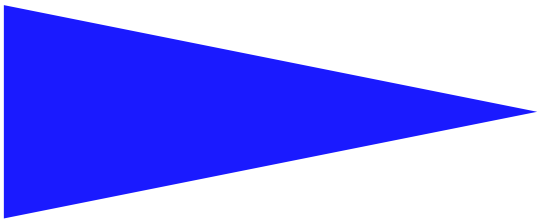
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1429



REAL-TIME PERFORMANCE OF DYNAMIC MEMORY ALLOCATION ALGORITHMS

ISABELLE PUAUT

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Real-Time Performance of Dynamic Memory Allocation Algorithms

Isabelle Puaut

Thème 1 — Réseaux et systèmes
Projet Solidor

Publication interne n° 1429 — Janvier 2002 — 21 pages

Abstract: Dynamic memory management is an important aspect of modern software engineering techniques. However, developers of real-time systems avoid to use it because they fear that the worst-case execution time of the dynamic memory allocation routines is not bounded or is bounded with a too important 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 performances of a comprehensive panel of dynamic memory allocators (sequential fits, indexed fits, segregated fits, buddy systems). For each allocator, we compare its worst-case behavior obtained analytically, with (i) the worst timing behavior observed by executing real and synthetic workloads, (ii) its average timing performance. We also quantify on a real workload (a MPEG player) the impact of the allocators worst-case allocation times on the workload end-to-end execution time. 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.

Key-words: Real-time systems, dynamic memory management, worst-case execution time, performance evaluation

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Performance Temps-réel d'Algorithmes d'Allocation Dynamique de Mémoire

Résumé : La gestion dynamique de la mémoire est un élément important dans les méthodes modernes de développement des logiciels. Toutefois, les concepteurs de systèmes temps-réel évitent d'allouer dynamiquement de la mémoire, car ils craignent que le pire temps pour allouer/libérer de la mémoire ne soit pas borné ou que la borne soit trop importante. Nous nous intéressons dans cet article à évaluer dans quelle mesure une telle crainte est justifiée, en évaluant les temps d'allocation et libération (en moyenne et au pire) d'un large éventail d'allocateurs dynamique de mémoire. Pour chaque allocateur mémoire, nous comparons son pire comportement temporel obtenu par analyse de l'algorithme avec (i) son pire comportement effectivement observé sur des applications de charge (synthétique et réelle) (ii) son comportement temporel moyen. Nous examinons également sur une application réelle (un décodeur MPEG) l'impact des temps pire cas de gestion dynamique de la mémoire sur les temps d'exécution de l'application. Les résultats présentés dans cette article fournissent des guides aux concepteurs de systèmes temps-réel pour savoir s'ils doivent ou non utiliser de l'allocation dynamique de mémoire, et quel algorithme de gestion dynamique de mémoire est le plus adapté de par son comportement temporel.

Mots clés : Systèmes temps-réel, gestion dynamique de mémoire, temps d'exécution au pire cas, évaluation de performance

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 (see [WJNB95] 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. This problem is known as *fragmentation*. The goal when designing an allocator is usually to minimize this wasted space without undue time cost, or vice versa. The time and memory performance of dynamic memory allocators are very often evaluated using simulation, using either *real traces* [DTM95, NG95, Joh97] or *synthetic* traces [Knu73, ZG94]. 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 the construction of 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 highly recommended, so that schedulability analysis methods can be used safely to validate the system temporal correctness. Safe estimations of worst-case execution times of programs can be obtained using static analysis of their source code (see [PB00] for a state of the art of the research on WCET analysis). 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 a too important bound. While some studies have been undertaken to identify the worst-case memory needs of dynamic memory allocators despite fragmentation [Rob74, Rob75, LL85], most work concerning the timing behavior of dynamic memory allocation has focused on optimizing the allocators timing behavior in the *average* case.

An exception to this rule is the work presented in [NG95]. 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 [NG95], 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). We also quantify on a real workload (a MPEG player) the impact of the allocators worst-case allocation times on the workload end-to-end execution time.

Paper contents and contributions

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 compare 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 obtain precise timing values and to be able to compare the real and synthetic workloads.

The main contributions of our work are: (i) a analytical and quantitative evaluation of the worst-case allocation/deallocation times for a given hardware platform; (ii) a quantification of the impact of the worst-case allocation/deallocation times on the execution time of a given application.

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 as far as predictability is concerned.

Paper organization

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 according to which the real-time performance of these algorithms have been obtained. 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 cost into a schedulability analysis. Finally, we conclude in section 6.

2 Dynamic Memory Allocation Algorithms Studied

A wide variety of dynamic memory allocators have been implemented and used in computer systems (see [WJNB95] for an overview). 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 implemented share a number of features (§2.1), but differ by 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). A summary of the allocators characteristics is given in § 2.6.

2.1 Common characteristics of the algorithms implemented

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*. We assume that programs share the same real memory space and that no address translation nor paging exists. There is thus no possibility to request extra memory from the operating system by fixed-sized pages (e.g. sbrk).
- *No block relocation*, to compact memory while a block is in use.
- *Immediate coalescing.* A block is merged with neighboring free blocks as soon as it is freed.
- *Alignment constraints.* All allocated memory is aligned on 2-bytes boundaries.
- *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. Otherwise, the entire block is allocated, thus creating internal fragmentation.

The values for the minimum block sizes, the alignment 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 (e.g. 8 bytes for a doubly-linked list). The values actually used are given in table 1, § 2.6.

Some of the algorithms we have implemented use *boundary tags* to coalesce free blocks into larger blocks. This technique, introduced by Knuth in [Knu73] 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 footer or the preceding block. In the case of coalescing with the preceding block, the size information in the footer is used to locate the beginning of the block, and the two blocks are merged by adding their sizes together. The following block can be coalesced in the same manner. Note that since blocks are aligned on 2 bytes boundaries, we use the least significant bit of the block size for the free/busy flag.

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) is 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 smaller 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 more sophisticated data structure than the linear list used in sequential fits. We have implemented two algorithms of this class.

2.3.1 Ordered binary tree best-fit

This allocator uses a self-adjusting binary tree ordered by block size to link free blocks with each others. 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 made, the tree is traversed until a block with the best size (generating the smaller 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 chain the free blocks is not balanced: in average, the finding a free block in the tree is logarithmic with respect to the number of free blocks, but at worst is linear with the number of free blocks.

Knuth's boundary tags (§ 2.1) are used to coalesce blocks.

2.3.2 Fast-fit (Cartesian tree best-fit)

The fast-fit allocation, introduced in [Ste83], uses a Cartesian tree, sorted in both size and address. A Cartesian tree [Vui80] 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 is a normal totally-ordered tree 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 which 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: the tree is searched 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, that we name *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 [WW88]. This allocator is an hybrid algorithm in the sense that it uses two different allocation algorithms depending on the size of the requested memory.

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 not stored in the free blocks but is rather 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 Buddy systems: 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 if there are any allocated blocks within it (there may be only a small portion of the buddy being used, but it must be still considered allocated for the purpose of coalescing).

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 serie.

Both allocators are implemented as follows. There is 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.

2.6 Summary of the allocators features

The main characteristics of the allocators studied are summarized in table 1.

3 Experiment Description

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

Name	Minimum block size	Splitting threshold	Description
first-fit	16 bytes	16 bytes	Doubly-linked list of free blocks managed in LIFO order. Returns the first free block in the list.
best-fit	16 bytes	16 bytes	Doubly-linked list of free blocks managed in FIFO order. Returns the best free block in the list.
btree-best-fit	10 bytes	10 bytes	Self-adjusting binary tree of free blocks sorted by size. Returns the best free block in the tree.
fast-fit	12 bytes	12 bytes	Cartesian tree of free blocks sorted by address and size. Returns the best free block in the tree.
quick-fit	8 bytes	8 bytes	Next-fit for large blocks, doubly-linked lists for small blocks.
buddy-bin	16 bytes	NA	Buddy system with power of two block sizes
buddy-fibo	16 bytes	NA	Buddy system with block sizes belonging to Fibonacci series

Table 1: Summary of allocators features

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 [PB00], which returns an upper bound of the time required to execute a program using static analysis of its source code, or by worst-case complexity analysis of the algorithm.

Using the first class of method 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 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 of size M 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 can see from this example that even the WCET analysis techniques that use flow analysis to obtain automatically loop bounds (e.g. [EG97]) are not able to derive the worst-case number of iterations for this algorithm automatically.

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 automatic methods such as 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 measures 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). The way these measures are obtained is detailed in section 3.4.

3.1.1 Worst-case allocation scenarios

The worst-case allocation scenarios for the algorithms studied are the following:

- For the first-fit and best-fit allocators, 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. The maximum number of pointer traversals is then $\frac{H}{2M}$, where H is the heap size and M is the minimum block size.
- For the binary tree best fit, the worst time taken for block allocation is when the tree that chains the free blocks is completely unbalanced and has the longest possible length, which occurs when the heap is fragmented. A similar worst-case allocation scenario occurs for the fast-fit allocator.
- The worst-case allocation time for the quick-fit allocator (with the selected heap size of 4 MB) occurs when an allocation of the smallest possible size (here 8 bytes) 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.
- The worst-case allocation times for the binary and Fibonacci buddies occurs when the heap is initially entirely empty and an allocation of the smallest possible size (here 16 bytes) is requested. In this situation, the free lists for all possible sizes have to be updated.

3.1.2 Worst-case deallocation scenarios

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

- The first-fit and best-fit allocations have a very similar behavior regarding memory deallocation. Due to the insertion policy in the free list for both allocators (LIFO for the first-fit allocator, and FIFO for the best-fit allocator), the insertion of a block in the free list is achieved in constant time. 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.
- 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 or not is achieved in constant time due to the use of boundary tags. However, when a freed block has to be merged with its two neighbors, three tree reorganizations have to be performed in order to keep the free blocks sorted according to their sizes and to keep the tree

balanced: two tree reorganizations to remove the tree nodes corresponding to the block neighbors, and one reorganization to insert the merged block. 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.

- The worst time taken to free a block for the quick-fit allocator is when the last fragment of a page is freed, causing the page to be freed and coalesced with the neighboring large free blocks.
- Finally, the worst-case deallocation times 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. Symmetrically to the allocation of the block, the free lists for all possible sizes have to be updated.

3.2 Real application

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. For instance, in [NG95], the worst measured memory allocation times are obtained on general purpose applications with no timing constraint.

In this work, we have used a soft real-time application: the mpg123 open-source MPEG audio player [Hip99], that plays audio MPEG level 3 streams. In the measurements, the application is started with a set of fixed parameters (in particular audio stream). The application has a very regular pattern regarding memory allocation: a set of data structures are allocated dynamically at application start, and are freed at application termination; a small set of data structures of fixed size are allocated and deallocated each time a frame is decoded. For this application, the mean allocated size is 143 bytes, the mean lifetime for the allocated blocks is 1.5 ms, the mean time between successive block allocation requests is 0.8 ms.

3.3 Synthetic workload

The synthetic workload application we have used is the MEAN model of memory allocation evaluated (among other models) in [ZG94]. This model, which is very simple, 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 uniform distribution with values ranging from zero to twice the mean. This model is slightly less accurate than other models of memory allocation (see [ZG94] for a performance comparison), but performs well on relatively simple allocation algorithms and requires much less coding effort and storage than more sophisticated models.

For the experiments, we have used the following parameters: a mean block size of 150 bytes, a ratio $\frac{block\ lifetime}{allocation\ inter-arrival}$ of 150, and an allocation inter-arrival of 500000 processor

cycles (this figure has been selected such that in average for the seven allocators, memory allocation requests consume one percent of the total execution time). Compared to the mpg123 application, the "mean size" parameter conforms to the mean block size allocated by the real application. In contrast, the synthetic workload uses a larger memory volume than the mpg123 application, and issues memory allocation requests more frequently.

3.4 Measurement method

Measurements were obtained by executing the mpg123 application, synthetic workload and worst-case workload on a Pentium 90 MHz machine.

The code is loaded and monitored using the Pentane tool[1]. Pentane allows the non-intrusive monitoring of applications (there is no activity other than the module to be tested - in particular no operating system). It provides control over the hardware (enabling/disabling of hardware features such as instruction/data caches, branch prediction); it allows to count the number of occurrences of a number of events (e.g. number of processor cycles, number of misses in the instruction cache, etc).

All the timing measures given in the next section are expressed in number of processor cycles. Moreover, 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.

A 4 MB heap is used in the experiments. It is entirely free at the beginning of the experiments.

4 Results

4.1 Real-time performance of memory allocation

Table 2 gives for the three workloads (mpg123, synthetic and worst-case) the allocation times measured. For the mpg123 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.

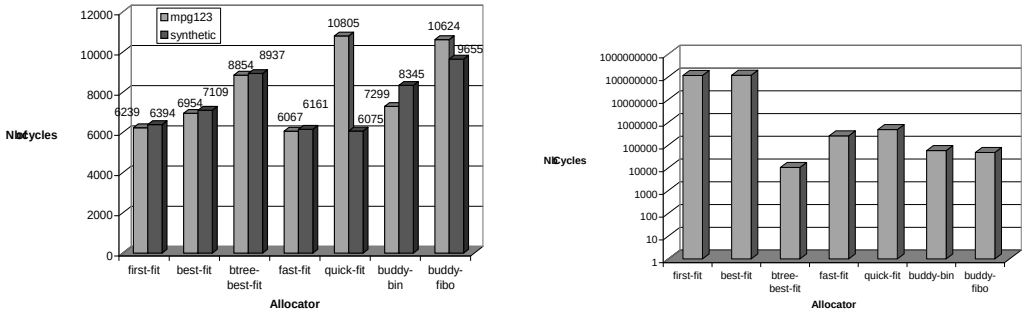
The left part of figure 1 depicts the average allocation times measured during the execution of the mpg123 application and the synthetic workload. The algorithms that exhibit the best average-case performance are the fast-fit and quick-fit allocators, followed by the sequential fits allocators (first-fit and best-fit). Note that the quick-fit allocator performs well in average for the synthetic workload and not for the mpg123 application. Actually, in the mpg123 application, for a given block size, there is an allocation shortly followed by a deallocation of this 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

[1] Available at http:/www.irisa.fr/solidor/work/hades.html

	mpg123-worst	mpg123-mean	synthetic-worst	synthetic-mean	analytical-worst
first-fit	6286	6239	6421	6394	115014229
best-fit	7012	6954	7144	7109	117573562
btree-best-fit	8872	8854	8984	8937	10414
fast-fit	6110	6067	6192	6161	256382
quick-fit	67081	10805	244529	6075	481267
buddy-bin	46167	7299	47041	8345	57827
buddy-fibo	39740	10624	41654	9655	47320

Table 2: Worst and average performance of memory allocation (nb. of processor cycles)

an important time overhead. Deferred coalescing would be required to improve the average allocation time of such an allocator.



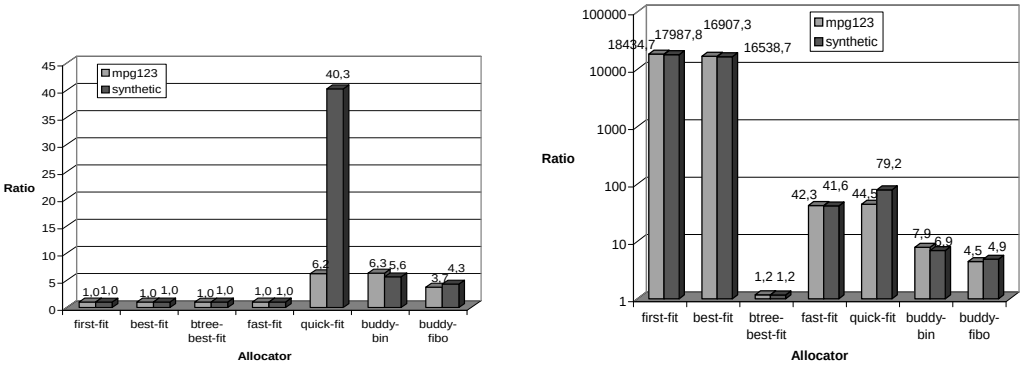
(a) Average allocation times

(b) Worst-case allocation times obtained analytically

Figure 1: Average vs worst-case allocation times

The right part of figure 1 depicts the worst-case allocation times (obtained analytically) for the allocators studied. The worst 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 a 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 seven 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 mpg123 and synthetic workload, while in the right part of the figure, the worst-case allocation times are those obtained *analytically* (see 3.1).



(a) Worst-case actually observed

(b) Worst-case allocation times obtained analytically

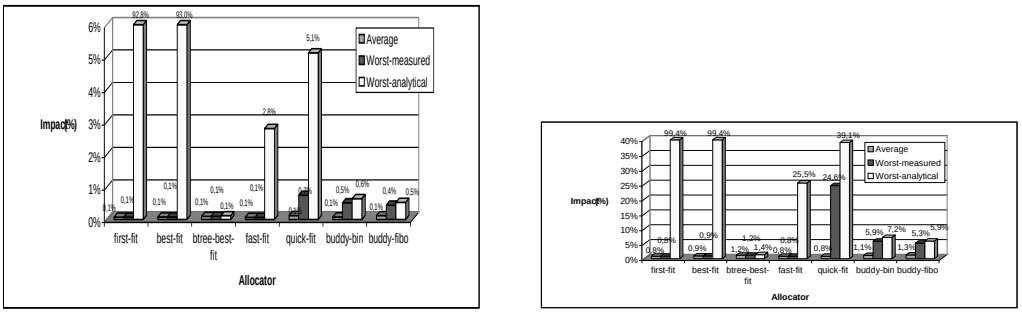
Figure 2: Ratio between worst and average allocation times

The ratio between actually observed worst-case allocation times and worst-case allocation times (left part of the figure) 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, the ratio becomes much larger, especially for sequential-fit allocators. However, this ratio stays reasonable for some allocators like the buddy systems, for which the ratio is around seven.

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. The latter bound is safe, but is often much higher than the former, as shown in figure 2.

Figure 3 examines the impact of the selection of one of these two upper bounds on the application execution time for mpg123 and the synthetic workload. What is depicted in the figure is the percentage of the workload execution time spent allocating memory, with possible durations for block allocation requests: *average* allocation times; (ii) worst-case allocation times measured on the application and (iii) worst-case allocation time obtained analytically. The left part of the figure depicts the results for the real workload mpg123 while its right part depicts the results for the synthetic workload.

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



(a) Mpg123 workload

(b) Synthetic workload

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

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 flow of control that can vary 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 obviously, 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 synthetic workload is always higher than the one on the mpg123 application, because in the former workload, memory allocation requests are more frequent and varied in size than in the latter. An interesting result is that for some allocators, like for instance the buddy systems allocators and the fast-fit allocators, considering the worst possible allocation time is realistic, at least for the workloads studied and the selected heap size (4 MB).

4.2 Real-time performance of memory deallocation

Table 3 gives the worst-case deallocation times measured on the mpg123 and synthetic workloads, as well as the worst-case deallocation times obtained analytically.

	mpg123-worst	synthetic-worst	analytical-worst
first-fit	11065	9648	11011
best-fit	13446	9964	11660
btree-best-fit	11580	8777	403299
fast-fit	10272	6697	1478499
quick-fit	13970	3036	207609
buddy-bin	11017	10750	65739
buddy-fibo	18640	13976	55341

Table 3: Worst-case performance of memory deallocation (nb. of processor cycles)

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, and are at worst linear with the number of free blocks in the tree (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 memory consuming, exhibits 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 is intended to vary 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 On the worst-case behavior of dynamic memory allocation

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

- sequential and indexed fits, 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.
- 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 (segregated fit), which has a different worst-case allocation behavior depending on the heap size: for small heaps, the worst-case allocation time occurs when fragmenting a page into the smallest possible block size ; for large heaps, the worst-case allocation time occurs when the list of large free blocks (pages) has to be scanned entirely to find a free block.

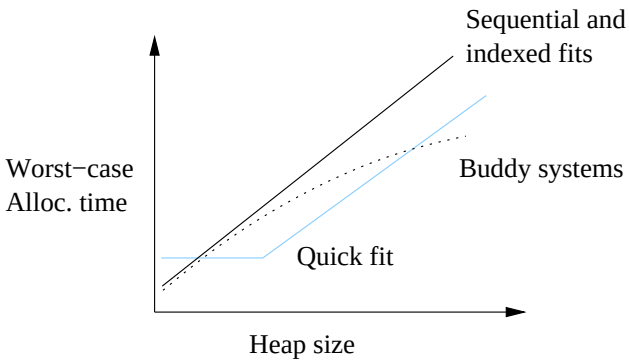


Figure 4: Evolution of worst-case allocation times with heap size

Although not quantified by measures yet, we expect the worst-case allocation times of the three classes of allocators to evolve with the heap size as shown in figure 4. 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, and this for any type of 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 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 either the worst-case analytical allocation time or the worst-case measured allocation time, one could also use a probabilistic real-time bound as suggested in [PB99] and more recently in [EB01]. Such a bound is a worst-case execution estimate m associated to a confidence level. The confidence level, obtained through statistical methods applied to a large set of execution time measurements, is the likelihood that the worst-case execution time is greater or equal to m. Establishing such a probabilistic real-time bound requires in particular that the different tests of execution times be independent [EB01]. 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. For this reason, we resorted to analytical analysis rather than to statistical methods for obtaining worst-case allocation times.

6 Conclusion

This paper has given detailed average and worst-case allocation and deallocation times of a comprehensive panel of dynamic memory allocators (sequential fits, indexed fits, segregated fits, buddy systems). For every allocator, we have compared its worst-case behavior obtained analytically with (i) the worst timing behavior observed by executing real and synthetic workloads, (ii) its average timing performance.

A result of our performance analysis is that some algorithms (sequential fits, indexed fits) have good average allocation times but perform poorly in the worst-case. In contrast, buddy systems and segregated fits such as quick-fit, while having more modest average performance, have less important worst-case allocation/deallocation times. We have also shown that when the allocation rate of applications is moderate, worst-case allocation/deallocation times obtained analytically can be used without an excessive impact on application times for the most predictable allocators (buddy systems and quick-fit). We have briefly examined the intended impact of the heap size of the worst-case behavior of the memory allocators, and discussed the context and conditions in which analytical and actually measured worst-case allocation times are best suited.

Our work can be extended in several directions. Firstly, the implementation of a larger set of applications is required to assess the worst-case observed timing behavior of the allocators and compare it to the worst-case behavior obtained analytically. The problem here is to

find applications that are representative of real-time workloads and use dynamic memory management. Another related issue would be to have a look at the distributions of the execution times of allocation/deallocation requests. Secondly, the worst-case behavior of the allocators has to be quantified for different heap sizes. Another important issue to work on is to find the best trade-off between time and memory consumption, the more predictable allocators not being necessarily the more space-efficient. Finally, the impact of hardware performance enabling features (e.g. caching) on the worst-case performance of the allocators has also to be considered.

Acknowledgments

The author would like to thank Maxime Bellengé, student at INSA Rennes, who implemented the dynamic memory allocation algorithms and conducted a large part of the experiments. Thanks also to David Decotigny (IRISA), Pascal Chevochot (Sogitec), and Guillem Bernat (Univ. of York) for their comments on earlier drafts of this paper.

References

[DTM95] 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, August 1995.

[EB01] Stewart Edgar and Alan Burns. Statistical analysis of WCET for scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, pages 215–224, London, UK, December 2001.

[EG97] 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, August 1997.

[Hip99] Michael 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.

[Joh97] M.S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, December 1997.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, chapter Information structures (chapter 2 of volume 1), pages 228–463. Addison–Wesley, 1973.

[LL85] E. L. Lloyd and M. C. Loui. On the worst case performance of buddy systems. *Acta Informatica*, 22:451–473, 1985.

[NG95] K.D. 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.

[PB99] 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.

[PB00] 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.

[Rob74] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):491–499, 1974.

[Rob75] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 1975.

[Ste83] 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, October 1983.

[Vui80] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, April 1980.

[WJNB95] P.R. Wilson, M.S. 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.

[WW88] C. B. Weinstock and W. A. Wulf. Quickfit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–144, October 1988.

[ZG94] B. Zorn and D. Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation*, 4(1):107–131, January 1994.