Improving Memory Efficiency of Dynamic Memory Allocators for Real-Time Embedded Systems

Junghee Lee and Joonhwan Yi

Dynamic memory allocators for real-time embedded systems need to fulfill three fundamental requirements: bounded worst-case execution time, fast average execution time, and minimal fragmentation. Since embedded systems generally run continuously during their whole lifetime, fragmentation is one of the most important factors in designing the memory allocator. This paper focuses on minimizing fragmentation while other requirements are still satisfied. To minimize fragmentation, a part of a memory region is segregated by the proposed budgeting method that exploits the memory profile of the given application. The budgeting method can be applied for any existing memory allocators. Experimental results show that the memory efficiency of allocators can be improved by up to 18.85% by using the budgeting method. Its worst-case execution time is analyzed to be bounded.

Keywords: Dynamic storage management, main memory, real-time systems.

Manuscript received May 11, 2010; revised July 4, 2010; accepted July 19, 2010.

This work was supported partly by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology, Rep. of Korea (grant no. 2009-0088455), and partly by the 2009 Research Grant of Kwangwoon University, Seoul, Rep. of Korea.

Junghee Lee (phone: +1 404 494 6797, email: konnen27@gmail.com) is with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Georgia, USA.

Joonhwan Yi (corresponding author, email: joonhwan.yi@kw.ac.kr) is with the Department of Computer Engineering, Kwangwoon University, Seoul, Rep. of Korea.

doi:10.4218/etrij.11.0110.0268

I. Introduction

There are three important requirements for a dynamic memory allocator to be used for real-time systems [1], [2]: bounded worst-case execution time, fast average execution time, and minimal fragmentation. In this paper, we focus on minimizing fragmentation while the other requirements are still satisfied. Memory fragmentation is a serious problem for real-time systems that run continuously during their whole lifetime [3].

Since real-time embedded systems generally run with fixed applications, there are objects whose type can be determined a priori while the number of objects to be alive in memory is unpredictable. An object, in this sense, is a kind of structured data item, such as a Pascal record, C struct, or C++ object [4]. Based on the fact that the types of objects to be stored in the heap storage are predetermined, a memory allocator can be optimized for better memory efficiency.

This paper proposes a budgeting method that divides the heap storage into two parts, dedicated and shared, and reduces fragmentation by budgeting dedicated storage for predetermined objects. To budget the optimum size of dedicated storage, the budgeting method utilizes the given memory profile that traces when and which object is allocated and freed. The experimental results show that the proposed method improves the memory efficiency of state-of-the-art allocators in [3], [5], and [6] by up to 18.85% on average.

Section II reviews the related works. Sections III and IV describe the budgeted allocator and the budgeting method, respectively. Section V shows the experimental results to prove the effectiveness of this method and provides an analysis of its costs. Finally, section VI provides the conclusion.

II. Related Work

While dynamic memory allocators are prevalently used in most software, they are rarely adopted in real-time systems due to their unpredictable execution time. A half-fit allocator [7] with constant execution time was proposed but suffers from high fragmentation [1]. Recently, a two-level segregated-fit (TLSF) allocator [3] was reported to achieve both bounded execution time and low fragmentation.

Minimizing fragmentation is becoming more important since the number of real-time applications requiring a large amount of memory is increasing, such as video streaming, virtual reality, scientific data gathering, and data acquisition [1].

In order to reduce fragmentation, a hybrid allocator [8] was proposed which applies different management algorithms depending on the size of objects. Similarly, the Lea allocator [9] uses different algorithms depending on the size of requested objects, such as block cache, segregated list, and first-fit. Static allocators do not cause fragmentation [10] because they allocate a fixed-size memory region immediately after the system initialization and do not allocate or free memory regions at run time. However, this may also lead to wastage of memory because memory must be allocated for the worst case.

Fragmentation can be reduced by improving the locality. Improving the locality can be accomplished by analyzing the lifetime [11], freeing all the objects in a region at the same time [12], or caching frequently used objects [13]. Berger and others [9] enhanced memory efficiency by providing deletion of individual objects as well as a whole region.

Existing techniques for reducing fragmentation [8], [9], [11]-[13], however, cannot guarantee bounded execution time. The budgeting method proposed in this paper reduces fragmentation of any existing allocator if the memory profile of objects is provided. In this case, the execution time of the allocator is still bounded.

The budgeting method reduces fragmentation by providing dedicated regions for each object. The concept of a dedicated region is similar to that of a reservation [1], [14]. The technique in [1] assumes each task reserves its memory to analyze schedulability of tasks rather than to reduce fragmentation. The method in [14] uses reservation to reduce execution time and power consumption.

The slab allocator [13] also uses a similar concept of dedicating regions, named *slabs*, to certain objects. The size of a slab is a multiple of a page size, and each slab is dedicated to a type of object. So, internal fragmentation can still be significant in the dedicated region if a small number of such objects is allocated. On the other hand, there is no internal fragmentation in the dedicated region of the proposed allocator because the size of the dedicated region can be an arbitrary number. The main contribution of our paper is to provide a systematic way to determine the size of the dedicated region by analyzing memory traces a priori.

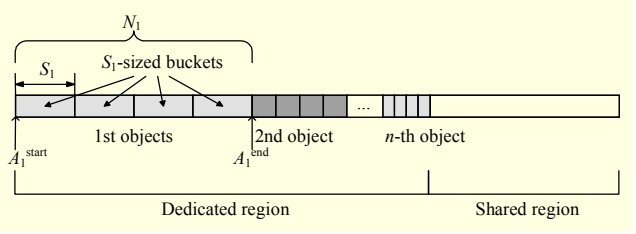

Fig. 1. Composition of heap storage.

III. Budgeted Allocator

A budgeted allocator comprises a *dedicated (region) allocator* and a *shared (region) allocator*. Accordingly, the heap storage is divided into two parts: the dedicated region and shared region as shown in Fig. 1. The dedicated region is a segregated storage [4] for predetermined objects O_i of size S_i where $S_i \neq S_j$ $(i \neq j)$ for $1 \leq i, j \leq n$. Note that different objects a and b can be O_i if their sizes are the same as S_i. The dedicated region for O_i is composed of N_i buckets that are S_i-sized and dynamically managed. The shared region is a normal storage that is managed by a shared allocator.

The budgeting method improves the memory efficiency of the shared allocator by using the dedicated region for predetermined objects. The exact segregated fit allocator [4] is the dedicated allocator, and any proper existing memory allocator can be used as a shared allocator. The contribution of the proposed method is to improve memory efficiency by sizing (or budgeting) the dedicated region well.

Object O_i is first allocated to the dedicated region by the dedicated allocator. If there is no free bucket of the exact size S_i in the dedicated region, O_i is allocated to the shared region rather than allocated to a larger bucket in the dedicated region. In this way, no fragmentation is possible by the dedicated allocator. Any other object O_s where $s > n$ is always allocated to the shared region by the shared allocator.

Figure 2 shows the pseudocode of the budgeted allocator. Note that S_i and N_i for $1 \leq i \leq n$ are predetermined. Object size S_i is provided by the user, and N_i is determined by the budgeting method described in section IV. Free buckets of size S_i are managed by a stack ST_i. So, there are n stacks $\{ST_1, ST_2, \ldots, ST_n\}$ for the free buckets.

The function *init* takes two arguments, S_i and N_i, and reserves N_i buckets of size S_i by a low-level allocator. The low-level allocator is a type of system call to increase the data space of the program with respect to the requested size. For example, library routine *sbrk* [4] in C behaves as a low-level allocator.

```
data structure for i-th object in the dedicated region
    ST_i   // Stack for tracking the free list of S_i-sized buckets
    A_i^start // Start address of S_i-sized buckets
    A_i^end // End address of S_i-sized buckets
    S_i // Size of one i-th object
end data structure

function init(S_i, N_i)
    // S_i denotes the size of objects O_i in the dedicated region
    // N_i denotes the number of dedicated buckets for O_i
     Low-level allocator allocates N_i buckets of size S_i
     Create a data structure
     Push N_i buckets into ST_i
     Store the start address A_i^start of S_i-sized buckets
     Store the end address A_i^end of S_i-sized buckets
     Store S_i
     return i; // the index of O_i
end function

function malloc(i)
    // i denotes the index of the requested object
    if ST_i is not empty
        remove and return a bucket from the top of ST_i
    else
         call the shared allocator for a region of size S_i
         return the result
    end if
end function

function free(b, i)
    // b denotes the pointer of the bucket to be freed
    // i denotes the index of the requested object
    If A_i^start ≤ b ≤ A_i^end
         add bucket b to the top of ST_i
    else
         return b to the shared storage allocator
    end if
end function

function main // example
    i_0 = init(S_0, N_0)
    i_1 = init(S_1, N_1)
    for x=0 to 100
         O_0 = malloc(i_0)
         O_1 = malloc(i_1)
         process(O_0, O_1, x)
         free(O_0, i_0)
         free(O_1, i_1)
    end for
end function
```

Fig. 2. Pseudocode of budgeted allocator.

Most dynamic memory allocators reserve memory regions to be managed by themselves using a low-level allocator. Then, it creates a data structure for the requested object and fills each field. The data structure consists of a stack ST_i, start and end addresses of N_i buckets, and the size S_i of the object. The start and the end addresses of the buckets are needed to determine whether a bucket to be freed belongs to the dedicated region or the shared region when the function *free* is called. The size of the object is stored for requesting a region by the shared allocator when there are no more available buckets in the dedicated region. Finally, it returns the index of the object.

The function *malloc* returns a bucket from the top of stack ST_i if there remains a free bucket. Otherwise, it passes the request to the shared allocator and returns a part of the shared region.

The function *free* determines if the bucket to be freed belongs to the dedicated region by comparing the pointer of the bucket with the start and the end addresses of S_i-sized buckets. If the bucket to be freed belongs to the S_i-sized buckets, it is added into ST_i. Otherwise, it is passed to the shared allocator to be freed.

The caller should call the function *init* for each object and store its index for calling the function *malloc* and *free*. This interface is different from that of traditional memory allocators. We chose this interface for real-time applications by improving the average execution time and by making the worst case execution time bounded.

To keep the same interface of traditional memory allocators, the function *free* needs to have only one argument *b*. Because the other argument *i* is not provided anymore, it requires a linear search which makes the execution time longer and unpredictable. So, the index *i* is provided for function *free* so that *free* can be done in a constant time.

Buckets in the dedicated region can be allocated and freed within a constant time. For buckets in the shared region, their allocation and de-allocation time is bounded as long as the worst cast execution time of the shared allocator is bounded. This means that the budgeting method can be applied to memory allocators for real-time applications whose worst case execution time needs to be bounded.

IV. Budgeting Method

This section describes how to determine N_i, which is the number of buckets dedicated to O_i in the dedicated region. Three solutions are available for the determination of N_i: user-driven, compiler-driven, and profile-driven solutions [11]. The user-driven solution may suffer from human errors [11]. Lack of information of run time behaviors inhibits applying the compiler-driven solution to the budgeting method. Thus, we selected the profile-driven solution, which is widely used for customizing memory allocators [9], [11], [12], [15].

The memory profile $P_i(t)$ is obtained by post-processing the memory trace. The number of buckets in use of size S_i at time t is denoted by $P_i(t)$. To generate the trace, the application is compiled and run with a conventional general-purpose allocator that contains augmented codes to generate the trace. In the allocation function (for example, *malloc*), the requested

size and the returned address are traced with a timestamp. The timestamp is only used for ordering the trace. In the free function (for example, *free*) the address to be freed is traced with a timestamp. Then, $P_i(t)$ is incremented by 1 if O_i is allocated at time t and decremented by 1 if O_i is freed.

1. Problem Formulation

The main objective of determining N_i is to minimize the size of the heap storage. At first, we formulate the required size $R(t)$ of storage at time t for any memory allocator in a general form by

$$R(t) = U(t) + F(t) + H(t)\,, \tag{1}$$

where $U(t)$ is the amount of buckets in use, $F(t)$ is the internal and external fragmentation, and $H(t)$ is the overhead due to memory allocators. Here, the buckets in use refer to the allocated or reserved buckets for one object, which cannot be used for other objects. The internal fragmentation occurs in the unused region within a bucket because the allocated bucket is larger than the requested size, and the external fragmentation occurs due to the free buckets that cannot be allocated because their sizes are smaller than the requested size [15]. The overhead refers to the memory region that is used, not by the requestor, but by the allocator to manage buckets.

The size of the heap storage should be greater than the maximum R_{max} of $R(t)$. In order to minimize the size of the heap storage, R_{max} should be minimized. Overhead $H(t)$ can be ignored because it does not significantly affect to R_{max}. Fragmentation $F(t)$ cannot be estimated accurately without simulation because it depends mostly on the scenario. Equation (2) shows $U(t)$ in a different form:

$$U(t) = \sum_i (S_i \times P_i(t))\,. \tag{2}$$

Recall that S_i is the size of object O_i, and $P_i(t)$ is the number of S_i-sized buckets allocated by O_i's.

Now, we consider (1) for our budgeted allocator. At first, no fragmentation occurs in the dedicated region. Therefore, it is very likely that $F(t)$ proportionally decreases as N_i increases. In the meantime, the maximum value U_{max} of $U(t)$ may increase, which we show later. For budgeted allocators, (1) is rewritten as

$$R^B(t) = U^B(t) + F^B(t) + H^B(t)\,. \tag{3}$$

The superscript B denotes that each variable is for the budgeted allocator. Then, $U^B(t)$ can be written as

$$U^B(t) = \sum_i (S_i \times \mathrm{MAX}(N_i, P_i(t)))\,, \tag{4}$$

where MAX(A, B) denotes the larger value between A and B. This equation implies that $U^B(t)$ is larger than $U(t)$ if $N_i > P_i(t)$ for each i because other objects cannot use the S_i-sized buckets in the dedicated region. Note that $P_i(t)$ is independent of $P_j(t)$ where $i \neq j$. Our goal is to minimize R_{max} while maintaining U^B_{max} smaller than U_{max}. That is,

$$U^B_{max} \le U_{max}. \tag{5}$$

Note that if $N_i > P_{i,max}$, $(N_i - P_{i,max})$ buckets in the dedicated region are useless where $P_{i,max}$ is the maximum of $P_i(t)$. So the following condition should also be enforced:

$$N_i \le P_{i,max}\,, \quad \text{for every } i\,. \tag{6}$$

In addition, the dedicated region should be maximized to minimize $F^B(t)$. As the size of the dedicated region is

$$D = \sum_i (S_i \times N_i)\,. \tag{7}$$

D should be maximized while $U^B_{max} \le U_{max}$ from (5) and $N_i \le P_{i,max}$ from (6) for every i. In summary, the problem is defined as the following.

Problem Definition. Given S_i and $P_i(t)$, find N_i such that D is maximized while $U^B_{max} \le U_{max}$ and $N_i \le P_{i,max}$ for every i, where

$$D = \sum_i (S_i \times N_i),\ U(t) = \sum_i (S_i \times P_i(t)), \text{ and}$$

$$U^B(t) = \sum_i (S_i \times \mathrm{MAX}(N_i, P_i(t)))\,.$$

An illustrative example is shown in Fig. 3. In Fig. 3(a) and

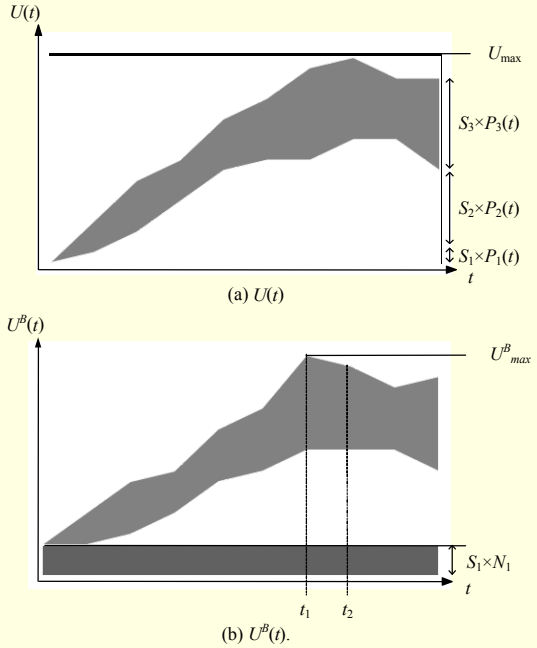

Fig. 3. Illustrative example of $U(t)$ and $U^B(t)$.

Fig. 3(b), $U(t)$ and $U^B(t)$ are depicted, respectively, where only N_1 is non-zero and N_2 and N_3 are zero. As $U(t)$ is a summation of $S_i \times P_i(t)$ for every i, the graph has split regions. The region due to $S_1 \times P_1(t)$ is the darkest one. Note that the height of the darkest region in $U^B(t)$ is always greater than $S_1 \times N_1$. In this example, N_1 is set to $P_1(t_2)$ when $U(t_2)$ is U_{max}. This selection of N_1 does not cause $U^B(t_2)$ increase in Fig. 3(b). However, it causes $U^B(t_1)$ to exceed U_{max}, which means that $U^B{}_{max}$ becomes greater than U_{max}. To maintain $U^B{}_{max}$ smaller than U_{max}, N_1 should be set to smaller value so that $U^B(t)$ does not exceed U_{max} all the time. An algorithm to compute N_i satisfying (5) is presented in the following subsection.

2. Solution

This problem cannot be solved within the polynomial time by an exhaustive algorithm. From (6), N_i can be any value from 0 to $P_{i,max}$, and we need to explore the combination of $\{N_1, N_2, \ldots, N_n\}$ that makes the size of the dedicated region largest while maintaining $U^B{}_{max} \le U_{max}$. Consequently, the complexity is proportional to $\prod_i P_{i,max}$. Such algorithms can hardly be applied when n or $P_{i,max}$ is large. As a practical solution, we propose a greedy algorithm whose pseudocode is shown in Fig. 4. This algorithm can solve the problem within polynomial time as its complexity is proportional to n.

Every time t, following (8) should be satisfied:

$$U^B(t) \le U_{max}. \tag{8}$$

Inserting (4) to (8) becomes

$$\sum_i (S_i \times \mathrm{MAX}(N_i, P_i(t))) \le U_{max}, \tag{9}$$

for every i and t. A sub-optimum N_i can be computed by assuming that N_j ($i \ne j$) are fixed a priori. The following assumptions are used to compute sub-optimum N_i:

1) S_i is ordered such that $S_i \ge S_j$ if $i < j$.
2) N_i is computed by assuming that N_k is precomputed and $N_j = 0$ where $k < i < j$.

For N_1, assume that N_j ($1 < j \le n$) is zero. Then, (9) can be rewritten as

$$S_1 \times \mathrm{MAX}(N_1, P_1(t)) \le U_{max} - \sum_{j=2}^{n} (S_j \times P_j(t)). \tag{10}$$

Note that every variable except for N_1 is known. If $N_1 \le P_1(t)$ for every t, (10) becomes

$$S_1 \times P_1(t) \le U_{max} - \sum_{j=2}^{n} (S_j \times P_j(t)), \tag{11}$$

that is always true by the definition of U_{max}. In other words, if

```
begin
    for i = 1 to n
        compute P_{i,max}
    end for
    compute U_max

    for i = 1 to n
        compute M_{i,min}
            where
                M_i(t) = U_max − Σ_{j=1}^{i−1} (S_j × MAX(N_j, P_j(t))) − Σ_{j=i+1}^{n} (S_j × P_j(t))
        N_i = MIN(P_{i,max}, ⌊M_{i,min} / S_i⌋)
    end for
end
```

Fig. 4. Pseudocode of greedy algorithm.

$N_1 \le P_1(t)$ for every t, it is always true that $U^B{}_{max} \le U_{max}$. However, if $N_1 > P_1(t)$ at a time t, from (10)

$$S_1 \times N_1 \le \left\{ U_{max} - \sum_{j=2}^{n} (S_j \times P_j(t)) \right\} = M_1(t). \tag{12}$$

Note that the right hand side is denoted by $M_1(t)$ that is a function of t. The semantics of $M_1(t)$ is the amount of memory region that can be reserved for the object O_1 at time t. That is,

$$N_1 \le \frac{M_1(t)}{S_1}, \tag{13}$$

for every t. Therefore, if

$$N_1 \le P_1(t) \quad \text{or} \quad P_1(t) < N_1 \le \frac{M_1(t)}{S_1}, \tag{14}$$

for every t, $U^B{}_{max} \le U_{max}$. In other words,

$$N_1 \le \frac{M_{1,min}}{S_1}, \tag{15}$$

where $M_{1,min}$ is the minimum of $M_1(t)$. Because we are looking for the largest N_1 that satisfies $U^B{}_{max} \le U_{max}$ and $N_1 \le P_{1,max}$,

$$N_1 = \mathrm{MIN}\left(\left\lfloor \frac{M_{1,min}}{S_1} \right\rfloor, P_{1,max} \right). \tag{16}$$

In general, the solution is given as the following.

Solution. The number N_i of dedicated storages for i-th object is

$$N_i = \mathrm{MIN}\left(\left\lfloor \frac{M_{i,min}}{S_i} \right\rfloor, P_{i,max} \right),$$

where

$$M_i(t) = U_{max} - \sum_{j=1}^{i-1} (S_j \times \mathrm{MAX}(N_j, P_j(t))) - \sum_{j=i+1}^{n} (S_j \times P_j(t)).$$

Our experimental results in section V show that the proposed

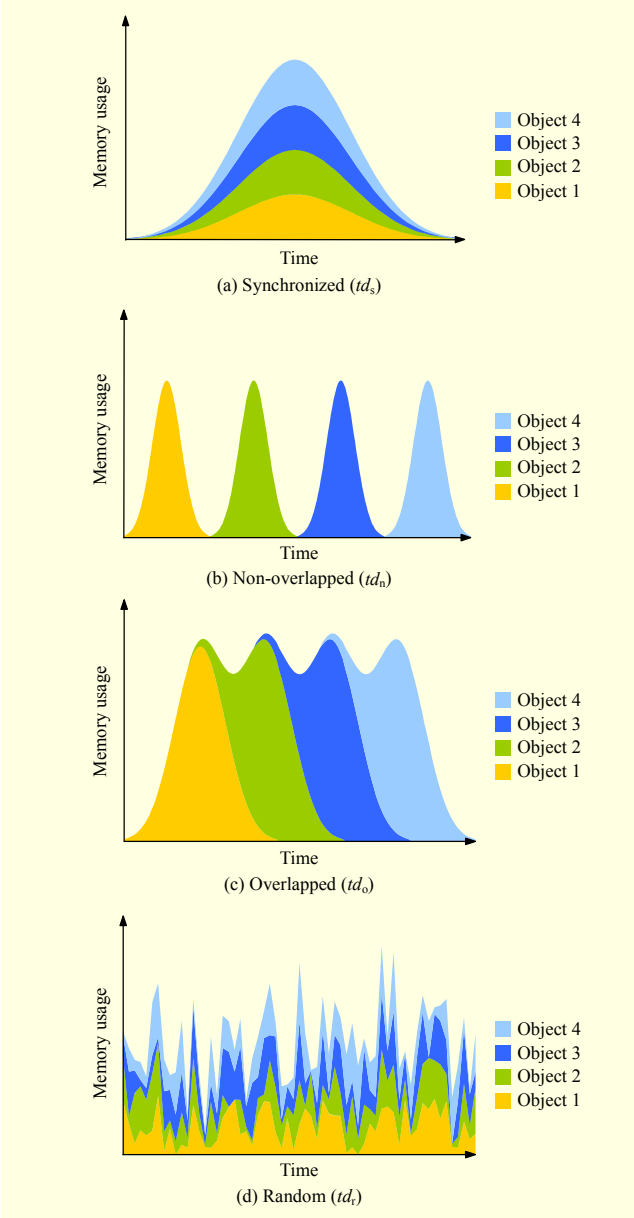

Fig. 5. Four temporal distributions $TD = \{td_s, td_n, td_o, td_r\}$ of objects over time considered in this experiment.

algorithm is much faster than the exhaustive search algorithm and the results of $\{N_1, N_2, \ldots, N_n\}$ are close to the optimum values.

V. Experiments

The methodology used for the experiments is explained first, and the impacts of the proposed budgeting methods on memory efficiencies are presented. Various known memory allocators have been used for the experiments. In addition, to show the efficiency of the proposed budgeting algorithms, the execution time is compared to the exhaustive search algorithm. Finally, the overhead of the budgeted method is measured by the execution time and code size of allocators with and without budgeting methods.

The execution time is measured by using an electronic system level simulator, Carbon Design System's SoC Designer [16], with a cycle-accurate ARM926 model. In order to minimize nondeterministic behaviors, all the instructions and data were stored on a static random access memory while the caches were turned off. The code size of each allocator is measured after compiling them with ARM RVCT2.2 [17].

1. Methodology

The amount of fragmentation depends largely on the scenarios. Providing an appropriate scenario is crucial to experiments of fragmentation [7]. There are mainly two approaches to generate scenarios: standard benchmark applications and synthetic workload models [3]. As mentioned in [3], the requirements of real-time applications are different from those of the standard benchmark applications. Real-time applications should run continuously during their lifetime responding to unpredictable inputs. It is the reason why synthetic workload models are adopted by many studies on memory allocators of real-time applications [1], [3], [7], [18].

Four temporal distributions of objects shown in Fig. 5 with five object configurations shown in Table 1 are considered in our experiments. More detail discussions on Fig. 5 and Table 1 are given in the following paragraphs. A scenario is composed of a temporal distribution of objects with an object configuration, and thus there are twenty scenarios under consideration. Our methodology is similar to that in [7] but refined according to the characteristics of real-time applications and algorithms used by allocators.

Figure 5 illustrates four temporal distributions $TD = \{td_s, td_n, td_o, td_r\}$ of objects over time. Figure 5(a) shows the synchronized temporal distribution td_s where the amount of memory required for each object reaches the peak at the same time. This scenario happens when multiple objects are allocated in a single task. If the sizes of each object and memory profile are known a priori, the simple segregated fit allocator performs most efficiently for td_s because there is no chance to share space among objects at the peak time. Figure 5(b) depicts another extreme case, non-overlapped distribution td_n. There are no overlapped intervals among lifetimes of objects. Thus, the fragmentation is likely to be lower than other distributions because only one type of object is used during a certain time. This scenario illustrates a situation where multiple independent tasks are running concurrently in a system, and the lifetime of objects is within the task. The most realistic scenario would be the overlapped distribution td_o shown in Fig. 5(c).

Table 1. Five object configurations $OC = \{oc_{2l}, oc_{2s}, oc_l, oc_s, oc_{mix}\}$ with eight objects O_1 through O_8 and their size S_1 through S_8.

Object size / Name	2^k large (oc_{2l})	2^k small (oc_{2s})	Arbitrary large (oc_l)	Arbitrary small (oc_s)	Mixed (oc_{mix})
S_1	131,072	1,024	91,936	2,952	80,176
S_2	65,536	512	80,176	1,978	65,536
S_3	32,768	256	64,032	1,040	32,768
S_4	16,384	128	49,472	996	19,008
S_5	8,192	64	35,432	384	2,952
S_6	4,096	32	19,008	192	1,024
S_7	2,048	16	17,424	40	40
S_8	1,024	8	5,184	10	10

There are some overlapped and some non-overlapped intervals among lifetimes. This scenario may happen either in using multiple objects in a single task or in using objects across multiple tasks. Figure 5(d) shows a purely random scenario td_r.

Table 1 shows five object configurations $OC = \{oc_{2l}, oc_{2s}, oc_l, oc_s, oc_{mix}\}$ characterized by the number of objects and their size. For simplicity, every configuration has eight objects, O_1 through O_8. The size S_i of each object O_i was selected to stimulate all the aspects of nine dynamic allocators. The nine dynamic allocators consist of the six allocators in [6] named from test 1 to test 6, the Kingsley (DJGPP in [6]), the Lea [5], and TLSF [3]. TLSF is the only memory allocator whose execution time is bounded. Although this paper focuses on real-time applications, the budgeting method can be applied to any memory allocators in any applications as long as their memory profile is provided. Eight other allocators are included to show that the budgeting method can improve memory efficiency of allocators whose execution time is not bounded.

Allocators in test 1 through test 3 always allocate 2^k-sized buckets. Kingsley and Lea allocators allocate 2^k-sized buckets for small objects. They are likely to be efficient if every S_i is 2^k-sized. Thus, we use both object configurations with only 2^k-sized objects, oc_{2l} and oc_{2s}, and without 2^k-sized objects, oc_l and oc_s. Then, we divided them into those with larger than 1,024 bytes and those smaller than 1,024 bytes, that is, oc_{2l} and oc_{2s}, respectively.

All the allocators, except in test 1 and the TLSF allocator, allocate an additional memory chunk C from the low-level allocator if there is no more room. Thus, the relative size between S_i and C affects their memory efficiency measured by the low-level allocator. In this experiment, a 1,024 byte memory was used for memory chunk C.

To generate each scenario, we assumed periodic tasks. It should be noticed that periodic tasks were assumed only for the convenience of scenario generation. The budgeting method does not need to assume periodic tasks. One task corresponds to one object, which means that a task allocates and frees one object during its active interval. An active interval of each task was randomly selected and corresponding object were randomly allocated and freed. By adjusting the distribution of the active interval, periods of tasks, and intervals among tasks all of the scenarios were generated. The synchronized distribution td_s was set to reach peak at the same time forcefully. The maximum number $P_{i,\max}$ of objects requested for the i-th object O_i was set to 100 for every i.

Finally, we added the mixed object configuration. The four other configurations $\{oc_{2l}, oc_{2s}, oc_l, oc_s\}$ were chosen to stimulate all the aspects of the nine allocators. However, these are less realistic. For instance, it is rare for an application to use only 2^k-sized objects such as oc_{2l} and oc_{2s}. The mixed configuration oc_{mix} was made up by mixing two sizes from $\{oc_{2l}, oc_{2s}, oc_l, oc_s\}$.

The memory efficiency was measured by comparing the maximum size R_{max} of $R(t)$ for different memory allocators. If an allocator A has smaller R_{max} than another allocator B does, A has higher memory efficiency than B because A requires smaller memory for the same application. The maximum amount of allocated regions by the low-level allocator was measured as R_{max}. TLSF was an exceptional case because it simply fails the system if the pre-requested storage size is not. For TLSF, R_{max} was measured by trial and errors. That is, each scenario was iteratively simulated with large heap storage at the beginning, and then the size of the storage was reduced until the TLSF allocator failed to respond to a memory request.

2. Experimental Results

The budgeting methods are applied to the twenty scenarios defined in the previous subsection. The memory efficiency is measured by the reduction rate of the R_{max}, the maximum size of the required memory. Recall that each m_a of nine previously published allocators is compared against the budgeted allocator employing m_a as the shared allocator. For each scenario and each memory allocator, the reduction rate of R_{max} is computed. That is, $(R^B_{max}/R_{max}) \times 100$ is computed where R^B_{max} denotes the maximum size of required memory after the budgeting method is applied. Then, the average reduction rate of R_{max} for nine memory allocators is computed for each scenario.

Table 2 shows the average reduction rate of R_{max} for the twenty scenarios. Except one scenario $\{td_n, oc_l\}$, the budgeted allocator achieves up to 18% of memory efficiency. More detail analysis follows.

As shown in Table 2, the improvement in memory efficiency mostly depends on the temporal distribution TD rather than

Table 2. Average reduction rate of R_{max} for twenty scenarios.

Scenario	Synchronized (td_s)	Non-overlapped (td_n)	Overlapped (td_o)	Random (td_r)
2^k large (oc_{2l})	10.74%	2.71%	5.34%	8.31%
2^k small (oc_{2s})	18.85%	3.21%	7.56%	11.76%
Arbitrary large (oc_l)	14.55%	–1.37%	4.46%	8.43%
Arbitrary small (oc_s)	14.52%	2.00%	8.48%	11.59%
Mixed (oc_{mix})	15.44%	0.30%	8.20%	13.39%

object configuration *OC*. So, in Fig. 6, we depict the memory efficiency of different memory allocators for various temporal distributions with mixed object configuration only, that is, scenarios $\{td_s, oc_{mix}\}$, $\{td_n, oc_{mix}\}$, $\{td_o, oc_{mix}\}$, and $\{td_r, oc_{mix}\}$. For different scenarios, N_i's need to be recalculated. Thus, the four scenarios shown in Fig. 6 use different N_i's. While each cell in Table 2 shows the average reduction rate of nine memory allocators for the given scenario, Fig. 6 shows R_{max} and R^B_{max} of nine dynamic allocators for the given scenario. The vertical axis indicates the size of R_{max} and R^B_{max} in bytes.

Figure 6(a) shows the result for scenario $\{td_s, oc_{mix}\}$. Our budgeting algorithm results show that $N_i = P_{i,max}$, which means that only the dedicated allocator should be used, and the shared allocator should be never used. That is the reason why the R^B_{max} of every allocator is same. As a result, memory efficiency was improved by 15.44% on average. We also obtained similar results when the object configurations were changed.

Figure 6(b) shows the result for scenario $\{td_n, oc_{mix}\}$. Our budgeting algorithm results show that most N_i's are zero. This is one of the worst cases from the viewpoint of the budgeting method because there is little fragmentation. This case was chosen to illustrate the limitation of the proposed budgeting method. Since the budgeting method improves the memory efficiency by reducing fragmentation, it hardly works when there is little fragmentation. Moreover, it may even worsen the efficiency as illustrated in Fig. 3 (see the results of test 2 in Fig. 6(b)). The budgeting method increases the possibility that the fragmentation is reduced but cannot guarantee it. That is also the reason why the memory efficiency worsened for the scenario $\{td_n, oc_l\}$ with arbitrary large objects shown in Table 2.

Figures 6(c) and 6(d) show the results for scenarios $\{td_o, oc_{mix}\}$ and $\{td_r, oc_{mix}\}$, respectively. The budgeting method improves memory efficiencies by 8.20% and 13.39%, respectively, on average.

Now, we demonstrate the efficiency of the proposed greedy algorithms that compute the size of dedicated regions. Table 3 shows the comparison results of the exhaustive algorithm and

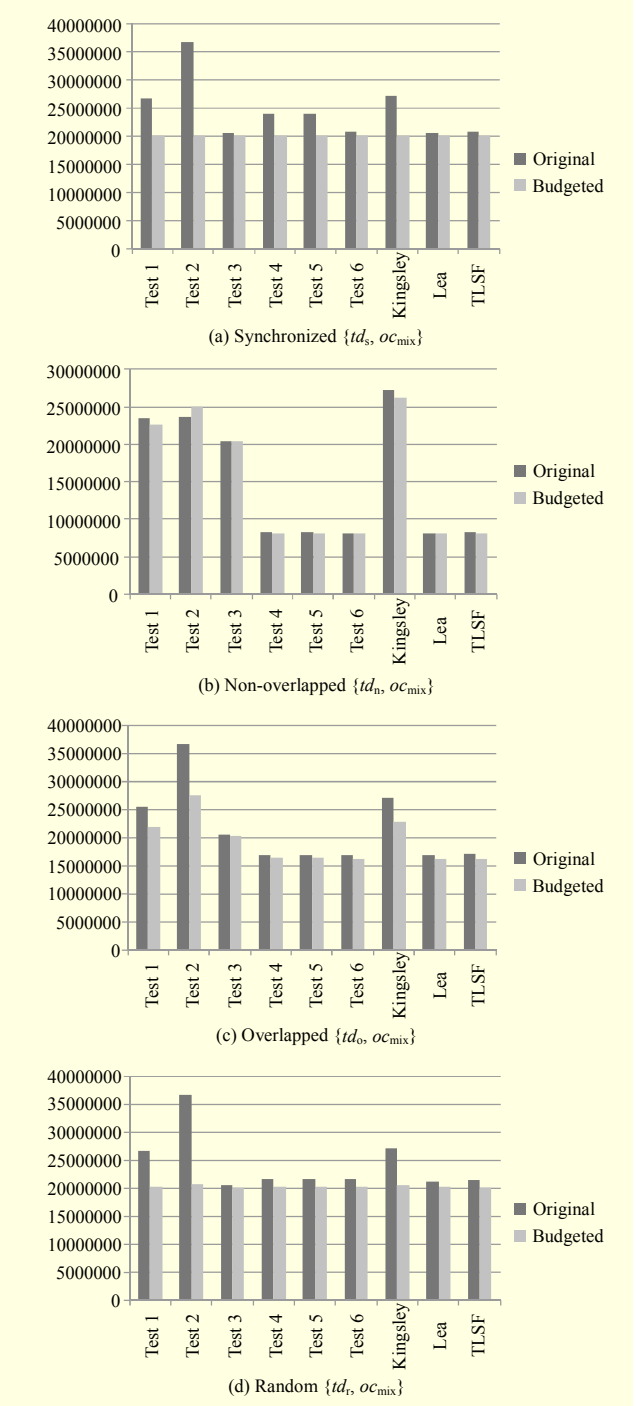

Fig. 6. Comparison of memory efficiency for different temporal distributions with mixed object configuration oc_{mix}.

the proposed greedy algorithm. The execution time speedup is the execution time ratio of the exhaustive algorithm to the proposed greedy one. The greedy algorithm is up to 15,000 times faster than the exhaustive algorithm, while it finds near optimum values of N_i that differ at most 0.2% from the optimum values in our experiments. Note that if the number n

Table 3. Comparison of budgeting algorithms in execution time and accuracy.

n		4	4	5
$P_{i,\max}$		10	20	10
Execution time (ms)	Exhaustive	7,391	33,781	157,671
	Greedy	7	8	10
	Speedup	1,056	4,223	15,767
$\sum_i (S_i \times N_i)$	Exhaustive	15,520	34,688	16,016
	Greedy	15,520	34,688	15,984
	Ratio (accuracy)	100%	100%	99.8%

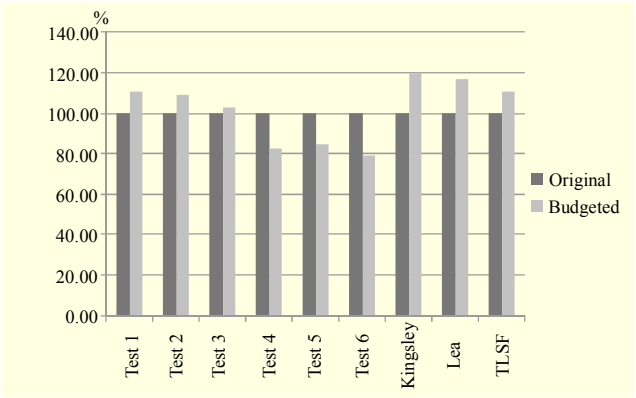

Fig. 7. Execution time overhead due to budgeting method.

of objects with different sizes and $P_{i,\max}$ are increasing, the execution time of the exhaustive algorithm increases exponentially. Considering that most of real-time systems use $n >> 5$, the exhaustive algorithm would be infeasible for such a large number of objects.

Figure 7 shows the cost of the budgeting method. The average execution time tends to increase as shown in Fig. 7. On average, the execution time is increased by 1.66%. This is because budgeted allocators need to check first if there is a free bucket in the dedicated region. Considering real-time applications, in the case of the TLSF allocator, its average increment in execution time is 9.36%.

Whether this overhead is significant or negligible depends on the application. Because the fragmentation is also highly dependent on the application, the reduction rate of the required memory size varies with the application, too. The designer should make a decision considering the trade-off between the memory size and the average execution time. However, the proposed method provides a way to enhance memory efficiency with less overhead than using more efficient but more complicated, slower, and not WCET-bounded memory allocators.

The code size of the budgeted allocator inevitably increases because it is composed of two allocators. However, the amount of the increment was measured as 548 bytes. That is negligibly small considering the memory requirement of emerging real-time applications, such as video streaming, virtual reality, scientific data gathering, and data acquisition. Because the budgeted allocator achieves higher memory efficiency than other allocators even with the higher code size overhead, it is clear that the budgeted allocator improves fragmentation.

VI. Conclusion

A budgeting method of memory regions was proposed to enhance memory efficiency while real-time requirements on execution time are satisfied. To minimize fragmentation, the budgeted allocator exploits a dedicated storage for predetermined sizes of objects. To determine the number of dedicated buckets for each object, a formulation and its heuristic solution were presented. The budgeting method was applied to nine memory allocators. Our experiments show that the budgeted allocators achieved improvements in memory efficiency by up to 18.85%. We also designed budgeted allocators such that the worst-case execution time of the budgeted allocator is bounded as long as the shared allocator is bounded. Although we focus on real-time applications in this paper, the budgeting method can benefit any applications whose memory efficiency needs to be improved if its memory profile is provided.

References

[1] A. Marchand et al., "Memory Resource Management for Real-time Systems," *Proc. Euromicro Conf. Real-Time Systems*, Pisa, Italy, July 2007, pp. 201-210.

[2] M. Ramakrishna et al., "Smart Dynamic Memory Allocator for Embedded Systems," *Proc. Int. Symp. Computer Inf. Sci.*, 2008, pp. 1-6.

[3] M. Masmano et al., "TLSF: A New Dynamic Memory Allocator for Real-Time Systems," *Proc. Euromicro Conf. Real-Time Syst.*, Catania, Italy, June 2004, pp. 79-88.

[4] P. Wilson et al., "Dynamic Storage Allocation: A Survey and Critical Review," *Technical Report*, Department of Computer Science, Univ. of Texas, Austin, 1995.

[5] D. Lea, *A Memory Allocator*. Available: http://g.oswego.edu/dl/html/malloc.html

[6] Delorie software. Available: http://www.delorie.com/djgpp/malloc

[7] T. Ogasawara, "An Algorithm with Constant Execution Time for Dynamic Storage Allocation," *Proc. 2nd Int. Workshop Real-Time Computing Syst. Appl.*, 1995, pp. 21-25.

[8] Y. Hasan and J. Chang, "A Hybrid Allocator," *Proc. IEEE Int.*

Symp. Performance Anal. Syst. Software, Austin, USA, Mar. 2003, pp. 214-222.

[9] E. Berger, B. Zorn, and K. McKinley, "Reconsidering Custom Memory Allocation," *Proc. ACM Conf. Object-Oriented Programming, Syst., Languages, Appl.*, Seattle, USA, Nov. 2002, pp. 1-12.

[10] H. Zhe, Z. Jun, and L. Xiling, "Design and Realization of Efficient Memory Management for Embedded Real-Time Application," *Proc. Int. Conference ITS Telecommun.*, Chengdu, China, June 2006, pp. 174-177.

[11] M. Seidl and B. Zorn, "Segregating Heap Objects by Reference Behavior and Lifetime," *Proc. Int. Conf. Architectural Support for Programming Languages Operating Syst.*, San Jose, USA, Oct. 1998, pp. 12-23.

[12] M. Tofte and J. Talpin, "Region-Based Memory Management," *Inf. Computation*, vol. 132, no. 2, Feb. 1997, pp. 109-176.

[13] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," *Proc. USENIX Technical Conf.*, Boston, USA, June 1994, pp. 87-98.

[14] A. Eswaran and R. Rajkumar, "Energy-Aware Memory Firewalling for QoS-Sensitive Applications," *Proc. Euromicro Conf. Real-Time Syst.*, 2005, pp. 11-20.

[15] D. Atienze et al., "Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications," *Proc. Design, Automation and Test in Europe*, Acropolis, France, Apr. 2004, pp. 532-537.

[16] Carbon Design Systems, *SoC Designer*. Available: http://carbondesignsystems. com/Products/SoCDesigner.aspx

[17] ARM Ltd., *RealView Compilation Tools (RVCT)* 2.2. Available: http://www. arm.com

[18] B. Zorn and D. Grunwald, "Evaluating Models of Memory Allocation," *ACM Trans. Modeling Computer Simulation*, vol. 4, no. 1, Jan. 1994, pp. 107-131.

Junghee Lee received the BS and MS in computer engineering from Seoul National University, Rep. of Korea, in 2000 and 2003, respectively. He has been a PhD student of Georgia Institute of Technology since 2008. From 2003 to 2008, he was with Samsung Electronics, where he worked on the electrical system level design of mobile system-on-chips. His current research interests include architecture design of microprocessors, memory hierarchy, and storage systems for high performance computing and embedded systems.

Joonhwan Yi received the BS in electronics engineering from Yonsei University, Seoul, Rep. of Korea, in 1991, and the MS and PhD in electrical engineering and computer science from the University of Michigan, Ann Arbor, in 1998 and 2002, respectively. From 1991 to 1995, he was with Semiconductor Business, Samsung Electronics Corporation, Rep. of Korea, where he was involved in developing application specific integrated circuit cell libraries. In 2000, he was a summer intern with Cisco, Santa Clara, CA, where he worked on path delay fault testing. From 2003 to 2008, he was with Telecommunication Networks, Samsung Electronics Company, Suwon, Rep. of Korea, where he worked on the system-on-chip architectural design for mobile applications. Since 2008, he has been a faculty member of the Computer Engineering Department, Kwangwoon University, Seoul, Rep. of Korea. His current research interests include C-level system modeling for fast hardware and software co-simulation for computer vision based applications, system-level power analysis and optimization, and high-level testing.