A Comparison of Memory Allocators for Real-Time Applications

Miguel Masmano Ismael Ripoll Alfons Crespo
Real-Time Systems Group
Universidad Politecnica de Valencia
Valencia, Spain
{mmasmano, iripoll,acrespo}@disca.upv.es

ABSTRACT

Real-Time applications can require dynamic storage management. However this feature has been sistematically avoided due to the general belief about the poor performance of allocation and deallocation operations in time and space. Actually, the use of Java technologies in real-time require to analyse in detail the performance of this feature due to its intensive use. In a previous paper, the authors proposed a new dynamic storage allocator that perform malloc and free operations in constant time ($O(1)$) with a very high efficiency. In this paper, we compare the behaviour of several allocators under "real-time" loads measuring the temporal cost and the fragmentation incurred by each allocator. In order to compare the temporal cost of the allocators, two parameters have been considered: number of instructions and processor cycles. To measure the fragmentation, we have calculated the relation between the maximum memory used by the each allocator relative to the point of the maximum amount of memory used by the load. Additionally, we have measured the impact of delayed deallocation in a similar way a periodic garbage collector server will do. The results of this paper show that TLSF allocator obtains the best resuts when both aspects, temporal and spatial are considered.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Performance; D.4.2 [**Allocation/deallocation strategies**]: Metrics—*complexity measures, performance measures*

General Terms

Dynamic storage management

Keywords

Dynamic storage management, Real-time systems, Operating systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES '06, October 11-13, 2006 Paris, France
Copyright 2006 ACM 1-59593-544-4/06/10 ...$5.00.

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.

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 exhausted. A key factor in an allocator is the data structure used to record 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[1] may occur.

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 in the analysis of an explicitly on low level allocation and deallocation primitives. Garbage collection is not addressed in this work. This explicit allocation is a low level functionality extensively used by the Java technologies.

There are several dynamic storage allocation strategies that have been proposed and analysed under different real or synthetical loads. In [19] a detailed survey of dynamic storage allocation was presented which has been considered the main reference since then. In the literature, several efficient implementations of dynamic memory allocators exists, [4, 8, 9, 13, 10, 3, 1, 18, 4]. Also there exist worst-case analysis of these algorithm from the temporal or spatial point of view [16, 17, 7].

There exist some allocators proposed for real-time. Ogasawara [12] proposed the Half-fit allocator, which was the

[1]Although the term "wasted memory" describes better the inability to use some parts of the memory, for historical reasons we will use the term "fragmentation" to refer to the same idea.

first to performs in constant time both to allocate and deallocate. TLSF [11] is another allocator performing these operations in constant time. In 2002, Puaut [15] presented a performance analysis of a set of general purpose allocators regarding with respect to real-time requirements.

In this paper, we compare the behaviour of several allocators under "real-time" loads measuring the temporal cost and the fragmentation incurred by each allocator. In the absence of "real-time" workloads for dynamic memory use, we have generated synthetic loads following a model similar to the Real-Time Java memory model. In order to compare the temporal cost of the allocators, two parameters have been considered: number of instructions and processor cycles. To measure the fragmentation, we have calculated the relation between the maximum memory used by the each allocator relative to the point of the maximum amount of memory used by the load (live memory).

An allocator must fulfil two requirements in order to be used in real-time systems: 1) it must have a bounded response time, so that schedulability analysis can be performed; 2) it must cause low fragmentation. The results of this paper show that TLSF obtain the best resuts when both aspects, temporal and spatial, are considered.

The paper is organised as follows: the following section provides a brief description of the allocators used in this paper. Section Section 3 presents the workload model and the test generation used in the comparison. Section 4 describes the metrics and experimental framework. In Section 4, 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. DESCRIPTION OF THE ALLOCATORS

This section presents a brief descrition of the allocators used in the evaluation. Allocators can be classified attending to the policy used (first fit, best fit, good fit, etc.) and the mechanism implemented (doubly linked lists, segregated lists, bitmaps, etc.) based on the work of Wilson et al.[19]

In order to perform the evaluation presented in this paper, we have selected some of the allocatores more representatives taking into account considerations like:

Representatives of very well known policies. First-fit and Best-fit are two of the most representative sequential fit allocators. First-fit allocator is used in all comparisons. It does not provide good results in terms of time and fragmentation but it is a reference. Best-fit provides very good results on fragmentation but bad results in time. Both of them are usually implemented with a doubly linked list. The pointers which implement the list are emb]dded inside the header of each free block. First-fit allocator searches the free list and selects the first block whose size is equal or greater than the requested size, whereas Best-fit goes further to select the block which best fits the request.

Widely used in several environments. Doug Lea's allocator [10] is the most representative of hybrid allocator and it is used in Linux systems and several environments. It is a combination of several mechanisms. 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". Each of 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 can not serve a request.

Labelled as "real-time" allocators. Binary-Buddy and Half-fit are good-fit allocators that provide excellent results in time reponse. However, the fragmentation produced by these allocators is known to be non negligible.

Buddy systems [9] 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 [13] such as Binary-buddy, Fibonacci-buddy, Weighted buddy and Double-buddy.

The Binary-buddy [9] 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.

Half-fit [12] uses bitmaps to find free blocks rapidly without having to perform an exhaustive search. 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 $i' = \lfloor \log_2(r') \rfloor$.

New real-time allocator. TLSF (Two-Level Segregated Fit) [11] is a bounded-time, good-fit allocator. TLSF implements a combination of segregated and bitmap fits mechanisms. The use of bitmaps allow to implement fast, bounded-time mapping and searching functions. TLSF data structure can be represented as a two-dimension array. The first dimension splits free blocks in size-ranges a power of two apart from each other, so that first-level index i refers to free blocks of sizes in the range $[2^i, 2^{i+1}[$. The second dimension splits each first-level range linearly in a number of ranges of an equal width. The number of such ranges, $2^{\mathcal{L}}$, should not exceed the number of bits of the underlying architecture, so that a one-word bitmap

can represent the availability of free blocks in all the ranges. TLSF uses word-size bitmaps and processor bit instructions to find a suitable list in constant time. The range of sizes of the segregated lists has been chosen so 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) of the range is similar for small blocks than for large blocks. In other words, there are more lists used for smaller blocks than for larger blocks.

One important aspect is the theoretical temporal cost (complexity) of each allocator. Table 1 summarises these costs for each allocator.

Table 1: Algorithm complexity

	Allocation	Deallocation
First-fit/Best-fit	$O\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$	$O(1)$
Binary-buddy	$O(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$	$O(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right))$
DLalloc	$O\left(\frac{\mathcal{H}}{\mathcal{M}}\right)$	$O(1)$
Half-fit	$O(1)$	$O(1)$
TLSF	$O(1)$	$O(1)$

In [11], the worst-case or bad-case[2] scenario of each allocator has been analysed and detailed. For each allocator, a sinthetic load was generated to conduct it to its worst-case allocating and deallocating scenarios. Once these scenarios were reached, we measured the number of instructions performed by the allocation or deallocation operations. Table 2 shows a summary of these results.

Table 2: Worst-case (WC) and Bad-case (BC) allocation and deallocation: Processor instructions

Allocators	Malloc	Free
First-Fit	81995	126
Best-Fit	98385	126
Bunary-Buddy	1403	1379
DLalloc	721108	83
Half-Fit	136	166
TLSF	170	194

These results can slightly change depending on the compiler version and the optimisation options used.

3. WORKLOAD 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. Considering the temporal requirements, real-time tasks are commonly represented as a set of parameters (computation time, deadline and period). This model is both simple, because the three parameters are characteristics that can be easily understood and measured, and complete because it is possible to make schedulability analysis using only that few parameters.

Memory requirements in task model has been considered in some works as [6, 5, 14]. However, it is only considered the maximum amount of memory that can be allocated per task. Feizabadi et al. [6] used this simple model and suggested to extend it to take into account the allocation patterns of tasks. Feizabadi called that new model WCAR (Worst Case Allocation Requirements).

At 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 [2]. In this model, a `RealtimeThread` can define its memory requirements using the `MemoryParameters` class, which contains the following information:

`maxMemoryArea` A limit on the amount of memory the thread may allocate in the memory area.

`maxImmortal` A limit on the amount of memory the thread may allocate in the immortal area (memory that live until the end of the application).

`allocationRate` A limit on the rate of allocation in the heap. Units are in bytes per second.

The Real-Time Java memory model has been designed around the needs of the garbage collector. That is, the activation period of the garbage collector can be calculated from the parameters of the tasks.

It is an open question to define a memory model for real-time periodic tasks. A complete memory model should contain parameters that determine the minimum and maximum block size, the holding time, maximum live memory, etc.

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

In this model a task T_i can ask for several blocks until a maximum of g_i bytes per period. Each request r_{ik} can have different holding time but shorter than h_i units of time.

To serve all memory requests, the system provides an area of free memory (also known as heap) of $\mathcal{H}$ bytes.

From this model, it is easy to derive the parameters proposed by the Real-Time Specification for Java.

Independently of the deallocation policy of each allocator, in order to cope different global deallocation strategies, two alternatives for freeding blocks have been considered:

Immediate deallocation : As soon as a block is not used, an explicit free operation is executed by the application task.

Delayed deallocation : Application tasks do not perform explicit block deallocation. Instead of deallocating blocks application task mark blocks as "not used". A periodic deallocation task performs the free operation on blocks that are not used by any task.

Whereas the first strategy permits the analysis of explicit allocation/deallocation applications, the second one can be used to analyse the effects of implicit deallocation systems as those based on garbage collector.

3.1 Workload model

In order to perform an evaluation of the considered allocators under the proposed model, a synthetic load generator

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

has been designed. The workload model generate set of tasks under the following premises:

1. Periods (P) are randomly generated using uniform distribution between a minimum and maximum admissible period.
2. Maximum amount of memory requested by period ($Gmax$) is randonmly generated using a uniform distribution in the range of maximum and minimum block size defined by the user.
3. Maximum number of requests (Mnr) per period follows a uniform distribution in the range of maximum and minimum block size.
4. The amount of memory by request follows a normal distribution with an average value ($Gavg$) obtained as $Gmax/Mnr$ and a standard deviation, calculated as a constant factor of the size block.
5. Holding time is determined using a uniform distribution between the minimum and maximum admissible holding time.

Three different profiles of tasks have been defined:

1. Set of tasks allocating high amount of memory per period requesting big blocks.
2. Set of tasks allocating small amount of memory per period requesting small blocks
3. Set of tasks of any of the previous profiles

3.2 Test definition

In order to evaluate each allocator the following tests have been designed.

Test 1: This first test is designed to evaluate the behaviour of allocators when requesting big blocks. The load consists in a set of [3..10] tasks with periods in the range [10..150]. The maximum memory per period g_i of each task is in the range of [2048..20480]. with a range of [2..5] requests per period. The holding time h_i of each request has also been calculated with an uniform distribution in the period range of [30..50]. The policy used to free blocks is immediate deallocation. An example of these scenarios is:

Table 3: Task set example of profile 1.

	Per.	Gmax	Gavg	Gsdv	Hmax	Htmin
T0	10	10203	10200	300	43	11
T1	19	17728	8471	200	24	7
T2	29	14503	6404	150	21	11

Figure 1 shows an example of the histogram of the mallocs sizes generated by this profile.

Test 2: The second load is oriented to evaluate the allocator behaviour when only small blocks are requested. The difference with respect to previous test is on the maximum memory per period of each task which is in the range of [8..1024]. One of the possible scenarios generated is shown in table 4

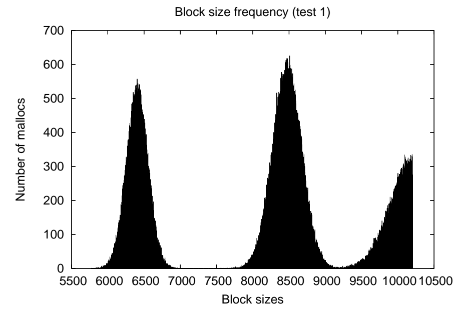

Figure 1: Block size histogram example of profile 1

Table 4: Task set example of profile 2.

	Per.	Gmax	Gavg	Gsdv	Hmax	Htmin
T0	23	786	235	28	43	14
T1	50	128	31	24	45	9
T2	60	418	169	32	30	12

Test 3: The third load tries to cover both previous cases. Now the range of the maximum amount of memory requested by period can vary in the range [8..20480].

Test 4: This test has been designed to analyse the effects of the delayed deallocation policy on the memory needs to serve a workload. One of the scenarios generated by Test 3 is taken as basis for the analysis. Using this set of tasks, we executed it for a range of deallocation task period from the smallest task and 10 times the largest period of the application task set (see section 4.4).

4. EXPERIMENTAL ANALYSIS OF THE ALLOCATORS

In this section we present the results obtained by the selected allocators under different loads. First fit, Best fit and Binary buddy allocators has been implemented for these evaluation from other implementations. DLmalloc code has been taken from the author web site (version 2.7.2). Finally, we could not find any detailed implementation of Half-fit which has been implemented from the Ogasawara paper description.

Each allocator test consists in the execution of 10 tests of each profile with different random seeds. Each test measures the following metrics:

Execution time: The number of processor cycles needed by each allocation and deallocation operation has been measured. In order to avoid the effects of system interrupts these operations have been executed with disabled interrupts. Although the test platform has been designed to reduce the interferences of other processes, still there are some factors (as cache faults, TLB, etc.) that can produce significant variations in the execution. Additionally, each test was executed twice, using the same seed. As the operation size requests are in the same order, the results of both execution should produce very similar number of cycles for each operation. Differences are associated to processor interferences that could be minimise selecting the minimum number of cycles of both outputs. Final results are

Table 5: Temporal cost of the allocator operations in processor cycles

(a) Malloc

	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	315	239	2185	97	248	243	2274	97	291	279	4758	97
Best-fit	511	513	2330	112	347	352	2380	95	1134	1133	6432	107
Binary-buddy	170	472	5517	143	157	262	7433	105	161	263	5960	121
DLmalloc	342	345	5769	114	249	278	5859	79	292	296	6985	83
Half-fit	196	332	1237	129	148	569	1269	108	161	520	1491	128
TLSF	216	274	2017	137	173	257	2056	115	196	230	2473	115

(b) Free

	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	163	189	1433	84	148	185	1387	84	176	196	1529	85
Best-fit	153	189	1420	85	120	212	1292	84	143	178	1448	85
Binary-buddy	147	285	1728	120	150	287	855	120	153	284	1412	120
DLmalloc	124	198	644	85	99	354	425	74	128	181	732	75
Half-fit	184	209	1273	104	173	216	1209	104	186	210	1099	110
TLSF	201	220	1641	118	170	223	741	111	191	217	1095	118

presented as average, standard deviation, maximum and minimum number of cycles.

Processor instructions: One way to eliminate the interferences is to measure the number of instructions of each allocator. In order to measure it, the test program have been instrumented using the *ptrace* system call. This system call allows a parent process to control the execution of the child (test) process. The single step mode permits to the parent process be notified each time a child instruction is executed. As in the previous metric, results are provided in average, standard deviation, maximum and minimum number of instructions executed.

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).

In order to clarify the fragmentation measure, figure 2 shows a first trivial example generated by a single task under the Best-Fit allocator. The figure shows the memory used by this allocator to satisfy the requests of this task. The continous line shows the maximum memory address reached by the allocator while the non-continous line plots the live memory (used by the task). The live memory draws a periodic shape whose rise coincides with the period of the task (15 u.t.) and it falls at the end of the holding time of each allocation (60 u.t.). In the figure 2, $\mathcal{F}$ is 47.06% and corresponds to the point 1 relative to the point 2.

The simulation time is measured in number of mallocs units. The number of mallocs analysed has been 300.000 for processor cycles or number of instructions measurement and 1.500.000 for fragmentation measurement.

4.1 Execution time results

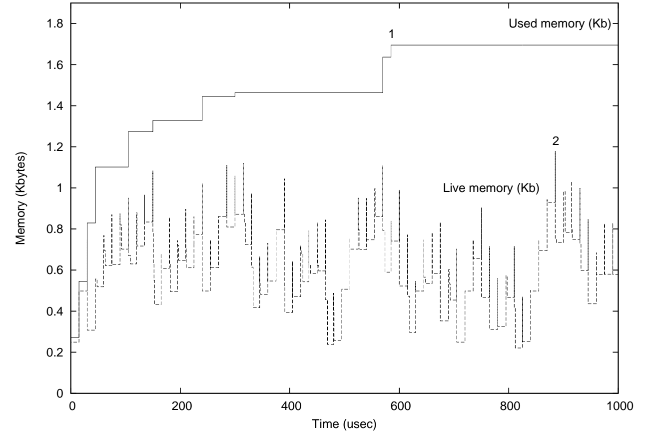

Figure 2: Memory usage of Best-Fit with 1 task

Table 5 shows a summary of the processor cycles spent for each allocator for both operations: malloc and free.

In general, all allocators need more cycles when allocate higher blocks than small blocks. Binary-Buddy, Half-fit and TLSF show a stable (lower standar deviation) and efficient (less cycles) than other allocators. Doug Lea allocator presents a good execution performance. In this allocator, the cost of deferred coalescence used is measured in some malloc operations (see maximum number of cycles), so the maximum value is relatively higher than other ones.

This deferred coalescence is the reason why Doug Lea allocator presents the best results when free operations are executed. All allocators perform the free operation with a reduced number of cycles and similar standard deviation. The free operation implementation of First-fit and Best-fit allocators is the same obtaining very similar results.

4.2 Processor instructions results

Table 6 summarises the results of numbers of instructions executed by each allocator in the previous designed tests.

Table 6: Temporal cost of the allocator operations in processor instructions

(a) Malloc

	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	204	21	478	71	201	17	818	70	203	23	957	70
Best-fit	582	69	798	76	442	130	1006	76	805	179	1539	76
Binary-buddy	169	17	843	157	136	22	1113	95	153	24	1113	95
DLmalloc	279	107	921	64	161	126	933	49	232	152	1277	57
Half-fit	118	1	123	115	116	7	123	76	118	1	123	82
TLSF	147	13	164	104	118	25	164	84	133	22	164	84

(b) Free

	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	93	96	128	59	90	92	128	57	92	95	128	57
Best-fit	91	115	126	57	69	148	126	57	79	198	128	57
Binary-buddy	68	70	225	65	68	72	277	65	69	73	228	65
DLmalloc	70	128	77	53	59	177	77	39	67	168	77	39
Half-fit	117	117	167	73	115	116	165	76	117	117	167	76
TLSF	140	140	217	91	107	110	216	87	120	122	217	87

TLSF and Half-fit demonstrate the excellent behavior achieved. Both obtain the lowest average and standard deviation. For the same reasons stated in the number of cycles, DLmalloc presents reasonable good average but high maximum values.

With respect to the free operations the results are very similar. As expected, DLmalloc presents the best results.

A more detailed analysis of each allocator behaviour can be seen in figure 3. These plots show the evolution of the processor instructions executed by the malloc operation on time (in number of malloc requests). Note that plots have different y axis range. We can appreciate that Half Fit and TLSF response time (instructions) is constant, less than 139 and 170 respectively.

At the beginning of the simulation, First fit allocator works reasonably well, but as the number of free blocks gets higher, the variation of the response time gets larger. Best fit presents the worst behaviour of the analysed allocators. The more time the application runs, the more free blocks exists, and so, the more time it needs to find the block which fits best. In the Binary buddy plot, it is possible to appreciate the bands caused by the power of two splits. Although the average response time of the Douglas Lea is very fast, from time to time it has to coalesce blocks (points above 500 instructions), which is a costly operation.

A deeper analysis of the TLSF plot (Figure 3(f)) permits to detect the different groups of instructions executed depending on the sizes of the allocation requests. The two horizontal bands between 90 and 100 instructions correspond to block requests whose size is lower than 128 bytes when the block used to serve the request is not split. Bands between 105 and 115 correspond to request of blocks greater than 128 bytes without block split. When the TLSF has to split a block in order to attend a request (split and insert the remaining block on the structure) the cost gets higher, bands above 140 instructions. Variations around a band are consequence of the insertion in a empty list or deletion of the last element of a list.

Half fit has also two groups: points below 85 and points above 110 instructions. The lower bands happens when no split is done, and the upper bands correspond to requests that required to split the free block.

4.3 Fragmentation results

Table 7 shows the fragmentation obtained by each allocator. As it was detailed above, factor $\mathcal{F}$ has been measured at the end of each scenario. This factor provides information about the percentage of the additional memory required to allocate the application requests.

These results show that, overall, TLSF is the allocator that requires less memory (less fragmentation) closely followed by Best-Fit and DLmalloc. As shown, TLSF behaves better even than Best-Fit in most cases, that can be explained due that TLSF always rounds-up all petitions to fit them to any existing list, allowing TLSF to reuse one block with several petitions with a similar size, independenly of their arrival order. On the other hand, Best-Fit always splits blocks to fit the requested sizes, making impossible to reuse some blocks when these blocks have previously been allocated to slightly smaller petitions. For example, for a request of 130 bytes, TLSF will always allocate 132 bytes. Best-Fit will allocate 130 bytes instead. If this block is released and 132 bytes are requested later, TLSF will be able to reuse the previous block whereas Best-Fit will not. Besides, DLalloc and Best-fit allocators show worst results when small blocks are requested (Test 2). And finally, as can be seen, simulating the same load with a different seed in the simulator has outcome in almost the same results, that is a very low standard deviation.

On the other hand we have Binary-Buddy and Half-Fit, both of them with a fragmentation around 80% in the case of Half-Fit and 70% in Binary-Buddy. As expected, the high fragmentation caused by Binary-buddy is due to the excesive size round up (round up to power of two). All wasted memory of Binary-buddy is caused by internal fragmentation. Half-Fit's fragmentation was also expected because of its *incomplete memory use*. As can be seen, both allocators are quite sensitive request sizes that are not close to power of two, causing a high fragmentation, internal fragmenta-

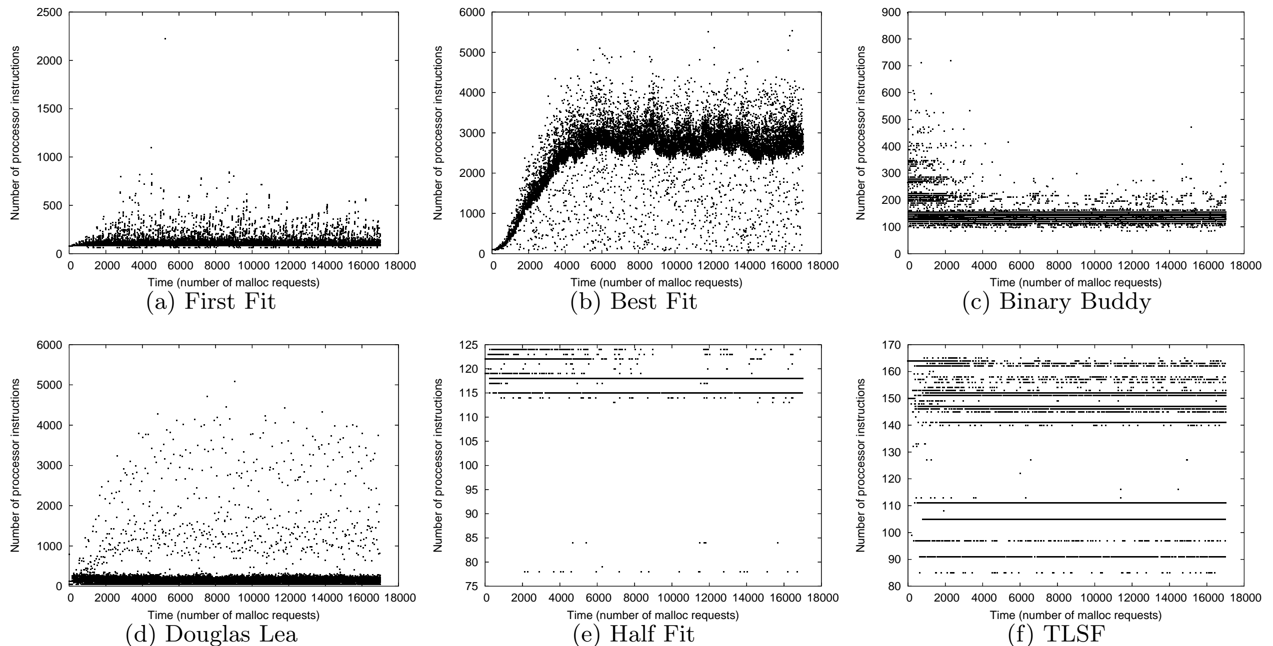

Figure 3: Malloc processor instructions (Test 1)

Table 7: Fragmentation results: Factor $\mathcal{F}$

	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	93.25	3.99	99.58	87.57	83.21	9.04	98.17	70.67	87.63	4.41	94.82	70.76
Best-fit	10.26	1.25	14.23	7.20	21.51	2.73	26.77	17.17	11.76	1.32	14.14	9.71
Binary-buddy	73.56	6.36	85.25	66.61	61.97	1.97	65.06	58.79	77.58	5.39	84.34	64.88
DLmalloc	10.11	1.55	12.90	7.39	17.13	2.07	21.75	14.71	11.79	1.39	13.72	9.90
Half-fit	84.67	3.02	90.07	80.40	71.50	3.44	75.45	65.02	98.14	3.12	104.67	94.21
TLSF	10.49	1.66	11.79	6.51	14.86	2.15	18.56	9.86	11.15	1.10	13.91	7.48

tion in the case of the Binary-Buddy and external one in the Half-Fit case.

First-Fit, which has been studied due to its relevance in the existing theoretical fragmentation analysis, presents the worst fragmentation in all the cases. First-Fit tends to split large blocks to satisfy small-size request, preventing its use for incoming request.

Figure 4 shows the evolution of the fragmentation with the number of mallocs requested. The lowest plot corresponds with the requested workload (blocks requested by tasks). As the previous table showed, TLSF, Best-fit and DLalloc have similar evolution. The higher fragmentation is obtained by Binary-buddy and Half-fit. In the plot we have represented the first 1400 of 1500000 mallocs of the simulation.

4.4 Analysis of the impact of delayed deallocation policy

The goal of Test 4 was to analyse the evolution of the fragmentation incurred by each allocator when tasks do not perform explicit deallocation. Deallocation is carried out by the *deallocation task* which free all block that are not in use. The fragmentation of the set of task with explicit deallocation (without deallocation task) is considered as the *reference task set*. It means that the fragmentation incurred when there is the deallocation task, is calculated as the point of the maximum memory used by the allocator in the *reference task set* relative to the point of the maximum amount of memory used by the load in the scenario with *deallocation task*. This approach permits to evaluate the impact of delayed deallocation in the fragmentation.

For each deallocation task period, we executed 50 times the task set with different seed. Average values of the memory used and maximum memory required were obtained. These results are shown in figure 5. The initial value of the fragmentation corresponds to the same scenario with explicit deallocation (*reference task set*). The range of the task periods varies from the shortest application task period to 10 times the largest period (in this case [45..1400]).

As shown in figure 5, TLSF, DLalloc and Best-fit work reasonably well increasing the framentation (measured by factor $\mathcal{F}$) from the initial value of 7% to 45%. We have included only these allocators because others present very high fragmentation.

5. CONCLUSIONS

TLSF is a “good-fit” dynamic storage allocator designed to meet real-time requirements. TLSF has been designed as a compromise between constant and fast response time

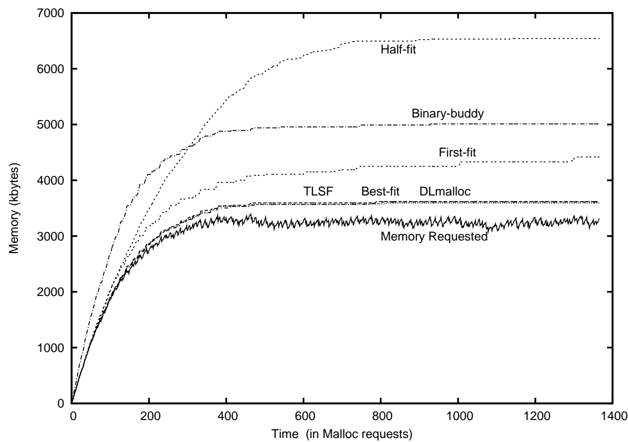

Figure 4: Memory used evolution during one of the simulations (Test 3)

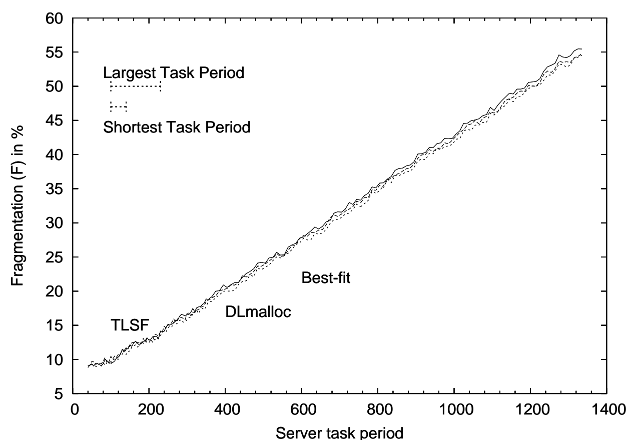

Figure 5: Fragmentation incurred when the deallocation task period changes (test4)

and efficient memory use (low fragmentation). In this paper we have compared it with other allocators under "real-time" workloads.

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

Our analysis also shows that allocators designed to optimise average response time by considering the usage pattern of conventional applications, such as DLalloc or Binary-buddy, are not suitable for real-time systems.

Since real-time applications are long running and the start-up and the shutdown is usually done during the non-critical phase of the system, the analysis has been focused on the stable phase designing experiments with very huge number of malloc operations. Also, the specific characteristics of real-time applications have be considered, including: periodic request patterns, limited amount of allocated memory per task, bounded holding time, etc.

From the fragmentation point of view, TLSF and DLmalloc present the best results followed by Best-fit. Binary-Buddy and Half-fit generate a very important fragmentation.

In summary, an allocator must fulfil two requirements in order to be used in real-time systems: 1) it must have a bounded response 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. From this point of view, TLSF achieves both requirements. Moreover a bounded response time, the average response time is as good as DLmalloc and Binary-buddy. All the code is available at: http://rtportal.upv.es/rtmalloc.

6. REFERENCES

[1] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, 2001.

[2] G. Bollella and J. Gosling. The real-time specification for java. *IEEE Computer*, 33(6):47–54, 2000.

[3] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[4] R. P. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Trans. Program. Lang. Syst.*, 11(3):388–403, 1989.

[5] K. Danne and M. Platzner. Memory demanding periodic real-time applications on fpga computers. In *WIP Session of the 17th Euromicro Conference on Real-Time Systems. Palma de Mallorca, Spain*, pages 65–68, 2005.

[6] S. Feizabadi, B. Ravindran, and E. D. Jensen. Msa: a memory-aware utility accrual scheduling algorithm. In *SAC*, pages 857–862, 2005.

[7] M. Garey, R. Graham, and J. Ullman. Worst Case Analysis of Memory Allocation Algorithms. In *Proceedings of the 4th Annual ACM Symposium on the Theory of Computing (STOC'72)*. ACM Press, 1972.

[8] D. Grunwald and B. Zorn. Customalloc: Efficient synthesized memory allocators. *Software Practice and Experience*, 23(8):851–869, 1993.

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

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

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

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

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

[14] A.-M. D. Pierre-Emmanuel Hladik, Hadrien Cambazard and N. Jussien. How to solve allocation problems with constraint programming. In *WIP Session of the 17th Euromicro Conference on Real-Time Systems. Palma de Mallorca, Spain*, pages 25–28, 2005.

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

[16] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *J. ACM*, 21(3):491–499, 1974.

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

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

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