Mininimum
FF	38.94	7.08	48.84	31.61
BF	7.18	2.31	16.18	4.71
BB	50.26	8.32	81.77	44.13
DL	6.38	1.84	11.98	3.32
AVL	8.23	2.65	19.21	6.29
HF	78.85	6.79	97.74	68.26
TLSF	6.78	2.05	13.03	5.16

As expected, the best results are obtained by DLmalloc and TLSF. Both allocators achieve very small fragmentation in all traces. Also the maximum fragmentations are very close. On the opposite side, Binary-buddy and Half-fit allocators produce high fragmentation. Also, it has to be considered that the Internet traces have a strong constraint in size (packets have a maximum size of 1500 bytes), which clearly benefits to DLmalloc which uses different policies depending on the size requested.

10. Conclusions and future lines of investigation

TLSF is a dynamic storage allocator designed to meet real-time requirements. This paper has focused on comparing TLSF timing performance, both in worst-case and average execution time, with other well-known existing allocators (First/Best Fit, Binary-buddy, DLmalloc, Half-fit and AVL). Two different types of workload were used:

1. Worst-case scenario: A set of synthetic workload was designed to bring each allocator to its worst-case state. After the scenario is achieved, a malloc (or free) operation is issued and the execution time and number of processor instructions required to execute it are reported. These tests provide real measures of the WCET which complement the analytical cost analysis.

2. Real workload: This second type of tests provides information on the behaviour of the allocators in real situations. We have used the workload proposed by Zorn and Grunwald in (Zorn and Grunwald, 1994) which has also been used by other authors.

Additionally, a case study of a simulated router using real internet traces has completed the allocator analysis.

TLSF and Half-fit exhibit a stable, bounded execution time, which make them suitable for real-time applications. The bounded execution time of TLSF is not achieved at the cost of wasted memory, as is the case with Half-fit. Besides a bounded execution time, a good average execution time is also achieved with some real workload.

Our analysis also shows that allocators designed to optimise average execution time by considering the usage pattern of conventional applications, such as DLmalloc, cannot be used in real-time systems.

We have considered real applications as black boxes. Nonetheless, most applications (and real-time applications in particular) can be split up into three phases attending the allocation pattern: 1) start-up: buffers and data structures are allocated and initialised; 2) stable phase: the application continuously provides the intended services; and 3) shutdown: memory is released and application exits. Since real-time applications are long running and the start-up and the shutdown are usually done during the non-critical phase of the system, the analysis should be focused on the stable phase.

An allocator must fulfill two requirements in order to be used in real-time systems: 1) it must have a bounded execution time, so that schedulability analysis can be performed; 2) it must cause low fragmentation. It will also be desirable to have some kind of worst-case fragmentation analysis, similar to those used in schedulability analysis of tasks in real-time systems. The first aspect has been tackled in this work, while the second one is still a challenge.

Several lines of research have been opened from this work:

New features in the allocator. Region aware allocation: managing areas, or regions of memory, differently according on the kind of objects allocated. Dynamic pool resizing: been able to adapt the memory pool to the system needs.

Task model. While there exists a complete and consolidated model for the temporal requirements of real-time applications, the memory parameters that describe task behaviour are far from being well-defined and understood. In (Feizabadi et al., 2005), it is considered the maximum amount of memory that can be allocated per task. In the Real-Time Specification for Java (Bollella and Gosling, 2000), the model considers a limit on the rate of allocation in the memory pool. A more general model including techniques to analyse the system is desirable. Also, the specific characteristics of real-time applications should be considered, including: periodic request patterns, limited amount of allocated memory per task, bounded holding time, etc.

Resource management. Memory can be considered another resource in a real-time system, such as CPU or network are. Whereas there are well known schedulability techniques to analyse these resources, further research is still needed to develop memory analysis techniques enabling to guarantee the use of dynamic memory in real-time applications.

Appendix

A. Real-Workload description

This appendix describes all the tests mentioned in section 7.2.

CFRAC: in each test a number is factorised into two prime numbers. As can be seen in the code the complexity of the operation grows after each test.
Test 1: 23533.
Test 2: 1000000001930000000057.
Test 3: 327905606740421458831903.
Test 4: 4175764634412486014593803028771.
Test 5: 41757646344123832613190542166099121.

Espresso: in each test a combinational circuit is optimised, the only difference between them being the number of inputs and outputs used.
Test 1: 7 inputs and 10 outputs.
Test 2: 8 inputs and 8 outputs.
Test 3: 24 inputs and 109 outputs.
Test 4: 16 inputs and 40 outputs.

GAWK:
Test 1: Processing of a short text to create a checksum.
Test 2: Processing of a short text to fill the lines of the text.
Test 3: Processing of a big text to fill the lines of the text.

GS:
Test 1: Processing a single page including two graphics.
Test 2: Processing of the GNU C++ user guide.
Test 3: Processing of the SELF language manual.

Perl:
Test 1: Script that sorts the whole content of a small file regarding a given key.
Test 2: Script that translates a "/etc/hosts" file from the unixops format to the CS one.
Test 3: Processing of a short text to fill the lines of the text.
Test 4: Processing of a long text to fill the lines of the text.

References

Atienza, D., S. Mamagkakis, M. Leeman, F. Catthoor, J. M. Mendias, D. Soudris, and G. Deconinck: 2003, 'Fast System-Level Prototyping of Power-Aware Dynamic Memory Managers for Embedded Systems'. In: *Workshop on Compilers and Operating Systems for Low Power*.

Berger, E. D., B. G. Zorn, and K. S. McKinley: 2002a, 'Reconsidering Custom Memory Allocation'. In: *" ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications"*. Seattle, WA, pp. 1–12.

Berger, E. D., B. G. Zorn, and K. S. McKinley: 2002b, 'Reconsidering custom memory allocation.'. In: *OOPSLA*. pp. 1–12.

Bollella, G. and J. Gosling: 2000, 'The Real-Time Specification for Java.'. *IEEE Computer* **33**(6), 47–54.

Bonwick, J.: 1994, 'The Slab Allocator: An Object-Caching Kernel Memory Allocator'. In: *USENIX Summer*. pp. 87–98.

Feizabadi, S., B. Ravindran, and E. D. Jensen: 2005, 'MSA: a memory-aware utility accrual scheduling algorithm.'. In: *SAC*. pp. 857–862.

Ford, R.: 1996, ‘Concurrent Algorithms for Real-Time Memory Management’. *IEEE Software* **5**(5), 10–23.

Grunwald, D. and B. Zorn: 1993, ‘CustoMalloc: Efficient Synthesized Memory Allocators’. *Software Practice and Experience* **23**(8), 851–869.

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

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

LBNLab: 2000, ‘The Internet Trafc Archive. Lawrence Berkeley National Laboratory.’. http://ita.ee.lbl.gov/.

Lea, D.: 1996, ‘A Memory Allocator’. *Unix/Mail* **6/96**.

Masmano, M., I. Ripoll, and A. Crespo: 2003, ‘Dynamic storage allocation for real-time embedded systems’. In: *Real-Time Systems Symposium, Work-in-Progress Session.* Cancun, Mexico.

Masmano, M., I. Ripoll, A. Crespo, and J. Real: 2004, ‘TLSF: A New Dynamic Memory Allocator for Real-Time Systems’. In: *16th Euromicro Conference on Real-Time Systems.* Catania, Italy, pp. 79–88, IEEE.

Masmano, M., I. Ripoll, J. Real, A. Crespo, and A. J. Wellings: 2007, ‘Implementation of a constant-time dynamic storage allocator’. *Software: Practice and Experience.* Published Online: Oct 2007. DOI: 10.1002/spe.858.

Ni, N. and L. Bhuyan: 2002, ‘Fair scheduling in internet routers’. *IEEE Transactions On Computers* **51**(6), 686–701.

Nielsen, N. R.: 1977, ‘Dynamic memory allocation in computer simulation’. *Commun. ACM* **20**(11), 864–873.

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

Ogasawara, T.: 1995, ‘An algorithm with constant execution time for dynamic storage allocation’. *2nd Int. Workshop on Real-Time Computing Systems and Applications* p. 21.

Peterson, J. and T. Norman: 1977, ‘Buddy Systems’. *Communications of the ACM* **20**(6), 421–431.

Puaut, I.: 2002, ‘Real-Time Performance of Dynamic Memory Allocation Algorithms’. *14 th Euromicro Conference on Real-Time Systems* p. 41.

Puschner, P. and A. Burns: 2000, ‘A Review of Worst-Case Execution-Time Analysis’. *Journal of Real-Time Systems* **18**(2/3), 115–128.

Robson, J. M.: 1971, ‘An Estimate of the Store Size Necessary for Dynamic Storage Allocation.’. *J. ACM* **18**(2), 416–423.

Robson, J. M.: 1974, ‘Bounds for Some Functions Concerning Dynamic Storage Allocation.’. *J. ACM* **21**(3), 491–499.

Robson, J. M.: 1977, ‘Worst Case Fragmentation of First Fit and Best Fit Storage Allocation Strategies.’. *The Computer Journal* **20**(3), 242–244.

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

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

Stephenson, C. J.: 1983, ‘Fast Fits: New Methods of Dynamic Storage Allocation’. *Operating Systems Review* **15**(5). Also in Proceedings of Ninth Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983.

Wilson, P. R., M. S. Johnstone, M. Neely, and D. Boles: 1995, ‘Dynamic Storage Allocation: A Survey and Critical Review’. In: H. Baker (ed.): *Proc. of the Int.*

Workshop on Memory Management, Kinross, Scotland, UK. Springer-Verlag, Berlin, Germany. Vol:986, pp:1–116.

Zorn, B. and D. Grunwald: 1994, 'Evaluating Models of Memory Allocation'. *ACM Transactions on Modeling and Computer Simulation* pp. 107–131.