논문 2008-45SC-2-8

내장형 시스템을 위한 Budgeted 메모리 할당기

(Budgeted Memory Allocator for Embedded Systems)

이 중 희*, 이 준 환**

(Junghee Lee and Joonhwan Yi)

요 약

내장형 시스템의 설계 유연성을 높이고 예측하기 어려운 입력과 출력을 다루기 위해 동적 메모리 할당기가 사용된다. 일반적으로 내장형 시스템은 사용 기간 동안 계속 수행되기 때문에 메모리 할당기를 설계하는데 있어서 단편화 문제가 중요한 고려 사항 중 하나이다. 본 논문에서는 미리 구분된 객체들에 대한 전용 영역을 활용하여 단편화를 최소화시키기는 budgeted 메모리 할당기를 제안한다. 최신의 메모리 할당기를 사용하는 대신 budgeted 메모리 할당기를 사용하면 필요한 힙 영역의 크기를 최대 49.5% 감소시킬 수 있었다. 힙 영역의 크기가 16KB 이상이면 budgeted 메모리 할당기를 사용함으로 늘어나는 코드의 크기를 줄어든 단편화로 보상할 수 있다.

Abstract

Dynamic memory allocators are used for embedded systems to increase flexibility to manage unpredictable inputs and outputs. As embedded systems generally run continuously during their whole lifetime, fragmentation is one of important factors for designing the memory allocator. To minimize fragmentation, a budgeted memory allocator that has dedicated storage for predetermined objects is proposed. A budgeting method based on a mathematical analysis is also presented. Experimental results show that the size of the heap storage can be reduced by up to 49.5% by using the budgeted memory allocator instead of a state-of-the-art allocator. The reduced fragmentation compensates for the increased code size due to budgeted allocator when the heap storage is larger than 16KB.

Keywords : Memory allocator, dynamic memory management, embedded system

I. Introduction

Many embedded systems adopt a dynamic memory allocator to deal with unpredictable inputs and outputs. As embedded systems generally run with fixed applications, there are objects whose type can be determined a priori while their number to be stored is unpredictable. An object, in this sense, is a kind of structured data item such as a Pascal record, C struct, or C++ object[2]. 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.

Memory efficiency is especially important for embedded systems that run continuously during its whole lifetime where the memory fragmentation problem is very serious[1]. However, our experimental results in Section V reveal that traditional allocators are not always efficient because of fragmentation.

This paper proposes a *budgeted memory allocator* or, in short, a *budgeted allocator*, which can reduce fragmentation by budgeting dedicated storage for predetermined objects. Further, a budgeting method based on mathematical analysis is presented. The experimental results show that the size required for the heap storage with our budgeted memory allocator

* 정회원, ** 정회원-교신저자, 삼성전자 통신연구소
(Telcommunication R&D Center, Samsung Electronics)
접수일자: 2008년2월13일, 수정완료일: 2008년3월13일

is always smaller than those required with state-of-the-art allocators[1, 5, 9].

Section II explains the motivation of this work and 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 allocator and provides an analysis of its costs. Finally, Section VI provides the conclusion.

II. Motivation and Related Works

Many studies have been conducted on dynamic allocators for general-purpose heap storage. The Kingsley and the Lea allocators are well-known state-of-the-art allocators[2]. The Kingsley allocator is used in Berkeley software distribution (BSD) and is well known for its high speed[7]. The Lea allocator used in Linux is known to be both fast and memory-efficient[7]. However, the experimental results in Section VI show that they are not always efficient because of fragmentation.

In order to reduce fragmentation, a hybrid allocator[12] was proposed, which applies different management algorithms depending on the size of objects. The scheme of the different behavior based on the size is also implemented in the Lea allocator[6]. Static allocators do not cause fragmentation[4] 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 worst cases.

Fragmentation may be reduced by improving the locality by analyzing the lifetime[8], freeing all the objects in a region at the same time[11] or caching frequently used objects[13]. Berger[6] could enhance the memory efficiency by providing individual objects deletion capability along with deletion of a whole region. However, abovementioned studies focus on reducing execution time rather than fragmentation.

Our budgeted allocator reduces fragmentation by budgeting a dedicated region and avoids wastage of memory by providing a shared region for worst cases. The budgeted allocator uses a general-purpose allocator for the shared region. Thus, previous studies on improving memory efficiency of the general-purpose allocator can be adopted for the shared region. The budgeted allocator is especially useful for real-time systems that runs continuously during its whole lifetime where the memory fragmentation problem is very serious[1]. Its cost is increase in the code size.

III. Budgeted Allocator

The budgeted allocator comprises a *dedicated (region) allocator* and *shared (region) allocator*. The heap storage is divided into two parts: the dedicated region and the shared region, as shown in Fig. 1. The dedicated region is a simple segregated storage[2] 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 the size of a and b are same to S_i. A bucket is a chunk of memory. The dedicated region for O_i is composed of N_i buckets that are S_i-sized and dynamically managed. Object O_i is first allocated to the dedicated region by the dedicated allocator. If there is no free bucket of 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 $s > n$ are always allocated to the shared region by the shared allocator. For the shared allocator, any conventional general-purpose dynamic allocator can be used. We used the two level segregate fit (TLSF) allocator[11] as the shared

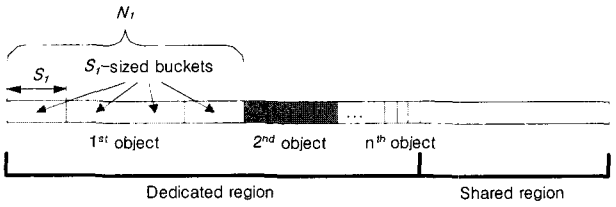


그림 1. 힙 영역의 구성
Fig. 1. Composition of the heap storage.

```
function init
// Si denotes the size of objects Oi managed in the
dedicated region
  for i = 1 to n
    Store the current position of the low-level
    allocator as the start address of the Si-sized
    buckets;
    Low-level allocator allocates Ni buckets of size
    Si;
    Store the current position of the low-level
    allocator as the end address of the Si-sized
    buckets;
    Create stack STi and push Ni buckets into STi;
    // STi.pop() - remove and return a bucket from
    the top of STi
    // STi.push(b) - add bucket b to the top of STi
  end for
end function

function alloc (s)
  if not initialized
    call init
  end if
  for i = 1 to n
    if there exists STi for s (i.e. s == Si) and STi is
    not empty
      return STi.pop();
    end if
  end for
  call the shared storage allocator;
  return the result;
end function

function free (b)
// b is the pointer of the bucket to be freed
  for i = 1 to n
    If b ≥ the start address of the Si-sized buckets
    and b < the end adderss of the Si-sized buckets
      STi.push(b);
      return
    endif
  end for
  return b to the shared storage allocator
end function
```

그림 2. Budgeted 할당기의 의사 코드
Fig. 2. Pseudo code of the budgeted allocator.

allocator because it meets real-time requirements. That is, TLSF can allocate and free memory space in a bounded time and utilizes memory efficiently.

Fig. 2 shows the pseudo code of the budgeted allocator. Note that O_i, S_i, and N_i for $1 \leq i \leq n$ are predetermined at compile time. Object size S_i is provided by the user and N_i is determined by the budgeting method described in the section Ⅳ. Free buckets of size S_i are managed by a stack ST_i. So, there are n stacks $\{ST_1, ST_2, \cdots, ST_n\}$ for free buckets. The free list ST_i can be any type of dynamic data structure.

The function `init` calls a low-level allocator to reserve a memory region to be managed by the dedicated 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`[12] in C behaves as a low-level allocator. Most dynamic memory allocators reserve memory regions to be managed by themselves using a low-level allocator.

The function `alloc` calls the function init if it is not initialized. If there exists ST_i for the requested size s, that is s equals to S_i, a free bucket is popped from the top of stack ST_i and returned. Otherwise, it passes the request to the shared allocator and returns a part of the shared region.

The function `free` determines whether the bucket to be freed belongs to the dedicated region. It also determines the size of the bucket. To keep the same interface with conventional memory allocators, the only available information is the pointer b of the region to be freed. In order to perform the above tasks only with the pointer, the function init should allocate a continuous memory region using the low-level allocator with size $S_i \times N_i$ for the S_i -sized buckets and evenly split the allocated memory region into N_i buckets. Then it is checked whether the address b belongs to S_i-sized region. If the bucket to be freed belongs to the S_i-sized buckets, it is added into ST_i. If the bucket to be freed belongs to the shared region, it is passed to the shared allocator to be freed.

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[8]. The user-driven solution may suffer from human errors[8]. Lack of information of run time behaviors inhibits applying the compiler-driven solution to the budgeting method. Thus, we select the profile-driven solution, which is widely used for customizing memory allocators[3, 6, 8, 11].

The memory profile $P_i(t)$ is obtained by post-processing the memory trace. $P_i(t)$ denotes the number of buckets in use of size S_i at time t. To generate the trace, the application is compiled and run with a conventional general-purpose allocator that contains augmented code 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 profiled according to the trace. When O_i is requested at time t, $P_i(t)$ becomes $P_i(t-\Delta t)+1$, where Δt is the time when object O_i is either freed or allocated lately. If O_i is freed at time t, $P_i(t)$ becomes $P_i(t-\Delta t)-1$. Otherwise, $P_i(t)$ keeps $P_i(t-\Delta t)$.

1. Problem Formulation

The main objective of determining N_i is to minimize the size of the heap storage. Eq. (1) shows the required size $R(t)$ of the storage at time t.

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

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, which cannot be used for other purposes. 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[3]. 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 value 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. However, no fragmentation occurs in the dedicated region when our budgeted memory allocator is used. 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 as N_i increases as shown below.

Eq. (2) shows $U(t)$ in a different form

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

For budgeted allocators, Eq. (1) is rewritten as below

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

The superscript B denotes that each variable is for budgeted allocators. Now, $U^B(t)$ can be written as follows

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

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 to $P_j(t)$ where $i \neq j$. If $N_i \leq P_i(t)$ for every t and i, $U^B(t)$ is always smaller than $U(t)$. However, our goal is to have smaller $U^B{}_{max}$ than U_{max}. That is,

$$U^B{}_{max} \leq U_{max} \tag{5}$$

Note that if $N_i > P_{i,max}$, $(N_i - P_{i,max})$ buckets in the dedicated region are useless. So the following condition should be enforced too:

$$N_i \leq P_{i,max} \text{ for every } i \tag{6}$$

where $P_{i,max}$ is the maximum of $P_i(t)$.

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

$$\sum_i (S_i \times N_i) \tag{7}$$

N_i should be maximized while $U^B_{max} \leq U_{max}$ from Eq. (5) and $N_i \leq P_{i,max}$ from Eq. (6) for every i.

To make it easy to understand the formulation, an illustrative example is shown in Fig. 3. Fig. 3a and 3b show $U(t)$ and $U^B(t)$, respectively, when only N_1 is non-zero and N_2 and N_3 are zero. As $U(t)$ is summation of $S_i \times P_i(t)$ for every i, the graph is illustrated as split regions. In this example, N_1 is set to $P_1(t2)$ when $U(t2)$ is U_{max}. Even though it does not cause $U(t2)$ increase, it causes $U(t1)$ 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.

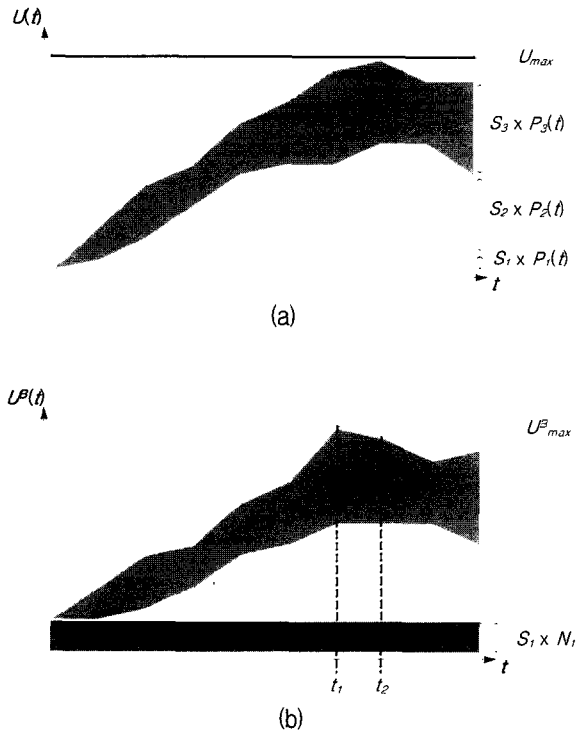


그림 3. (a) $U(t)$와 (b) $U^B(t)$의 예
Fig. 3. An illustrative example of (a) $U(t)$ and (b) $U^B(t)$

2. Solution

This problem cannot be solved within the polynomial time by an exhaustive algorithm. From Eq. (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,···, N_n } that makes the size of the dedicated region largest while maintaining $U^B_{max} \leq U_{max}$. 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 pseudo code is shown in Fig. 4. This algorithm can solve the problem within polynomial time because its complexity is proportional to n.

Every time t, following Eq. (8) should be satisfied.

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

Inserting Eq. (4) to Eq. (8) becomes

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

for every i and t. Now, a sub-optimum N_i is computed by assuming that N_j ($i \neq j$) are fixed priori. The following assumptions are used to compute sub-optimum N_i:

```
begin
   for i = 1 to n compute P1,max end for
   compute Umax

   for i = 1 to n
      compute M1,min
                  Mi(t) = Umax
                        - Σ_{j=1}^{i-1} (Sj × MAX(Nj, Pj(t)))
          where         - Σ_{j=i+1}^{n} (Sj × Pj(t))
      Ni = MIN(Pi,max, ⌊Mi,min / Si⌋)
   end for
end
```

그림 4. Greedy 알고리즘의 의사 코드
Fig. 4. Pseudocode of the greedy algorithm.

1. S_i is ordered such that $S_i > S_j$ 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 ($1 < j \leq n$) is zero. Then, Eq. (9) can be rewritten as Eq. (10)

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

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

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

that is always true by the definition of U_{max}. In other words, if $N_1 \leq P_1(t)$, it is always true that $U^B_{max} \leq U_{max}$.

However, if $N_1 > P_1(t)$, from Eq. (10)

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

Note that the right hand side is denoted by $M_1(t)$ that is a function of t. That is,

$$N_1 \leq \frac{M_1(t)}{S_1} \quad (13)$$

for every t. Therefore, if

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

for every t, $U^B_{max} \leq U_{max}$. In summary,

$$N_1 \leq \frac{M_{1,\min}}{S_1} \quad (15)$$

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

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

In general,

$$N_i = MIN\left(\left\lfloor \frac{M_{i,\min}}{S_i} \right\rfloor, P_{i,\max} \right) \quad (17)$$

where

$$M_i(t) = U_{\max} - \sum_{j=1}^{i-1} (S_j \times MAX(N_j, P_j(t))) - \sum_{j=i+1}^{n} (S_j \times P_j(t)) \quad (18)$$

Our experimental results in Section V shows that our heuristic algorithm is much faster than the exhaustive search algorithm and results of $\{N_1, N_2, \cdots, N_n\}$ are close to the optimum values.

V. Experiments

Table 1. shows the comparison results of the budgeting algorithms. Our proposed greedy algorithm is up to 15 thousand times faster than the exhaustive algorithm while the result is 99.8 % of the optimal solution. Note that if n and $P_{i,max}$ are increasing, the execution time of the exhaustive algorithm increases exponentially. Considering that most of embedded systems use more than five dynamically managed objects, the exhaustive algorithm would be infeasible for such a large number of objects.

The budgeted allocator is compared with previously proposed nine dynamic allocators and one static allocator. The nine dynamic allocators include the six allocators in [9] named from test1 to test6, the Kingsley (DJGPP in [9]), the Lea[5], and TLSF[1]. A simple static allocator is designed for comparison purpose.

표 1. Budgeting 알고리즘의 비교
Table 1. Comparison of budgeting algorithms.

n		4	4	5
$P_{i,max}$		10	20	10
Execution time (ms)	Exhaustive	7391	33781	157671
	Greedy	7	8	10
$\sum_i (S_i \times N_i)$	Exhaustive	15520	34688	16016
	Greedy	15520	34688	15984
	Ratio	100 %	100 %	99.8 %

For the performance evaluation of memory allocators, there are mainly two approaches to generate inputs: standard benchmark applications and synthetic workload models[1]. As mentioned in [1], the requirements of real-time systems are different from those of the standard benchmark applications. Thus, the synthetic workload models[14] are used to generate worst-case workload.

Generally dynamic allocators request predetermined size of storage M to a low-level allocator when a memory allocation for an object is needed. Then they manage the allocated storage M for further memory requests. Once M is full, dynamic allocators request additional storage to the low-level allocator. The predetermined sizes are usually 2^k bytes where k varies depending on the implementations. For example, test4 and test5 always request fixed size of storage. In this experiment 2^{14} is used. Also test1 requests various sizes 2^{n1}, 2^{n2},$\cdots$, and so on. On the other hands, Lea allocator may request arbitrary sizes. So, it is obvious that the performance of memory allocators is largely depending on the profile of workloads.

Five different types of workloads are used for the experiment depending on the size of the objects. Table 2. shows the profiles of five different workloads. Every workload has only four types of objects, that is n = 4, and the maximum number $P_{i,max}$ of allocated object O_i at the same time is limited to ten: that is $P_{i,max}$ = 10. All objects in workload1 are 2^k sized and larger than the predetermined size of storage M = 2^{14} for test4 and test5. All objects in workload2 are 2^k sized but smaller than M. Workload3 and workload4 are composed of non-2^k sized objects. All objects in workload3 are larger than M and all objects in workload4 are smaller than M. Workload5 is composed of mixture of objects in other four workloads. These workloads are synthesized such that every allocator used in this experiment has at least one workload profile that is optimum to the allocator.

Two aspects of each allocator are analyzed: memory efficiency, and code size. The memory efficiency is measured by the maximum size R_{max} of required storage $R(t)$. If an allocator A has smaller R_{max} than another allocator B has, 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 is measured as R_{max}. Allocator TLSF is an exceptional case because it simply fails the system if the pre-requested storage M is not sufficient for the

표 2. S_1, S_2 S_3 S_4으로 표시된 크기의 객체를 4개씩 가진 각 workload의 profile

Table 2. Profiles of workloads where every workload has four types of objects whose sizes are denoted by S_1, S_2 S_3, and S_4.

S_i	S_1	S_2	S_3	S_4
Workload 1	131072 (2^{17})	65536 (2^{16})	32768 (2^{15})	16384 (2^{14})
Workload 2	2048 (2^{11})	1024 (2^{10})	512 (2^9)	256 (2^8)
Workload 3	81936	80176	64032	49472
Workload 4	2952	1978	1040	528
Workload 5	131072 (2^{17})	80176	1024 (2^{10})	528

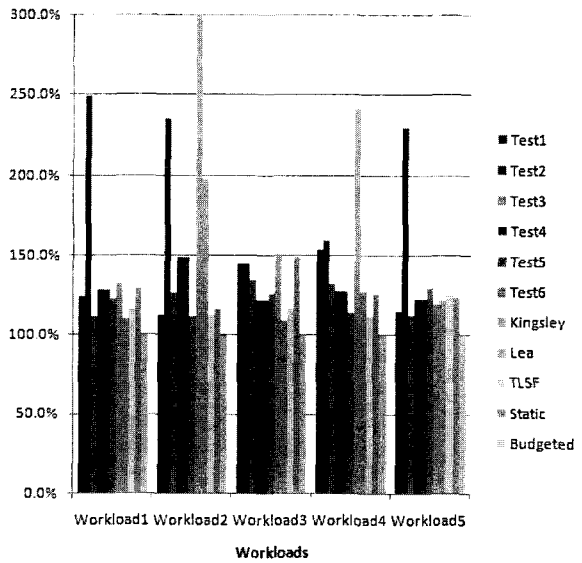


그림 5. 각 workload에 따른 메모리 할당기들의 메모리 효율성

Fig. 5. Memory efficiency of memory allocators for different workloads.

following memory requests. For TLSF, R_{max} is measured by trial and errors. That is, a workload is iteratively simulated with smaller heap storage until TLSF fails to respond a memory request.

Fig. 5 shows the memory efficiency of the eleven allocators including the budgeted allocator. The y-axis is the relative size of R_{max} where R_{max} of the budgeted allocator is set to 100%. As can be seen, the budgeted allocator is always more efficient in using memory than other ten allocators are. Lea allocator shows steady and good memory efficiency throughout the five workloads. Nevertheless, the proposed budgeted allocator requires 8.9%~49.5% less memory than Lea allocator does.

Note that R_{max}'s of different allocators vary with the simulated workloads as mentioned before. Test1 performed more efficiently in the 2^k small and mixed scenarios; Test3, in the 2^k large and mixed scenarios; Tests 4 and 5, in the arbitrary large scenario; Test6, in the 2^k small and arbitrary small scenarios; and TLSF, in the 2^k small and arbitrary small scenarios. We were not able to form a scenario where Test2 and the Kingsley allocator performed most efficiently because Test2 and the Kingsley allocator are types of fast allocators. The efficiency of Test4 and Test5 was always the same, but their execution times were different.

Fig. 6 shows the code size of the memory allocators. The code size of each allocator is measured after compiling them with ARM RVCT2.2[10]. The code size of the budgeted allocator is quite large because it is composed of two allocators. However, the code size of memory allocators is usually much smaller than the required memory size R_{max} and the code size overhead becomes negligible. Fig. 7 shows R_{max} plus the code size of memory allocators when S_i of workload5 is changed. To figure out the R_{max} when the code size of memory allocators is negligible, S_i of workload5 is reduced. For example, R_{max} plus the code size of the budgeted allocator is 8 KB when S_i of workload5 is

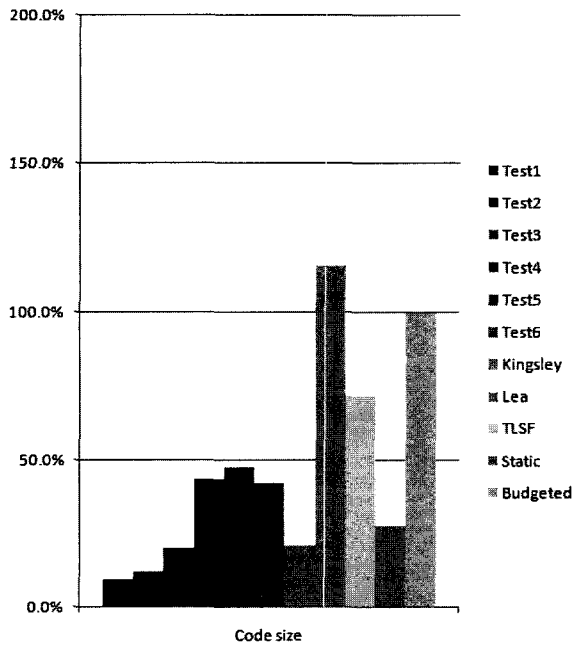


그림 6. 메모리 할당기들의 코드 크기
Fig. 6. Code size of memory allocators.

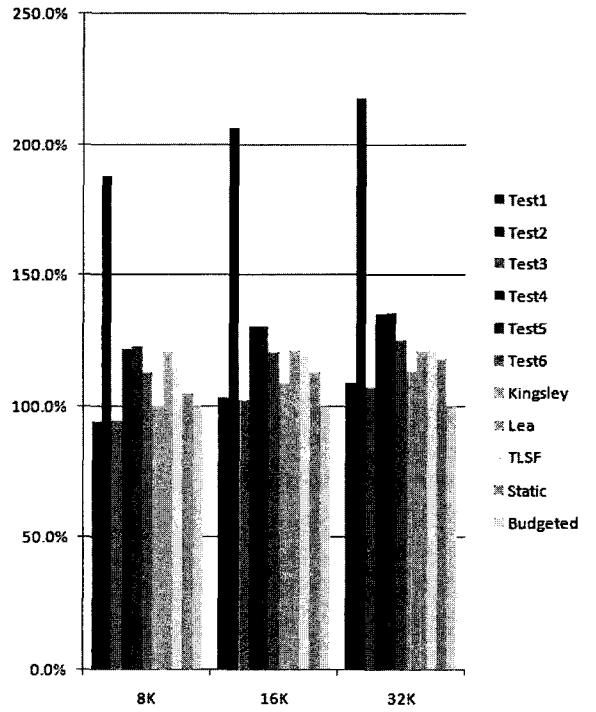


그림 7. 표 2 workload5의 S_i가 약 200배 줄었을 때 각 메모리 할당기들의 R_{max}와 코드 크기의 합. X축은 S_i값에 따른 각 메모리 할당기들의 R_{max}와 코드 크기의 합을 나타낸다.
Fig. 7. Required memory size R_{max} plus code sizes of various memory allocators when S_i of workload5 in Table 2 is reduced by about 200 times. The x-axis shows the R_{max} plus code size of the budgeted allocator for different S_i of workload5.

reduced by about 200 times from the Table 2. Once R_{max} is larger than 16 KB, the required memory size of the budgeted allocator becomes the smallest. Considering that R_{max} for normal applications is usually more than 1 MB, the code size overhead of the budgeted allocator is negligible.

VI. Conclusion

A dynamic memory allocator called a budgeted allocator is proposed to enhance memory efficiency for embedded applications running continuously during the whole lifetime of the system. The budgeted allocator has a dedicated storage for predetermined sizes of objects. Only exact size of objects can be allocated in the dedicated storage and all other objects are allocated in the shared storage that is managed by a general purpose dynamic memory allocator. In this way, the overall fragmentation can be significantly reduced. The budgeted allocator is compared with ten previously proposed allocators for five different workloads. Our experiments show that the budgeted allocator always requires least memory size for all cases. It requires 8.9 %~49.5 % less memory than the Lea allocator that shows steady and good memory efficiency throughout the experiments.

Reference

[1] M. Masmano *et al.*, "TLSF: a New Dynamic Memory Allocator for Real-Time Systems," in *Proc. of Euromicro Conference on Real-Time Systems*, pp,.79-86, Catania, Italy, June, 2004.

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

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

[4] H. Zhe *et al.*, "Design and Realization of Efficient Memory Management for Embedded Real-Time Application," in *Proc. of International Conference on ITS Telecommunications*, pp.174-177, Chengdu, China, June, 2006.

[5] D. Lea, A memory allocator, Available: http://g.oswego.edu/dl/html/malloc.html

[6] E. Berger, B. Zorn, and K. McKinley, "Reconsidering Custom Memory Allocation," in *Proc. of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.1-12, Seattle, USA, November, 2002.

[7] E. Berger, B. Zorn, and K. McKinley, "Composing High-Performance Memory Allocators," in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.114-124, Snowbird, USA, June, 2001.

[8] M. Seidl and B. Zorn, "Segregating Heap Objects by Reference Behavior and Lifetime," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.12-23, San Jose, USA, October, 1998.

[9] Delorie software, Available: http://www.delorie.com

[10] RVCT 2.2. Available: http://www.arm.com

[11] M. Tofte and J. Talpin, "Region-Based Memory Management," in *Information and Computation*, Volume 132, Issue 2, pp. 109 - 176, February, 1997.

[12] Y. Hasan and J. Chang, "A Hybrid Allocator," in *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pp.214-222, Austin, USA, March, 2003.

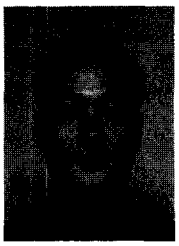
[13] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator, " in *Proc. of USENIX Technical Conference*, pp.87-98, Boston, USA, June, 1994.

[14] B. Zorn and D. Grunwald, "Evaluating Models of Memory Allocation," in *ACM Transactions on Modeling and Computer Simulation*, Volume 4, Issue 1, pp.107-131, January, 1994.

저 자 소 개

이 중 희(정회원)
2000년 서울대학교 컴퓨터공학과 학사 졸업.
2003년 서울대학교 컴퓨터공학과 석사 졸업.
2003년~현재 삼성전자 연구원

<주관심분야 : ESL design, MPSoC, SoC architecture, embedded system>



이 준 환(정회원)
1991년 연세대학교 전자공학과 학사 졸업.
1998년 Univ. of Michigan, Ann Arbor 석사 졸업.
2002년 Univ. of Michigan, Ann Arbor 박사 졸업.
1991년~1995년 삼성전자 연구원
2003년~현재 삼성전자 연구원

<주관심분야 : MPSoC, SoC architecture, C-level system modeling for fast hardware and software co-simulation, system-level power analysis and optimization, behavioral synthesis, and high-level testing>