Real-Time Performance of Dynamic Memory Allocation Algorithms*

Isabelle Puaut
INSA/IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France
e-mail: Isabelle.Puaut@irisa.fr, Tel: +33 2 99 84 73 10; Fax: +33 99 84 25 29

Abstract

Dynamic memory management is an important aspect of modern software engineering techniques. However, developers of real-time systems avoid using it because they fear that the worst-case execution time of the dynamic memory allocation routines is not bounded or is bounded with an excessively large bound. The degree to which this concern is valid is quantified in this paper, by giving detailed average and worst-case measurements of the timing performance of a comprehensive panel of dynamic memory allocators. For each allocator, we compare its worst-case behavior obtained analytically with the worst timing behavior observed by executing real and synthetic workloads, and with its average timing performance. The results provide a guideline to developers of real-time systems to choose whether to use dynamic memory management or not, and which dynamic allocation algorithm should be preferred from the viewpoint of predictability.

1. Introduction

Dynamic memory allocation has been a fundamental part of most computer systems since the sixties, and many different dynamic memory allocation algorithms have been developed to make dynamic memory allocation fast and memory-efficient [14] for a survey. A dynamic memory allocator must keep track of which parts of memory are in use and which parts are free. One of the problems the allocator must address is that the application program may allocate and free blocks of any size in any order, creating free blocks ("holes") amid used ones. If these holes are too numerous and small, they cannot be used to satisfy future requests for larger blocks (*fragmentation* problem). The goal when designing an allocator is usually to minimize this wasted space without undue time cost. The time and memory performance of dynamic memory allocators are very often evaluated using simulation, using either *real traces* [1, 7] or *synthetic* traces [5, 15]. Most efforts on the evaluation of dynamic memory allocators have focused on evaluating the *average* behavior of allocators, with respect to allocation times and wasted memory due to fragmentation.

While the average execution time of a program is suited as a performance measure for non real-time applications, different performance criteria apply for real-time systems. In *hard* real-time systems, it must be guaranteed that all tasks meet their deadlines. In such systems, the knowledge of tasks worst-case execution times (WCETs) is an essential input parameter of the schedulability analysis used to validate the system temporal correctness. Safe estimations of worst-case execution times of programs can be obtained using static analysis of their code [9]. In contrast, in *soft* real-time systems, a limited fraction of tasks may miss their deadlines. In such systems, the safety in the determination of tasks worst-case execution times is less stringent than in hard real-time systems, and a probabilistic knowledge of tasks worst-case execution times is sufficient.

Developers of real-time systems avoid the use of dynamic memory management because they fear that the worst-case execution time of dynamic memory allocation routines is not bounded or is bounded with an excessively large bound. While some studies have been undertaken to identify the worst-case memory needs of dynamic memory allocators despite fragmentation [10, 6], most work concerning the timing behavior of dynamic memory allocators has focused on optimizing their timing behavior in the *average* case. An exception to this rule is the work presented in [7]. This work gives detailed measurements of the worst-case allocation and deallocation times observed by executing a set of programs on a simulated architecture for different dynamic memory allocators. These figures are compared with average allocation/deallocation times. Compared to [7], the work presented in this paper does not restrict to worst-case allocation/deallocation times observed by *executing* real programs, but also gives worst-case figures obtained *analytically* (i.e. the worst possible allocation/deallocation times).

This paper gives detailed average and worst-case measurements of the timing performance of a comprehensive panel of dynamic memory allocators. The allocators studied are representative of the different classes of existing dynamic memory allocators (sequential fits, indexed fits, segregated fits, buddy systems). For every allocator, we com-

* 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, June 2002, pages 41-49

pare its worst-case behavior obtained analytically with (i) the worst timing behavior actually observed by *executing* real and synthetic workloads, (ii) its average timing performance. Both performance measures are given by executing the workload (real application, synthetic workload or worst-case workload) on a test platform, with careful attention put on the experimental conditions to be able to compare the real and synthetic workloads.

The main contributions of our work are: (i) an analytical and quantitative evaluation of the worst-case allocation/deallocation times for a given hardware platform; (ii) a quantification of the impact that could have the consideration of worst-case allocation/free times obtained analytically on the applications end-to-end execution time. The results given in this paper should provide a guideline to developers of real-time systems to choose whether to use dynamic memory management or not, and which dynamic allocation algorithm should be preferred for the viewpoint of predictability.

The remainder of this paper is structured as follows. Section 2 details the dynamic memory allocation algorithms we have studied. Section 3 describes the experimental conditions used for the performance evaluation. Section 4 details the experimental results. In section 5, we summarize the results so as to provide guidance to select an allocator under a set of constraints (e.g. heap size) and to integrate its timing costs into a schedulability analysis. We conclude in Section 6.

2. Dynamic Memory Allocation Algorithms

A wide variety of dynamic memory allocators have been implemented and used in computer systems [14]. They are typically categorized by the mechanisms they use for recording which areas of memory are free, and for merging adjacent free blocks into larger blocks (coalescing). The algorithms we have studied share a number of features (§ 2.1), but differ in the way the free blocks are identified and searched for: sequential fits (§ 2.2), indexed fits (§ 2.3), segregated fits (§ 2.4) and buddy systems (§ 2.5).

2.1. Common characteristics of the algorithms

The algorithms we have implemented share a set of common features:

– *Allocation in real memory*. Since we focus on real-time systems, we have restricted our study to algorithms that allocate areas of *real memory*, and assume that no address translation nor paging is used.
– *No block relocation* for memory defragmentation.
– *Immediate coalescing*. A block is merged with its neighboring free blocks as soon as it is freed.
– *Minimum block size*. The size of a small allocated block is rounded up to a minimum block size.
– *Splitting threshold*. Upon allocation of a block, when a free block larger than the requested size exists, it is split only if the remainder is larger than a known constant, named splitting threshold.

The values for the minimum block sizes and the splitting threshold are allocator-dependent. They depend on the size of the allocator data structures in the case they are stored in the free blocks. The values actually used are given in § 3.4.

Some of the algorithms we have implemented use *boundary tags* to coalesce free blocks into larger blocks. This technique, introduced by Knuth in [5], uses a header and a footer for every block indicating whether the block is free or not (the header is at the beginning of the block and the footer at the end). The size of the block is also stored in both header and footer. When a block is freed, coalescing can be achieved easily by looking at the header of the following block and the footer or the preceding block.

2.2. Sequential fits: first-fit and best-fit

Sequential fits are based on the use of a unique linear list of all the free blocks in memory, whatever their size is.

The first-fit and best-fit allocators we have implemented use a doubly-linked list to chain the free blocks. The pointers that implement the list of free blocks (free list) are embedded in the free blocks. The first-fit allocator searches the free list and selects the *first* block that is at least as large as the requested size, while the best-fit allocator selects the block that fits the request *best* (the block that generates the smallest remainder). Knuth's boundary tags (§ 2.1) are used for block coalescing. We have selected a LIFO strategy for inserting newly freed blocks in the free-list upon block deallocation for the first-fit allocator, and FIFO strategy for the best-fit allocator.

2.3. Indexed fits

Indexed fits are an alternative to sequential fits where the free blocks are linked together thanks to a data structure more sophisticated than the linear list used in the sequential fits. We have implemented two algorithms of this class.

Ordered binary tree best-fit. This allocator uses a binary tree ordered by block size to link free blocks with each other. There is one node in the tree per size of free blocks, containing a doubly-linked list of free blocks of that size. When a block allocation request is issued, the tree is traversed until a block with the best size (generating the smallest remainder) is found. The pointers used to implement the tree and lists data structures are embedded in the free blocks. By construction of the algorithm, the binary tree that chains the free blocks is not balanced. Knuth's boundary tags (§ 2.1) are used to coalesce blocks.

Fast-fit (Cartesian tree best-fit). The fast-fit allocation, introduced in [11], uses a Cartesian tree, sorted in both size and address. A Cartesian tree [12] encodes two-dimensional information in a binary tree, using two constraints on the tree shape. The tree is effectively sorted on a primary key and a secondary key. The tree nodes are totally-ordered with respect to the primary key (here, the addresses of free blocks). With respect to the secondary key (here, the sizes of free blocks), the tree is partially ordered with each node having a greater value than its descendants. This dual constraint limits the ability to re-balance the tree, because the shape of the tree is highly constrained by the dual indexing keys.

The Cartesian tree is used to implement an approximation of the best-fit policy: when allocating a block, the tree is traversed according to the "size" secondary key until the best free block large enough to satisfy the request is found. When a block is freed, the "address" primary key is used for block coalescing. The tree structure is embedded in the free blocks.

2.4. Segregated fits: quick-fit

The principle of segregated fits is to use separate data structures for free blocks of different sizes.

The segregated fits algorithm we have implemented, named *quick-fit* in the following, adapts the Gnu libc allocator to work in real memory only. It is based on the quick-fit allocator as introduced in [13]. This allocator is an hybrid algorithm in the sense that it uses two different allocation algorithms depending on the size of the requested block.

For allocation of a block larger than a page (a page is a fixed-sized chunk of memory, here 4KB), the allocator uses a next-fit allocation strategy. Next-fit is a common optimization of first-fit which does not scan the free-list from the beginning at every allocation. Instead, the scan is started from the position where the last search in the free-list was satisfied. The size granted for large blocks is rounded up to the upper integral number of pages. There is one header per page, used to implement the next-fit allocation policy (the free-list is an external data structure).

Blocks smaller than a page are allocated from page-sized chunks, within which all blocks are the same size (powers of two). The page header maintains a count of the number of free blocks within the page as well as a doubly-linked list of all free blocks. When the last free block of a page is freed, the page is immediately freed.

2.5. Binary and Fibonacci buddy systems

In buddy systems, the heap is recursively divided up into smaller and smaller blocks. At every level of the hierarchy, there are two possible blocks, which are further subdivided into two blocks and so on. When an allocation request is made, the requested size is rounded up to the closest possible block size in the hierarchy of blocks. A block's *buddy* is the other block at the same level of the hierarchy. When a block is freed, it can only be coalesced with its buddy. Note that the buddy is considered allocated as soon as there is an allocated block within it.

We have implemented two allocators in the class of buddy systems: a *binary buddy* allocator, in which all block sizes are a power of two, and a *Fibonacci buddy* allocator, in which block sizes are members of a Fibonacci series. Both allocators use a doubly-linked list of free blocks for every legal block size (e.g. powers of two for the binary buddy allocator). The pointers implementing the list are embedded in the free blocks.

3. Experiment Description

In this section, we give information concerning the workloads used to evaluate the performance of the dynamic memory allocators: the workload exhibiting the worst-case allocation/deallocation times (§ 3.1), real applications (§ 3.2) and a synthetic workload (§ 3.3). The measurement method used for the performance evaluation is given in § 3.4.

3.1. Worst-case allocation/deallocation workloads

Identifying the worst-case execution times of dynamic allocation algorithms can be achieved either by using static WCET analysis [9], which returns an upper bound on the time required to execute a program using static analysis of its source code, or by worst-case complexity analysis of the algorithm. Using static WCET analysis on memory allocation algorithms is not possible without having an in-depth knowledge of the allocation algorithms, because the time required to allocate a block does not only depends on the parameters of the memory allocation routines. It also depends on the allocator internal state (e.g. free lists), which itself depends on the history of the past allocation requests, which is in general unknown statically.

Let us consider for instance the first-fit allocator. The worst-case allocation time is obtained for the longest size of the free list. The actual size of the free list is unknown, unless the history of allocations/deallocations is known, which is not the case in general. However, knowing the minimum block size M and the heap size H, and noticing that the longest free list is obtained when the heap alternates between free blocks and busy blocks of size M, we see that the maximum length for the free list is $\frac{H}{2M}$, which allows us to bound the duration of a block allocation for the first-fit algorithm. We see from this example that the WCET analysis techniques that use flow analysis to obtain automatically loop bounds (e.g. [3]) are not able to derive automatically the worst-case number of iterations for this algorithm.

As a consequence, for all dynamic memory allocators studied, the worst-case allocation and deallocation times have been obtained through a worst-case complexity analysis of the allocation algorithms instead of using static WCET analysis. In the following, we present the worst-case behavior of the allocators studied in an informal manner (giving formal proofs of such worst-case behaviors is outside the scope of this paper).

To obtain quantitative data of the worst-case execution times of memory allocation/deallocation, we have *measured* the execution times of the allocation/deallocation on workloads that are representative of the worst-case scenarios, described below (§ 3.1.1 and § 3.1.2).

3.1.1 Worst-case allocation scenarios

The worst-case allocation scenarios for the algorithms studied are the following (in the text, H denotes the heap size):

Sequential fits. The worst time taken to allocate a block is when the free list has the maximum size, i.e. when the heap alternates between busy and free blocks of minimum size M. The maximum number of pointer traversals is then $\frac{H}{2M}$.

Indexed-fits. For the binary tree best fit, the worst time for block allocation is when the tree that chains the free blocks is unbalanced and has the longest possible length, which occurs when the heap is fragmented. Since free blocks of identical sizes are linked in a single node of the tree, the maximum length of the chain is when the memory alternates between busy blocks of the minimum size M and free blocks of every possible size. The maximum number of node traversals is then $\lceil\sqrt{c + \frac{2H}{M}}\rceil$, with c a constant. A similar worst-case allocation scenario occurs for the fast-fit allocator, the difference being that free blocks of identical sizes are chained into different tree nodes since they have a different address. The worst-case number of node traversals is then identical to the sequential-fits ($\frac{H}{2M}$).

Segregated-fits. The worst-case allocation time for the quick-fit allocator (with H = 4 MB) occurs when an allocation of the smallest possible size is requested and the corresponding free list is empty. In this situation, a page has to be allocated using the next-fit allocation strategy. It is then split into blocks of the smallest size, and the corresponding free list is initialized. More generally, for any heap size H and page size P the worst-case number of node traversals is $max(\frac{P}{M}, \frac{H}{P})$. The first term corresponds to the allocation of the smallest possible block in a page, and the second term corresponds to the allocation of a large block (set of pages).

Buddy systems. The worst-case allocation time for buddy systems is when the heap is initially entirely empty and an allocation of the smallest possible size M is requested. In this situation, the free lists for all possible sizes have to be updated. The number of lists updates is $\lceil log_2(\frac{H}{M})\rceil$ for the binary buddy and $\lceil log_\phi(\frac{H}{M})\rceil$ for the Fibonacci buddy, where ϕ is the golden number $\frac{1+\sqrt{5}}{2}$.

3.1.2 Worst-case deallocation scenarios

The worst-case deallocation scenarios for the algorithms studied are sketched below.

Sequential fits. Due to the use of boundary tags for block coalescing, the worst situation that can occur is when the two neighboring blocks to be freed are free, causing the three blocks to be merged into a single big block that is inserted in the free list.

Indexed-fits. For the binary tree best fit, the worst time taken to free a node is when its two neighboring nodes are free. Checking whether neighboring nodes are free is achieved in constant time due to the use of boundary tags. However, block merging needs tree reorganizations to keep the free blocks sorted according to their sizes. A similar worst-case deallocation scenario occurs for the fast-fit allocator, except for the identification of neighboring free nodes, that uses the Cartesian tree structure instead of the boundary tags.

Segregated-fits. The worst time taken to free a block for the quick-fit allocator for a 4MB heap is when the last fragment of a page is freed, causing the page to be freed and coalesced with the neighboring pages.

Buddy systems. The worst-case deallocation time for the buddy systems allocators occurs when a block of the smallest possible size is freed in a heap in which only this block is allocated.

3.2. Real applications

Since designers of real-time systems avoid the use of dynamic memory management, it's very hard to find task sets that are both representative of actual real-time workloads and use dynamic memory management. In this work, we have used three applications, the first one being a soft-real time application and the two other ones[1] being non real-time applications used in a previous work on dynamic memory management [7].

mpg123. mpg123 [4] is a player of MPEG level 3 streams. It has been executed with a 128Kbps audio stream as input.

Cfrac. Cfrac is a program to factor large integers using the continued fraction method. Input was the 22 digit number 1,000,000,001,930,000,000,057, which is the product of two primes.

Espr. Espr is a logic optimization program. The *largest* file, distributed with the release code, was used as input to the program.

A summary of the application characteristics is given in Table 1. The last two columns of the table give respectively

[1] Downloaded from ftp://ftp.cs.colorado.edu, directory /pub/cs/misc/malloc-benchmarks

Appli	# alloc	# free	largest alloc (B)	mean alloc (bytes)	Inter-arrival (cycles)	Alloc/free time (% of total, quick-fit)
mpg123	5987	5981	65656	143	8852358	0.20
cfrac	227091	227087	266	18	26608	35.86
espr	25585	24757	4608	41	60947	12.18
synthetic	25585	25585	345	40	60952	10.80

Table 1. Workload characteristics

	mgp123		cfrac		espr		synthetic		analytical
	worst	mean	worst	mean	worst	mean	worst	mean	worst
first-fit	6316	6283	13936	6482	22856	7349	93646	16647	61769847
best-fit	6790	6766	11060	7615	26010	13727	54265	28136	64410985
btree-best-fit	8770	8733	19279	8284	34300	12678	47890	11657	898391
fast-fit	6619	6550	32449	11277	62656	12240	53091	18438	62919925
quick-fit	66412	10752	248634	5735	248581	4327	244411	4212	275550
buddy-bin	43883	6583	65145	5785	62046	6376	62748	4830	66696
buddy-fibo	36994	8966	51567	7361	51807	7689	48310	5455	55090

Table 2. Worst-case and average performance of memory allocation (processor cycles)

the mean delay between allocation requests and the mean time spent performing dynamic memory management when using the quick-fit allocator.

3.3. Synthetic workload

The synthetic workload application we have used is the MEAN model of memory allocation evaluated in [15]. This simple model characterizes the behavior of a program by computing three means: the mean block size, the mean block lifetime and the mean time between successive allocation requests. These three values are then used to generate random values according to an exponential distribution. This model is slightly less accurate than other models of memory allocation [15], but performs well on relatively simple allocation algorithms and requires much less coding effort and storage than more sophisticated models. For the experiments, the model parameters have been selected to mimic the behavior of the Espr application, as shown in Table 1.

3.4. Measurement method

Measurements were conducted on a Pentium 90 MHz machine. The code is loaded and monitored using the Pentane tool, which allows the non-intrusive monitoring of applications (there is no activity other than the module to be tested). Pentane provides control over the hardware (e.g. enabling/disabling of caches), and allows to count the number of occurrences of a number of events (e.g. number of processor cycles). All the timing measures given in the next section are expressed in number of processor cycles. In order to be able to compare the timing of memory allocation using different workloads having very different locality properties (e.g. a real workload with synthetic workload that do not actually executes application code), all performance enabling features (instruction and data caches, branch prediction, super-scalar execution) were disabled.

An initially empty 4 MB heap is used in the experiments. The smallest value for minimum block size and splitting threshold possible for all allocators (16 bytes) is used.

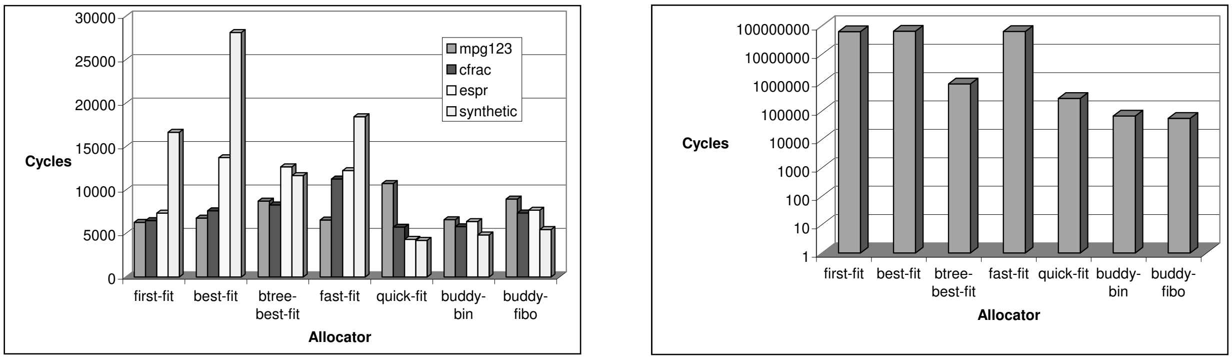
4. Results

4.1. Real-time performance of memory allocation

Table 2 gives for the three classes of workloads (real, synthetic and worst-case) the allocation times measured. For the real and synthetic workloads, table 2 gives both the average and worst measured allocation times. The remainder of this section is devoted to a detailed analysis of the contents of table 2, which is illustrated by a set of diagrams built from the contents of the table.

Subfigure 1(a) depicts the average allocation times measured during the execution of the real and the synthetic workloads. The allocator that exhibits the best average performance is the quick-fit allocator, followed by the binary buddy and the first-fit allocator. It can be noted that the quick-fit allocator has modest average performance for the mpg123 application. This is because mpg123 allocates and frees a small block every time a frame is decoded. This causes a page to be split into fragments at allocation time, and the same page to be freed at deallocation time, which has an important time overhead. Deferred coalescing would be required to improve the average allocation time of such an allocator. Also remark that most allocators perform better for real applications than on the synthetic workload. The reason is that the number of different sizes allocated by the synthetic workload is greater than the one of real applications, thus causing more fragmentation and increasing the average allocation time.

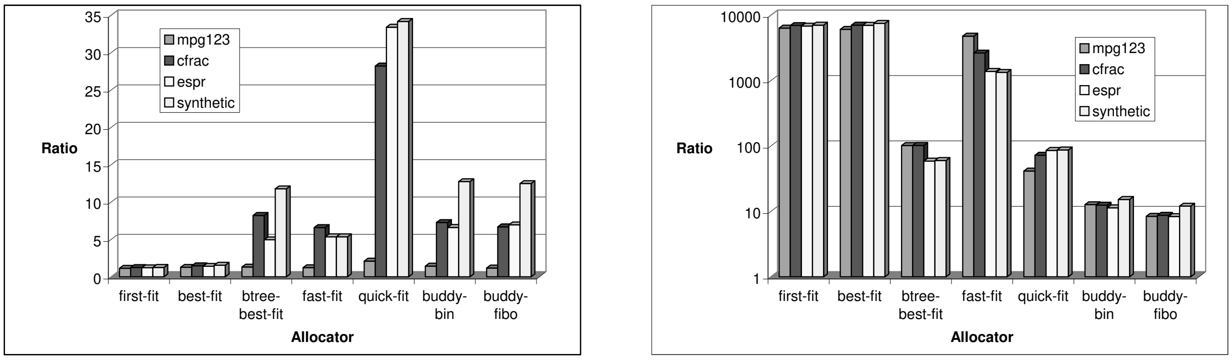
Subfigure 1(b) depicts the worst-case allocation times (obtained analytically) for the allocators studied. The worst



(a) Average allocation times

(b) Worst-case allocation times obtained analytically

Figure 1. Average vs worst-case allocation times



(a) Worst-case actually observed

(b) Worst-case allocation times obtained analytically

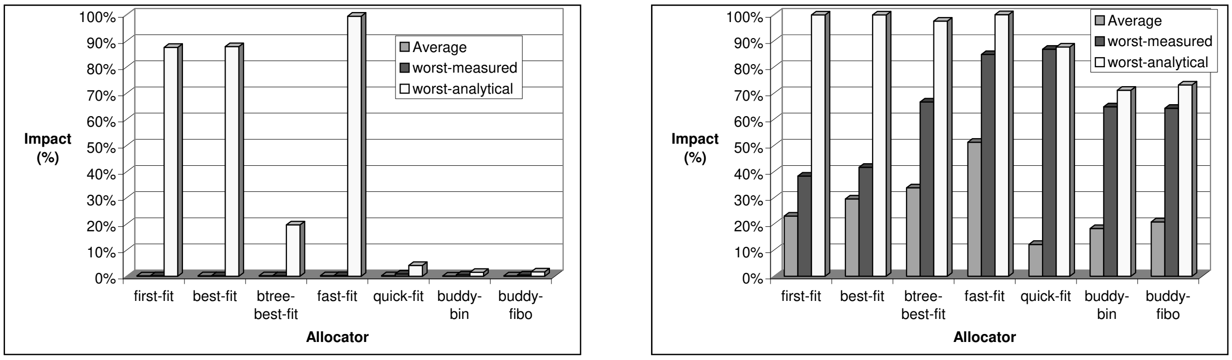
Figure 2. Ratio between worst and average allocation times

time required for block allocation in sequential fits is very large, because at worst, the free-list covers the entire memory and has to be scanned entirely to find a free block (see § 3.1). Sequential-fits, while exhibiting reasonably good average performance, have very poor worst-case performance. In contrast, buddy systems, that have a worse average timing behavior than sequential fits, have a much better worst-case behavior.

Figure 2 depicts the ratio between the worst-case allocation times and average allocation times for the allocators studied. In the left part of the figure, the worst-case allocation times used to compute the ratio are the worst allocation time *actually observed* while executing the real and synthetic workload, while in the right part of the figure, the worst-case allocation times are those obtained *analytically* (see § 3.1).

The ratio between actually observed worst-case allocation times and worst-case allocation times (subfigure 2(a)) is rather small (less than six) for most allocators, whatever workload is executed. When analytical worst-case allocation times are used instead of actually measured worst-case allocation times (subfigure 2(b)), the ratio becomes much larger, especially for the sequential-fits and the fast-fit allocators. However, this ratio stays reasonable for some allocators like the buddy systems, for which the ratio is lower than ten.

An important issue to address when using dynamic memory allocation in real-time systems, especially in hard real-time systems in which all deadlines must be met, is to select the upper bound of the time for memory allocation/deallocation to be used in the system schedulability analysis. As said in the introduction, the average allocation time is clearly not appropriate because it underestimates the time required for block allocation, as shown in figure 2. Two upper bounds can then be selected: the worst-case measured allocation times on a specific application, or the worst-case allocation times identified analytically. Figure 3 examines the impact of the selection of one of these two upper bounds on the application execution time for mpg123 and Espr applications (subfigures 3(a) and 3(b)). The figure depicts the



(a) Mpg123 workload

(b) Espr workload

Figure 3. Impact of worst-case allocation times on application execution time

	mgp123		cfrac		espr		synthetic		analytical
	worst	mean	worst	mean	worst	mean	worst	mean	worst
first-fit	11073	9720	11049	8884	10929	9060	10963	8783	10958
best-fit	13419	10635	13481	9139	12878	9198	13269	8581	13567
btree-best-fit	11487	8844	72283	8798	75570	15279	176274	14983	1262860
fast-fit	16092	13153	156900	23854	243349	45495	252109	47150	1467178556
quick-fit	13953	6666	107468	3810	107478	3219	108196	3168	108177
buddy-bin	7524	5261	38959	5364	38861	5887	55711	4377	68715
buddy-fibo	7729	6549	42122	6305	45874	6602	57085	4576	93822

Table 3. Worst-case and average performance of memory deallocation (processor cycles)

percentage of the workload execution time spent allocating and freeing memory, with three durations for block allocation and deallocation requests: *average* times; (ii) worst-case times measured on the application and (iii) worst-case times obtained analytically.

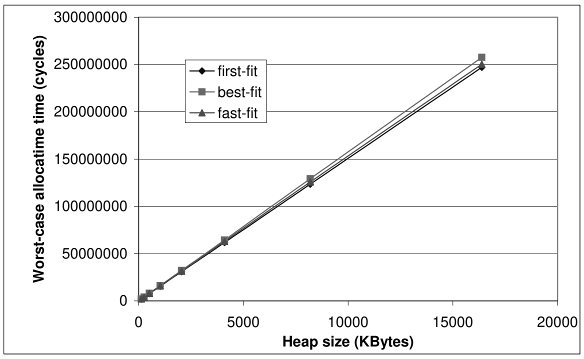
A first observation from the figure is that when using actually measured worst-case allocation times, their impact on the workload execution times is reasonable for almost all allocators. This value can then be used by the system schedulability analysis without introducing excessive pessimism, as far as the measurement conditions used to obtain the allocation times do not lead to allocation times lower than those that could be measured on other executions of the application. Such unsafe measurements can come for instance from the application control-flow that varies from one execution to another (for instance because of different input data), causing potentially different orders of memory allocation requests. Also note that the worst-measured allocation times are application-dependent, in contrast to the analytically obtained worst-case allocation time. Thus they should be determined every time a different application is built and only used on a per-application basis.

If we now consider analytically-obtained worst-case allocation times, one conclusion is that using them, albeit safe and application-independent, should not be used for the sequential-fit allocators. For instance, using them to analyze the schedulability of the mpg123 application, which is not allocation intensive, would be equivalent to consider that more than 90% of the workload execution time is devoted to memory allocation. We can also observe that the impact of the choice of an upper bound for dynamic memory allocation is influenced by the type of application executed: the impact on the Espr workload is always higher than the one on the mpg123 application, because in the former workload, memory allocation requests are more frequent and more varied in size than in the latter. An interesting result is that for some allocators, like for instance the buddy systems and the quick-fit allocators, considering the worst possible allocation time is realistic, at least for non memory intensive applications like mpg123 and the selected heap size (4 MB).

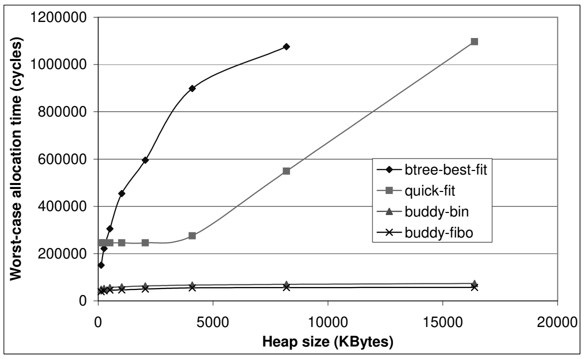
4.2. Real-time performance of memory deallocation

Table 3 gives the average and worst-case deallocation times measured on the real and synthetic workloads, as well as the worst-case deallocation times obtained analytically.

One can notice the very long worst-case deallocation times obtained analytically for the indexed-fits allocators, that come from the tree restructuring operations that occur at block deallocations (see § 3.1). Another remark is that the allocators that use boundary tags for block coalescing (sequential-fits) have low and predictable deallocation times. This block merging technique, while space consuming, ex-



(a) Sequential-fits and fast-fit



(b) Other allocators

Figure 4. Evolution of worst-case allocation times with heap size

hibits low and predictable deallocation times. Buddy systems also have reasonable worst-case deallocation times.

5. Discussion

We classify below the studied allocators with respect to their worst-case allocation times identified analytically in section 3.1, and examine how their worst-case allocation time varies with the heap size (§ 5.1). Then, we discuss which real-time bound should be used in which context (hard or soft real-time system) (§ 5.2).

5.1. Impact of heap size

The algorithms we have studied can be classified in four categories with respect to their worst-case allocation times obtained analytically (§ 3.1):

– sequential fits and fast-fit, for which the worst time required to allocate a block is linear with the number of free blocks, which at worst increases linearly with the heap size.
– the binary tree best fit allocator, for which the worst-case allocation time increases as the square root of the heap size.
– buddy systems, for which the worst-case allocation time is proportional to the number of different block sizes, which increases logarithmically with the heap size.
– quick-fit allocator, which has a different worst-case allocation behavior depending on the heap size: for small heaps, the worst-case allocation time is when a page is fragmented into the smallest block size; for large heaps, the worst-case allocation time is when the list of pages has to be scanned entirely to find a free block.

Figure 4 confirms this classification with measured values of the worst-case allocation times for different heap sizes, ranging from 128 KBytes to 16 MBytes. Because of the large variety of worst-case allocation times, the algorithms are divided into two subfigures: subfigure 4(a), for sequential fits and fast fit, for which the worst-case allocation time increases linearly with the heap size ; and subfigure 4(b) for the other algorithms. We see on the figure that the allocator with the lower worst-case allocation time is different for distinct heap sizes. The heap size is thus an important parameter in the process of selecting a suitable allocator according to its worst-case timing behavior.

5.2. Which metric for real-time performance of allocators ? In which context ?

The following table summarizes the pros and cons of using each of the two metrics of real-time performance evaluated in this paper, and the context of use for which they are best suited.

	(a) Worst-case obtained analytically	(b) Worst-case measured on specific application
Pros	Safety Context-independent	Reduced pessimism compared to (a)
Cons	Pessimism	Context-dependent Safety conditioned by the representativity of testing conditions
Context of use	Hard real-time systems	Hard real-time systems (if testing conditions correct) Soft real-time systems

The clear advantage of using the worst-case execution time obtained analytically is that it is safe and context-independent (it is an upper bound on the allocation times for any application). However, as shown earlier in the paper, the values obtained can be very high depending on the allocator and the heap size. Thus, the best suited context of use of such a value is the one of hard real-time systems, in which safety primes over pessimism reduction.

In contrast, using actually measured worst-case allocation times yields to less important values than the worst-case allocation times obtained analytically. However, the obtained value is context-dependent; in particular, it depends on the allocation pattern of the application. In addition, it is safe only if the measurement conditions actually lead to the worst-case allocation/deallocation scenario for this application. For instance, if the execution of the application is not deterministic and thus can lead to different traces of memory allocation requests for its different executions, the worst-case allocation time of a given execution is not necessarily the worst-case allocation time of any execution.

Instead of using one of the two above performance metrics, one could consider using a probabilistic real-time bound m together with a confidence level on this bound indicating the likelihood that the worst-case execution time is greater or equal to m [8, 2]. Establishing such a probabilistic real-time bound requires in particular that the different tests of execution times be independent [2]. Generating independent tests on the kind of algorithms studied in this paper seems to us practically difficult: the execution time of the algorithms depends not only on the allocation requests parameters but also on the system state (e.g. contents of the allocator data structures), the latter being difficult in practice to generate in a random manner.

6. Conclusion

This paper has given detailed average and worst-case timing analysis of a comprehensive panel of dynamic memory allocators. For every allocator, we have compared its worst-case behavior obtained analytically with the worst timing behavior observed by executing real and synthetic workloads and its average timing performance. A result of our performance analysis is the identification of a discrepancy between the allocators with the best average performance and the allocators with the best worst-case performance. We have shown that for applications with low allocation rates, the worst-case allocation/deallocation times obtained analytically can be used without an excessive impact on the application execution times for the most predictable allocators (buddy systems and quick-fit). We have also examined the impact of the heap size on the worst-case behavior of the memory allocators, and discussed the context in which analytical and actually measured worst-case allocation times are best suited.

Our work can be extended in several directions. First, it would be interesting to study the impact of parameters such as the minimum block size, splitting threshold, distribution of the sizes of allocated blocks, on the allocators timing performance. Another important issue to work on is to find the best trade-off between time and memory consumption, the most predictable allocators not being necessarily the most space-efficient. The impact of hardware performance enabling features (e.g. caching) on the worst-case performance of the allocators has also to be considered. Finally, we are currently studying the use of optimization methods such as genetic algorithms to automatically obtain approximations of the allocators worst-case execution times.

Acknowledgments. The author would like to thank M. Bellengé and K. Perros for their work on the performance measurements, as well as D. Decotigny (IRISA), P. Chevochot (Sogitec), and G. Bernat (Univ. of York) for their comments on earlier drafts of this paper.

References

[1] A. Diwan, D. Tarditi, and E. Moss. Memory system performance of programs with intensive heap allocation. *ACM Transaction on Computer Systems*, 13(3):244–273, Aug. 1995.

[2] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, pages 215–224, London, UK, Dec. 2001.

[3] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of executing time. In *Proc. Euro-Par'97 Parallel Processing*, volume 986 of *Lecture Notes in Computer Sciences*, pages 1298–1307. Springer-Verlag, Aug. 1997.

[4] M. Hipp. Mpg123 real time mpeg audio player (mpg123 0.59r). Free MPEG audio player available at http://www-ti.informatik.uni-tuebingen.de/~hippm/mpg123.html, 1999.

[5] D. E. Knuth. *The Art of Computer Programming*, chapter Information structures (chapter 2 of volume 1), pages 228–463. Addison–Wesley, 1973.

[6] E. L. Lloyd and M. C. Loui. On the worst case performance of buddy systems. *Acta Informatica*, 22:451–473, 1985.

[7] K. Nilsen and H. Gao. The real-time behavior of dynamic memory management in C++. In *Proceedings of the 1995 Real-Time technology and Applications Symposium*, pages 142–153, Chicago, Illinois, June 1995.

[8] P. Puschner and A. Burns. Time-constrained sorting - a comparison of different algorithms. In *Proc. of the 11th Euromicro Conference on Real-Time Systems*, pages 78–85, York, UK, June 1999.

[9] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

[10] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 1975.

[11] C. J. Stephenson. Fast fits: new methods for dynamic storage allocation. In *Proc. of the 9th Symposium on Operating Systems Principles*, pages 30–32, Betton Woods, New Hampshire, Oct. 1983.

[12] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, Apr. 1980.

[13] C. B. Weinstock and W. A. Wulf. Quickfit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–144, Oct. 1988.

[14] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Sciences*, Kinross, UK, 1995.

[15] B. Zorn and D. Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation*, 4(1):107–131, Jan. 1994.