

15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2010). July 2010. Bilbao (Spain).

A Constant-Time Region-Based Memory Allocator for Embedded Systems with Unpredictable Length Array Generation

Jordi Sánchez, Ginés Benet, José E. Simó

jorsanpe3@upvnet.upv.es ; gbenet@disca.upv.es ; jsimo@disca.upv.es

Instituto de Automática e Informática Industrial

Universidad Politécnica de Valencia

P.O.Box 22012 46080 Valencia, Spain

Abstract

This paper describes an implementation of a region-based memory manager that performs the allocation and de-allocation in constant-time. Also, additional functionality for generating arrays which can grow arbitrarily has been implemented. Thus, external fragmentation is overcome, and the appearance of memory leaks has been considerably reduced. All these features make this allocator specially useful for computer vision applications. The main goal has been to replace the general purpose allocator on some critical places in order to remove fragmentation and improve performance. The use of Regions also reduces programmer burden. The main disadvantage of this method is that it leads to a higher memory consumption peaks than general-purpose allocators. However, this is not a serious problem in systems without hard memory constraints. In this paper, the performance of our approach has been compared against an architecture optimized general purpose memory allocator in a realtime vision application.

Acknowledgments

This work has been partially supported by SENSE project (Specific Targeted Research Project within the thematic priority IST 2.5.3 of the 6th Framework Programme of the European Commission: IST Project 033279), and by the Spanish Government by the complementary funding TIN2007-30367-E

SENSE



A Constant-Time Region-Based Memory Allocator for Embedded Systems with Unpredictable Length Array Generation

Jordi Sánchez, Ginés Benet, José E. Simó
jorsanpe3@upvnet.upv.es ; gbenet@disca.upv.es ; jsimo@disca.upv.es
Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia
P.O.Box 22012 46080 Valencia, Spain

Abstract

This paper describes an implementation of a region-based memory manager that performs the allocation and deallocation in constant-time. Also, additional functionality for generating arrays which can grow arbitrarily has been implemented. Thus, external fragmentation is overcome, and the appearance of memory leaks has been considerably reduced. All these features make this allocator specially useful for computer vision applications. The main goal has been to replace the general purpose allocator on some critical places in order to remove fragmentation and improve performance. The use of Regions also reduces programmer burden. The main disadvantage of this method is that it leads to a higher memory consumption peaks than general-purpose allocators. However, this is not a serious problem in systems without hard memory constraints. In this paper, the performance of our approach has been compared against an architecture-optimized general purpose memory allocator in a real-time vision application.

1. Introduction

Memory management is a critical issue on computer vision algorithms. These systems work on long term executions or even on a permanent way. Vision systems make intensive use of memory, as they have to work with large amounts of data. Computer vision algorithms performance rely strongly on memory access times, therefore, on caching mechanisms and memory contiguity. Some of this data will be dynamic and, thus, a good management of it will be necessary. The most extended general-purpose memory allocators (GPMA) rely on data structures known as *heaps*. On *heaps*, free blocks and assigned ones are indexed, keeping track of their location, size and other features.

However, GPMA suffer from the largely studied fragmentation problem. Fragmentation occurs when a block surrounded by assigned chunks of memory is freed; then,

a hole appears. Having holes is a problem, as the total free memory is bigger than the maximum memory we can assign at once. In addition, *heaps* must search for a hole big enough to fit the required data. In long executions, lots of holes lead to slower searches and to a reduction of system performance or even to its instability unless some sort of garbage-collection algorithm is used. Such algorithm may introduce undeterministic processing overload and, hence, it is not interesting from a time-constrained-system point of view.

Some commercial systems for Digital Signal Processors development, (as the Analog Devices used in our work) have kernels with libraries for memory management based on *next-fit* policy. This policy is simple; it keeps a linked list of free memory chunks and, when the user requests memory, it searches for the next chunk of memory where the data fits, starting from the last chunk where memory was allocated. Bays [1], made a study in which stated that *next-fit* performs worse than *first-fit* and *best-fit* in fragmentation terms¹. Despite not being the best memory allocator, *next-fit* provided by Analog has been written in assembler and has been optimized to work on Analog's processors. In fact, we found that the best performance with low fragmentation was achieved by this allocator. For this reason, this architecture-optimized GPMA has been chosen to be compared against the region-based memory allocator (RBMA) which is presented in this paper, because the platform used in our work is the Analog's Blackfin BF561 dual core DSP.

On vision systems, efficient data access policies are required as a must have. Working with images causes huge quantities of memory accesses and, if not correctly managed, the low performance would cause vision algorithms to be useless. Physically contiguous arrays are the solution to optimize memory accesses on incremental reads, typical of vision systems. This way of storing data reduces cache misses when writing or reading memory, thus improving performance. In order to create an array on a general purpose memory allocators, two parameters are

¹We do not intend to explain here the policies for memory management. For a deeper study on memory allocators see [9].

needed at the time the memory is requested: the *size* of each element and the *number of elements* the array is going to contain. However, this is a problem as in some situations it is impossible to determine how many elements will be used a priori.

By using regions, problems above mentioned can be solved. A *region* is a chunk of memory used to serve memory requests. The main property of a region is that all the objects it contains are freed at the same time, and it is not possible to free individual objects. Regions have been an alternative to garbage-collection or explicit deallocation memory management policies. In [4, 12, 5] some examples of implementations of region-based memory allocators (RBMA), are reported. Also, Berger et al.[2] made a very interesting study on custom memory allocation algorithms and stated that, in general, custom memory allocators provide lower performance than general-purpose ones. However, custom allocators can provide an important speed-up on specific applications which can make use of them. In [6] an hybrid allocator is presented, where different policies can be used within the same memory space. Thus, this allocator can be tuned to meet application needs, in terms of memory consumption against performance. One of the most extended memory managers used today is the developed by Lea [8], and performs better than many custom memory allocators. Recently, in [10] a new constant-time memory allocator that allows its use on real-time systems has been described. Also, Hasan [7] has published a recent study about the upper bounds of memory storage for two different allocators: a general allocator that can allocate memory blocks anywhere in the available heap space, and a more economical *first-fit* allocator constrained by the address-ordered policy.

The use of a RBMA allows the user to avoid keeping complex structures to manage the memory allocation policies above mentioned. As a counterpart, the RBMA exhibits a higher memory overload, as regions do not allow individual objects to be freed. Thus, deprecated data cannot be deleted until the whole region is cleared. As already indicated, this could be a problem on systems with hard memory constraints. However, this is not likely to be the case of vision systems, where usually high performance processors are used.

In this paper, an implementation of a RBMA for its use on vision systems is presented. Despite this RBMA has been developed under the scope of a vision system, this implementation can be applied to other kinds of systems as well. Also, a new method to generate physically contiguous arrays with an undetermined size is provided. This allows faster memory accesses due to reduction of cache misses and, in some cases, can be used to avoid the use of more complex and inefficient data structures.

This paper is organized as follows. In Section 2 we describe the background project where our custom RBMA has been developed. In Section 3 a deep description of this allocator is given. Finally, Section 4 shows the results obtained using this RBMA, compared against the results

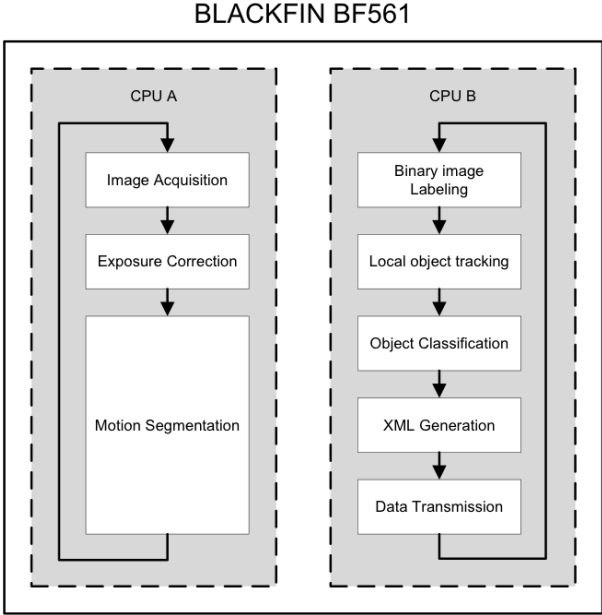


Figure 1. SENSE algorithm overview and execution pipeline for each core on processor Blackfin BF561

of a GPMA and gives a measure of the overallocation of the memory.

2. Scope and Motivation of the Proposal

The RBMA has been developed under the framework of computer vision project called SENSE. This project is a surveillance application, composed of several processing layers, each one running on a different processor. More info on SENSE can be found in their web page [11]. There are two low level layers processing audio and video. These layers feed the high level unit, which analyses the data and detects the state of the environment. The RBMA has been implemented in the scope of the video low level layer. This layer is in charge of detecting moving objects on scene and classifying them (people or luggage). The high level processing unit receives information about the classes and position of the objects and tracks them. In order to have a consistent trajectory inference, a good framerate has to be provided by the low level video processing layer.

Fig.1 depicts the block diagram of the SENSE video processing algorithm and the distribution of the phases of the algorithm between the two cores of the used processor. Notice that the first phases of the algorithm run on core A and the last ones on core B. As depicted, each processor has a global execution cycle, which we call *system cycle*. For example, for core B, the system cycle starts with the labeling of the image and ends when it has transmitted the results of the analysis.

Memory management has been one of the most com-

plex parts of the SENSE vision system. As seen on Fig.2, a distinction has to be made between different memory types:

- *Static memory.* The major part of the memory has been assigned as static memory. Specifically, all the images used by the system are stored statically. First, there is no point on allocating and releasing memory for each different time. Instead, the space is reused. But there is another reason for this, which is the most important one. The images must be distributed strategically over the different memory banks in order to get the best performance. Therefore, each phase will read an image stored on a memory bank and will write the results on a space allocated on a different bank.
- *Fixed block size dynamic memory.* The application deals with objects of the same type and, so, of the same size. We can avoid fragmentation produced by these objects by using memory pools. Memory pools are structures that manage blocks of memory with the same size.
- *Variable size dynamic memory.* There are many buffers which will have variable sizes. For example, the contours of the objects will vary from one frame to another, as the geometrical properties will be surely different. All the auxiliary buffers used by the vision algorithms, the geometrical properties of the objects and other non-fixed size dynamic memory are considered here.

This paper is focused on variable size dynamic memory. This kind of memory supports complex geometrical treatment as well as tracking algorithms. Although allocated buffers have small sizes, the number of allocations and deallocations can be very high. This condition makes the system to suffer from great fragmentation and is one of the reasons that motivated the development of our region-based allocator.

3. Description of the proposed RBMA model

Vision systems, like many other systems, have a cyclic nature. The whole process is a cycle which is repeated with each incoming frame. According to this, we can distinguish two kinds of data: *temporary* or *cyclic* data, which lasts for only a cycle, and *persistent* data, which lasts for many frames. *Temporary* data uses auxiliary buffers for image processing and geometrical data of the objects. Object geometry is intrinsic to a frame and gets deprecated when a new frame comes into the system. *Persistent* data corresponds to the high level data used to describe the scene. For example, in a surveillance system where people is tracked, each person is represented by a high level structure (id, position, velocity, size, etc...) and, as long as that person remains visible, this data will remain on the system. Thus, *persistent* data can always be

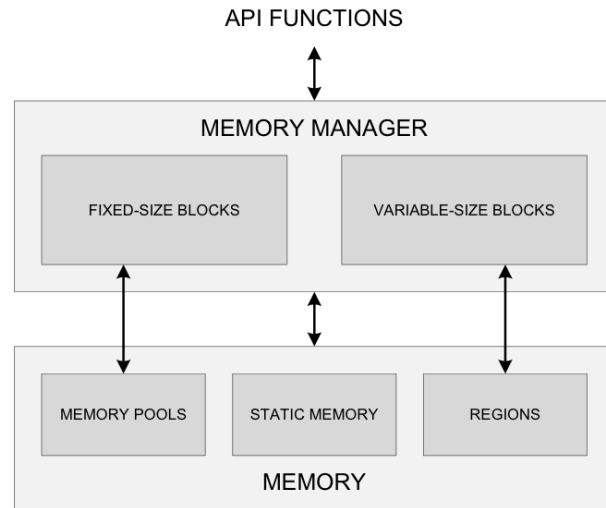


Figure 2. Structure of the memory manager of the vision system for the different memory types

stored as *fixed block size memory* and *temporary* data as *variable size* memory.

As stated before, if we store temporary and persistent data on the same memory structure the memory will suffer from fragmentation. Indeed, storing temporary data can produce fragmentation. Hence, the first solution to the fragmentation problem is to split persistent and temporary data into two different memory allocators, treating them separately.

Using the same allocator to store temporary data greatly simplifies their treatment. As we know that all data is going to be useless at the same time (at the end of the cycle), we can get rid of the GPMA and use regions. Unlike GPMA, in region allocators all the data stored is released at the same time. Buffers cannot be freed individually. This is the main feature (and also its main advantage) of the regions against general purpose allocators. Unlike GPMA, which need to perform a search over the memory to find a chunk large enough, on RBMA, the allocation is performed in a single step. This is also true for releasing operations, which are done on a single instruction. Indeed, GPMA requires the programmer to cover all the possible situations of the program that could produce a memory leak, so the region allocators are also much safer on this point. One disadvantage of the RBMA is that they keep all the allocated memory until released at the end of the cycle, whether it is still referenced or not, thus, using more memory than needed. This may cause some problems on systems with memory limitations.

The structure of the presented RBMA is depicted on Fig.3. We have an structure holding four values: *starting address* of the region, *address* of the free memory, *size* of the region and *used memory*. Memory is assigned by increasing the free memory pointer. When a region

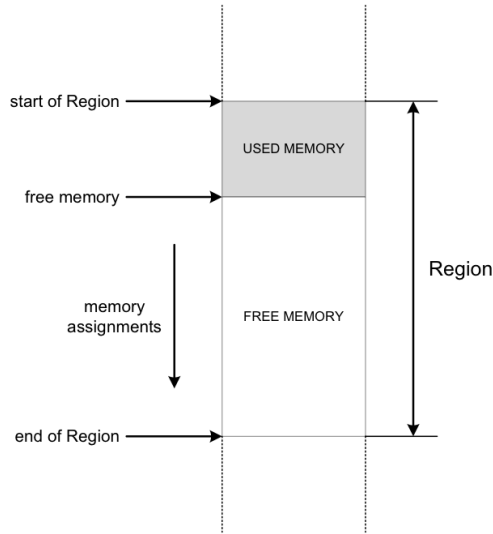


Figure 3. Structure of the proposed RBMA.

is cleared, the free pointer is updated to point to the start of the region. All these operations are very efficient and of constant (and hence predictable) cost. When allocating memory, only a sum is needed; when freeing memory, just an assignment is needed. The following fragment of code illustrates these operations:

```
region_t* new_region(int memsize);
int delete_region(region_t *region);
void *ralloc(region_t *region, int size);
void rfree_all(region_t *region);
void *rnew_array(region_t *region,
                 int itemsize,
                 int *maxsize);
void rclose_array(region_t *region,
                 int nitems);
```

A C-based interface is provided for working with regions, including operations for allocation and deallocation of objects, array generation and region creation and destruction. The interface is very simple and, as the programmer does not need to free the objects, the programming burden is reduced and, so, the occurrence of a memory leak due to an uncovered program situation.

A very interesting issue with the proposed implementation of the RBMA is that the delivered memory is always located to the bottom of the region. We have taken profit of this fact to implement a function to generate arrays. One of the problems of GPMA is that a specific size has to be provided to the memory manager at allocation time. Hence, arrays cannot grow arbitrarily with this allocators. A good example of a situation when arrays must grow arbitrarily is the case of image labeling algorithms and, more specifically, the algorithms based on contour tracing technique, as described in [3]. In particular, this labeling algorithm performs a search over a binary image to find an object. When it finds an element, a second search is performed to find all the pixels that belong to the contour of that object. Contours are later used, for example, to perform pattern recognition. These algorithms

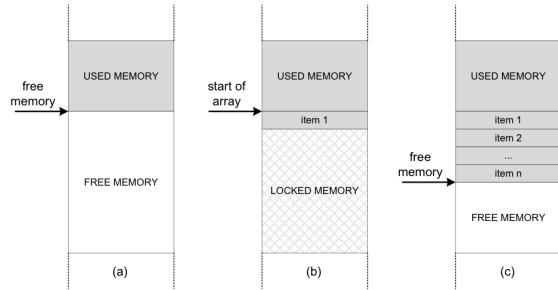


Figure 4. Array generation process in the proposed algorithm.

will surely need an efficient data access policy. As stated above, physically contiguous arrays are the most efficient way of accessing data incrementally, as this reduces cache misses.

Fig.4 depicts the proposed method to generate arrays without the need of knowing how many elements they are going to have. We take the starting element of the array as the address of the free memory (a). As the array grows we simply update the free memory pointer with the size of the items (b). Once the array is completed (c), it is closed and the free memory pointer is rounded to a relevant boundary, in our case a 4 bytes one.

There are two different methods to perform array growing. The first one is to update the free memory pointer as the array grows. This implementation provides higher safety when generating the array, but introduces some extra processing overload, as we have to include the update operation of the free pointer for each element. The second method is to ignore memory allocation until the whole array is generated, and then update the memory pointer, indicating the final size of the array. This last way is faster but unsafer, considering that the system is accessing memory addresses that are still not managed by the allocator. Another source of problems arises when there is not enough memory on the region for the array. In this case access to protected memory regions could be done while generating it.

Despite of this inconvenient, the second method is the one we have decided to use on our SENSE system [11] because of its higher performance. Consequently, to ensure the safety of the region, several factors must be taken into account. As stated above, array generation can be a source of errors if not correctly managed. When a new array is generated, we have to check how many memory remains on the region in order to narrow its size. Some other constraints have to be met when generating an array:

- First, the generation of an array must be done in a single cycle, before the region is cleared.
- Second, there cannot be any other memory allocation process while the array is growing, so we cannot mix data.

- And third, as in all arrays, all the allocated data must be of the same type.

In conclusion, we can understand array generation as a temporal allocation of the whole remaining memory on the region. When the array is finished, we release the memory we have not used.

The RBMA can be extended to a multithreaded application. Each thread should make use of its own region, despite several threads could be sharing the same one. In that case, additional considerations about critical regions should be taken into account.

4. Experimental results

In this section, we present the experimental results obtained with the described RBMA on runtime. The system has been implemented with the VisualDSP 5.0++ editor and runs on a Blackfin BF561 600MHz dual-core processor with 64MB of external RAM, 128KB of shared L2 data memory, and 64KB of L1 separated instruction/data memory for each core. As stated above, the kernel used has been VDK, which implements the basic functionalities of a kernel (threads, semaphores, driver interface, and so on). We have compared the results obtained with the above described RBMA against the pre-built system one. We have run the system under three different stress conditions, in order to obtain a better view of the performance improvement. For the *low* system overload execution, we had an average of two objects on the scene. The *medium* system overload was caused by a transit of seven to eight objects. Finally, the *highest* overload was produced by 20 or more transiting objects.

First of all we have compared the execution times of the GPMA, using the next-fit strategy, and our RBMA. These times have been obtained as the total time spent by the application allocating and deallocating memory over each frame. In order to compare the the performance of the RBMA against GPMA in an exact way, the results have been measured using both allocators on the same execution of the same code. That is, all memory assignment operations were performed twice, once for each allocator. While not affecting the system performance, this way of measuring the allocators performance is the most exact one. The allocators are called the same number of times and the memory requirements are identical. Fig.5 illustrates these results.

As can be seen in the Fig.5, the proposed RBMA has better execution times over any system overload in comparison to the general purpose one. Times achieved by the RBMA are approximately three times better than that obtained with the GPMA. Of course, time spent on memory allocation and deallocation instructions is quite low, and does not affect in a significant way the framerate of the system (about 133ms, corresponding to 7.5 fps). However, one of the problems of the non deterministic nature of this particular vision system is that as scene gets crowded, every phase of the application consumes more

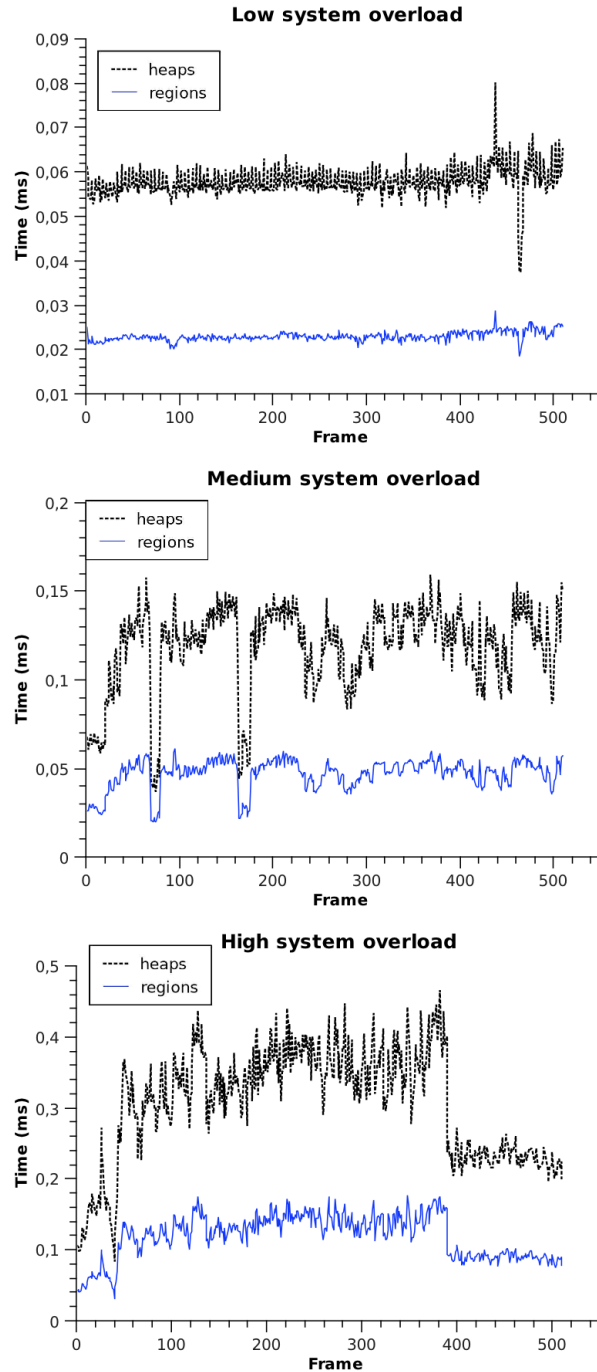


Figure 5. Total time spent by the video processing application allocating and deallocating memory over each frame. The figures compare the times using GPMA and using the proposed RBMA. Three different overload situations are showed in the separate figures.

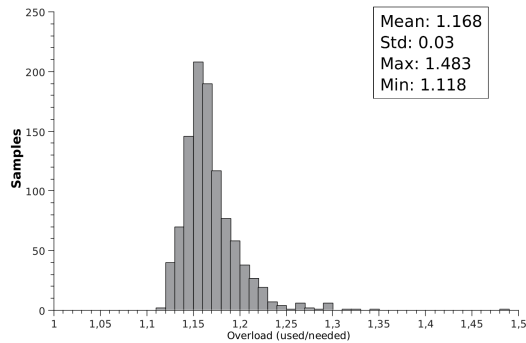


Figure 6. Histogram of memory overallocation produced during runtime using the RBMA. Memory overallocation is defined as the ratio $memory\ used / memory\ needed$.

time. The more objects are on a scene, the more features we have to extract, a higher number of objects must be classified, and so on. Notice this effect on Fig.5. Times under stress situations can raise up to 0.45 ms for GPMA, whereas the RBMA raises up to 0.16 ms.

Every memory allocator produces some *internal fragmentation*, because they allocate more memory than requested. This is usually due to the need of meeting architecture constraints. Structures are usually the main reason of internal memory fragmentation, as they are usually resized to architecture boundaries (typically 4 or 8 bytes). In our system, the buffers managed by regions were composed of basic types which are better packed than structures and cause minimal fragmentation. On our results, we have considered negligible the internal fragmentation.

Regions have the immediate benefit of simplifying system implementation as there is no need to keep track of every buffer to free it when necessary. The counterpart of regions is that they produce a higher *memory overallocation* due to the reason that individual objects cannot be freed from a region. We evaluate the *memory overallocation* as the relationship between memory used and memory needed. This parameter has been measured during runtime on our system and its statistical distribution has been presented in the Fig.6.

We have measured a mean overallocation of 1.168 with maximum peaks of 1.483. This could be a problem in applications which exploit intensively the memory. For this particular case, the total dynamic memory managed by regions was low enough to be affordable by the system. Taking into account the system capabilities (64 MB of RAM) and the memory usage depicted in Fig.7, having a memory overallocation of several KB is not significant. However, we have to interpret this result very carefully, as it is given in percentage of memory. If the memory consumption was too high, the overallocation could be unaffordable.

Regarding memory usage depicted on Fig.7, this, as well as memory overallocation depicted on Fig.6, have

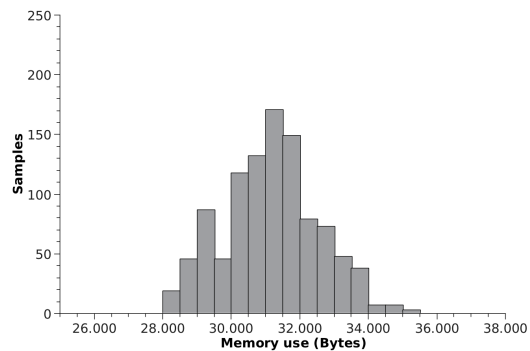


Figure 7. Histogram of the total memory usage during an execution over a crowded situation.

been measured under high stress conditions, in order to give an idea of how many variable-size dynamic memory the system is going to use. These results depict that the variable-size memory usage, in relation to the total memory needed by the system, is negligible.

5. Conclusions

A constant time RBMA has been presented. The features of this allocator makes it very attractive for its use on computer vision applications. RBMA's are a very good alternative to GPMA's in systems which can make use of them. They reduce programmer burden and, so, the possibility of appearing memory leaks. Results obtained show that performance is better than that provided by a GPMA. Also, fragmentation problems are overcome, which are a source of unpredictability and instability.

Also, a new method to generate physically contiguous arrays has been presented. A very simple interface for array generation has been given. Vision system developers will find very useful this functionality, which fits very well with the new labeling algorithms, based on contour tracing.

Finally, on the SENSE project, we have to state that the benefits of the allocator have come, besides performance, from a much better system stability. Having the different memory types separated reduces the complexity of the memory management, allowing more efficient algorithms and easier programming for vision system developers.

6. Acknowledgments

This work has been partially supported by SENSE project (Specific Targeted Research Project within the thematic priority IST 2.5.3 of the 6th Framework Programme of the European Commission: IST Project 033279), and and has been also co-funded by the Spanish research project SIDIRELI. MICINN: DPI2008-06737-C02-01/02).

References

- [1] C. Bays. A comparison of next-fit, first-fit, and best-fit. *Communications of the ACM*, 20(3):191–192, 1977.
- [2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proc. of the 17th ACM SIGPLAN conf. on Object-oriented programming, systems, languages, and applications*, pages 1–12, New York, 2002. ACM.
- [3] F. Chang, C.-J. Chen, and C.-J. Lu. A linear-time component-labeling algorithm using contour tracing technique. *Comput. Vis. Image Underst.*, 93(2):206–220, 2004.
- [4] D. Gay and A. Aiken. Memory management with explicit regions. In *Proc. of the ACM SIGPLAN conf. on Programming language design and implementation*, pages 313–323, New York, NY, USA, 1998. ACM.
- [5] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.*, 20(1):5–12, 1990.
- [6] Y. Hasan and J. M. Chang. A tunable hybrid memory allocator. *The Journal of Systems and Software*, 79(8):1051–1063, 2006.
- [7] Y. Hasan, W.-M. Chen, J. M. Chang, and B. M. Gharaibeh. Upper bounds for dynamic memory allocation. *IEEE Transactions on Computers*, 59:468–477, 2009.
- [8] D. Lea. A memory allocator. Web page, State University of New York at Oswego, 2000. <http://g.oswego.edu/dl/html/malloc.html>.
- [9] M. Masmano. *Dinamic Memory Management in Real-Time Systems*. PhD thesis, Universidad Politecnica de Valencia, 2006.
- [10] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *Proc. of the 16th Euromicro Conf. on Real-Time Systems*, pages 79–88, Catania, Italy, July 2004. IEEE.
- [11] SENSE Consortium. Smart Embedded Network of Sensing Entities. Web page: "<http://www.sense-ist.org>", (European Commission: IST Project 033279), September 2006.
- [12] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.