# The method, the tools and rationales for assessing dynamic memory efficiency in embedded real-time systems in practice

Christian Del Rosso
Nokia Research Center
Itämerenkatu 11-13, 00180
Helsinki, Finland
Email: christian.del-rosso@nokia.com

***Abstract*— A dynamic memory management system has to take care of the allocation and deallocation of memory blocks in a software system. Real-time embedded systems add some more constraints to the design and the implementation of dynamic memory management systems if compared with the PC world. An increasing number of features are added to embedded mobile devices, however, resources like dynamic memory are limited. In addition, in real-time systems, real-time deadlines must be respected and allocations and deallocations must be done in due time. In this paper we present a case study on evaluating dynamic memory management in embedded real-time systems. We have used a scenario-based approach and used a simulation environment to evaluate the performance of different dynamic memory management systems. Our contribution is to present a practical approach, the tools and the rationales to evaluate dynamic memory management in embedded real-time systems.**

## I. Introduction

Donald E. Knuth, in the Art of Computer Programming, Volume 1 [1] has given a good description of what to expect from dynamic storage allocation algorithms: ”We want algorithms for reserving and freeing variable-size blocks of memory from a larger storage area, where these blocks are to consist of consecutive memory locations. Such techniques are generally called dynamic storage allocation algorithms”.

The study of dynamic memory management systems started in the sixties [2], [3], [4], [5], and memory optimization was then one of the highest priorities at the design stage. At that time, limited memory demanded algorithms able to efficiently use the scarce resources available.

Although much research has since been done and the memory available has increased according to the Moore’s law, embedded real-time systems such as mobile phones must still be able to work with limited memory and memory efficiency is still one of the most important requirements. More and more features are included in the mobile phones but, on the other hand, the memory available is limited. While additional memory can be added, the cost of a product depends heavily on the amount of memory included.

In embedded systems and in particular in the system analyzed in this paper, there is no compaction of memory and requests are served using contiguous blocks of free memory. Additionally, only allocators with explicit references to the main memory are analyzed in this paper. Even though current research is focusing on garbage collection techniques, the performance of these allocators is still behind memory management systems with explicit reference to the main memory like C and C++ [6], [7], [8], [9].

Another sign that differentiates embedded systems from the PC world is the lack of virtual memory management. Most of the current embedded systems do not have the MMU (Memory Management Unit) and the address space is given by the amount of physical memory available. The heap size is fixed and there is no possibility to request additional memory, for example, by using the Unix *sbrk* function call.

Real-time embedded systems add additional constraints to the environment as real-time systems have time constraints that must be respected. Therefore, in analyzing and evaluating dynamic memory management systems for real-time embedded systems, memory efficiency and responsiveness are at the same level of importance. In the case a time deadline is missed, several failures can appear, for example the drop of a phone call.

Optimizing the dynamic memory management system for real-time embedded systems requires designing for memory efficiency and real-time constraints.

In this paper we describe an experience of evaluating dynamic memory management systems for embedded real-time systems. In our approach, as a starting point, significant scenarios are defined. The scenario-based approach defines the significant scenarios important for the analysis. Trace instrumentation was used to extract data from the running mobile phone software. We have introduced a set of metrics, proposed with the scope of evaluating dynamic memory management systems in embedded real-time systems. Data have been evaluated using a simulation environment where algorithms and different data structures were tested. The results from the simulation environment have been evaluated and analyzed.

## II. Dynamic Memory Management and Embedded Real-Time Systems

A dynamic memory management system is made of a data structure and its algorithms. The scope of such a system is to keep track of the memory blocks in use in the system but at the same time to optimize the usage of memory.

Different algorithms and heap layouts can be used to manage the dynamic memory system. Examples of general purpose algorithms that can be used are best-fit, first-fit, the Doug Lea algorithm [10] (which is used in Linux), and the Windows XP allocator [11]. On the other hand, custom algorithms for dynamic memory management have been used in different fields when the patterns of data allocations of the applications are known. Examples of custom allocators are the Apache web server allocator [12], the C++ Standard Template library allocator [13] and the GCC compiler allocator [14]. Custom allocators does not always perform significantly better than general purpose allocators and general purpose allocators have more advantages in terms of software maintainability and evolvability [15].

Data structures must also be considered for dynamic memory management systems. A dynamic memory management system can be made, for example, either of a single or multiple contiguous blocks of memory or made of segregated free lists. Different allocators can then be applied with different policies in different heap layouts. For example, Microsoft C++ uses one heap for allocations of less than 200 bytes, and a second heap for all other allocations [16]. The Doug Lea algorithm [10] manages small objects (smaller than 64 bytes) using exact size bins of a multiple of 8 bytes each; for larger objects it is a pure best-fit allocator and for very large requests (larger than 128K) it relies on the system memory mapping facilities (if supported).

In addition to the considerations above, we must include the requirements of our environment. In our case, when evaluating dynamic memory management systems we must consider the fact that we are analyzing embedded real-time systems. Limited memory, no memory compaction, and no virtual memory are properties of this system.

Real time constraints, or the ability of the system to respect time deadlines, are also one of the main concerns in the evaluation.

## III. Fragmentation and Memory Efficiency

Memory blocks are continuously allocated and deallocated from the dynamic memory, which leads to a phenomenon called fragmentation [17], [18], [19]. Fragmentation is the situation when the memory is divided into used and unused blocks. Because of high fragmentation the dynamic memory management system may not able to perform allocations and subsequently may negatively influence the stability of the whole system. For example, in figure 1, we have a case of a failure, because of fragmentation, in allocating a 100KB MMS even though the total free memory is 200KB

In order to improve the cohesion of memory, adjacent free blocks can be coalesced, but if the areas are not contiguous

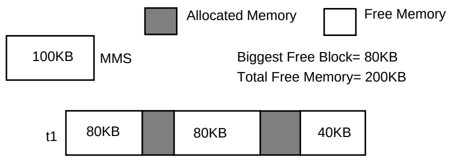

Fig. 1. Fragmentation

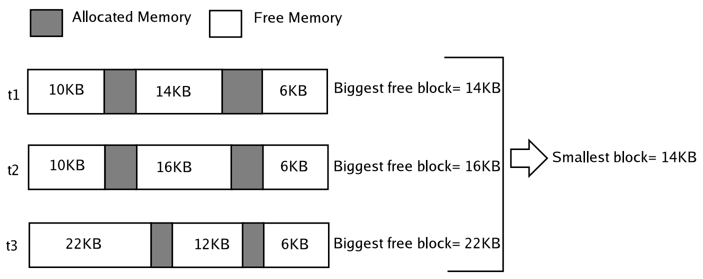

Fig. 2. Smallest-Biggest Block Metric

nothing can be done unless reallocation and compaction of the used blocks take place. A compaction of memory by rearranging the memory into free and used blocks can be implemented; however, real-time constraints make this approach unfeasible. Strict real-time deadlines must be respected and changing the pointers can leave the dynamic memory data structure in an inconsistent state.

The main concern in these systems is to limit the fragmentation problem by having large enough free blocks to serve future allocation requests and to assure real-time requirements. Metrics that measure fragmentation and memory usage efficiency in the context of embedded systems must be able to highlight those issues.

Fragmentation and memory efficiency metrics must measure the efficient of memory use. On the other hand, performance speed metrics must be used to establish whether the system is able to satisfy the deadlines and time requirements imposed: it is not pure speed that counts. In order to evaluate the fragmentation of the dynamic memory management system, we have defined a few metrics in the following subsections.

### A. Smallest-Biggest Block Metric (SBBM)

When serving a request, the main concern is to find a large enough free block. Because of fragmentation, a large enough free block may not be found even though the aggregate amount of free memory is higher than the request. In the case that a free block is not found, a failure arises.

The Smallest-Biggest Block Metric (SBBM) gives the amount of memory of a request that will always succeed in the scenario. If a larger amount than that given by the metric is requested, the allocator could fail, see figure 2. The metric is expressed in Bytes.

To extract the metric, the free list is analyzed during the entire simulation, and the biggest free block value in the free list is recorded. At the end of the simulation, we select the smallest value among the biggest free blocks recorded.

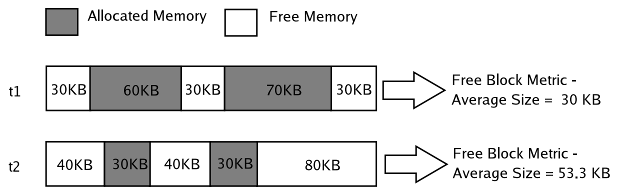

Fig. 3. Free Block metric

### B. *Free Block Metric - Average Size (FBM-AS)*

The Free Block Metric – Average Size (FBM−AS) is the average size of free blocks in the free list during the simulation, see figure 3. The metric is expressed in Bytes.

The metric highlights the fact that a heap with larger free blocks, on average, is more compact, and therefore, less fragmented.

The metric is expressed in Bytes to compare the average value with the largest request during the simulation of the scenario. An indication of a potential problem is shown when the average is below the amount of the largest block requested since, in those cases, the request may not be satisfied.

However, to compare this fragmentation value with other heap sizes, the metric can be normalized to the heap size and a value between 0 and 1 can be returned. In case of 1 we have a free heap with no allocations and, in case of values close to 0 we have a heap that is very fragmented.

### C. *Internal Fragmentation (IF)*

Internal fragmentation measures the memory wasted (in Bytes) when a request is served by a larger free block. The memory wasted is internal to the allocated block, therefore, it is called internal fragmentation.

Unfortunately it is not always possible to allocate the exact size free block, as rounding up the requests, for example, contributes to the internal fragmentation. In some allocators, the free block is split and only the necessary chunk of memory is allocated; however, leaving small blocks in the memory contributes to the increase of external fragmentation, especially if the remaining chunk of free memory is less than the minimum block size allocable in the system.

As an example, segregated free lists do not contribute to the external fragmentation, but they do contribute to the internal fragmentation when a request is served by a larger chunk of free memory.

### D. *Cost Metric (CM)*

Cost plays an important role when a product must be sold. In embedded systems resources are limited and the usage of memory must be efficient.

The cost metric (CM) has been defined with the scope of quantifying the amount of memory needed to run significant scenarios and subsequently establishing the amount of memory to be placed in the handset. At the same time, the cost metric measures the efficiency of the memory usage of different memory management systems.

The cost metric can answer which one of the allocators is able to cope with limited memory and therefore, is able to more efficiently use the memory. Different dynamic memory management systems, with different algorithms, heap layouts and parameters, have different cost metric values to run the scenarios and we are interested in the *cheapest* ones. The Cost Metric is expressed in Bytes. The cost metric is extracted by using a program that shrinks the heap size to find the smallest heap size needed to run the selected scenario. The Cost Metric can be normalized using the total memory alllocated for the dynamic memory management. A high cost in this case will correspond a value close to 0 and for a cheap cost we will have a value close to 1.

### E. *Performances Metric (PM)*

A different approach is needed when measuring the performance speed in a simulation environment. The performance speed cannot be measured using the execution time.

In order to measure the speed of the different dynamic memory management systems, the metric for performance speed is measured as the number of scans needed to access the memory blocks.

The metrics described here measure performance using the number of scans in the data structure needed to allocate and deallocate a memory block. Whenever an application requests a free block, the dynamic memory management system must find in due time a large enough free block. At the same time, when a block is deallocated, it must be inserted in the right place in the list of the free blocks. For example, a size-ordered free list will require the insertion of a freed block in the right position in the list according to its size.

The allocators used in the experiment have implemented the free list using a linked list data structure. The performance speed is measured in this case as the number of scans in the linked list. A different data structure can be used without invalidating the metric.

The performance speed metrics have been used for memory allocations and deallocations to measure the average and the worst case value. The worst case value is important considering we are investigating real-time systems. In the allocation scenario are included all the scans needed to allocate the request, which include the number of scans needed to insert a remaining free chunk in the free list in case the free block is split.

The performance metrics introduced have the benefit of being CPU independent. The advantage is also that the values are not influenced by the trace instrumentation technique used for extracting the data. Including trace instrumentation in the source code adds performance overhead that is not easy to calculate and in some cases can harm and corrupt the performance measurements [20], even though attempts have been made to limit the problem [21].

## IV. The Case Study

The scope of the experiment was to evaluate and to improve the dynamic memory management system of Nokia handsets.

Scenario selection is the first step in the process and sets the focus of the analysis. Scenarios contain the execution of one or more features. A feature is a user visible functional requirement, e.g. FM radio. Evaluating the dynamic memory management system means finding significant scenarios in terms of memory usage. However, features with strict real-time requirements must be included and analyzed. Ramps, peaks and plateaus were the patters described by Wilson et al. [19] and different patterns of data allocations can be found executing the scenarios. The features list is the starting point for selecting significant scenarios. However, in the scenario selection process, interviews and brainstorming sessions with developers and architects are fundamental.

One handset with multimedia features was selected for the experiment. The products in the Nokia handsets portfolio include handsets with basic functionalities (i.e. phone call, SMS) and handsets with multimedia features (i.e. MMS, Video Recording, etc.). Since we wanted to check also the impact of multimedia features on the dynamic memory usage, the multimedia mobile phone selected was representative for the study.

At the end of the scenario selection phase, significant scenarios were selected using not only core set features but also optional multimedia features (since not all products have multimedia features). The number of scenarios depends on the analysis scope and the time available for the analysis. A right balance must be found. A limited number of scenarios means an analysis better focused; on the other hand, too a narrow scope prevents a comprehensive and sound analysis. Some of the key performance scenarios selected were:

- Sending and Receiving $5$ MMS of $100$ KB each
- Browsing the Web
- Downloading and playing a Java Midlet
- Phone Start-Up

By running the scenarios in the handset we extracted the data of allocations and deallocations and we stored them in text log files. A summary of the data characteristics extracted from running the scenarios from the handset are in table I.

Using data of memory allocations and deallocations we were able to analyze the data characteristics and requirements of the features selected in the scenarios. A simulation environment was then used to test and evaluate different allocation strategies. The simulation environment was designed to test different allocation strategies, parameters and data structure for the heap. In order to be able the include additional algorithms and data structures, the simulator followed the *Abstract Factory* and the *Builder* design patterns [22]. The simulator gave as output the metrics described in III. A good reference for the design of a simulator for dynamic memory management systems is [23].

We evaluated different heap layouts and algorithms varying configuration parameters such as memory size and configuration of the pool system. In this paper, as a demonstrative example, we present the assessment of the the best-fit algorithm with two heap layouts using as input the data of the scenario *Sending and Receiving* $5$*MMS of* $100$*KB each*.

A heap layout in the experiment was either made of a single contiguous block of memory or made with a preallocated set of free lists plus a contiguous heap layout, as in figure 4. Segregated free lists (or the pool) are preallocated sets of bins with fixed-size chunks of memory [24], and the configuration of the pool is a parameter of the simulation environment. Small requests were served by the pool system. The bins were size-ordered and the requests were served in a FIFO (First In First Out) fashion.

The last phase in process is the analysis of the results obtained from the simulations. The best dynamic memory management system does not exist as an absolute concept. We must be able to evaluate the dynamic memory management according to the scenarios selected. A dynamic memory management might be able to serve large block of allocations efficiently but small block allocations are more important if the system is characterized by them predominantly. The process is iterative and the previous phases can be reiterated more times if needed.

In table II are summarized the metrics extracted for the scenario analyzed. The heap layouts in the simulation used the same amount of memory, with the difference that the heap plus the pool configuration had the memory split in two data structures.

In table III are presented the performance speed figures following the metrics described in section III-E.

## V. Results

The scope of the experiment was the evaluation of the dynamic memory management in the context of embedded real-time systems (Nokia handsets). In this experiment we have simulated two heap layouts with the best-fit algorithm and presented the metrics extracted for one scenario, see tables II and III.

By analyzing the metrics, we concluded that the single heap layout presented a more fragmented heap with many small blocks, as the Free Block Metric-Average Size showed. The result was not a surprise: in the other heap layout simulated, the heap with segregated free lists, the small blocks are not counted since segregated free lists do not contribute to external fragmentation. However, in addition to presenting a higher external fragmentation, the single heap layout presented a higher internal fragmentation.

Conversely, the Smallest-Biggest Block metric (SBBM) is the reference point to decide if the allocator is good enough. The SBBM is the value to be compared with the biggest block size value allocated executing the scenario, see the table I. In the experiment this value was higher than the value of the biggest block size allocated during the simulation of the scenarios, which showed that the problem of *not enough memory* was not an issue.

The higher normalized value (cheaper) of the Cost Metric by the best-fit with a single heap shown that this layout was able to better optimize the usage of memory and run the scenario with a smaller memory footprint. The Cost Metric was extracted by resizing only the heap and, in the heap plus

| Scenarios | Number of Obj alloc. in the simulation | Average Obj Size (Bytes) | Biggest Block Size (Bytes) |
|---|---|---|---|
| Sending and Receiving 5 MMS of 100 KB each | 781563 | 92.67 | 96793 |
| Downloading and playing a Java Midlet | 119932 | 83.87 | 46363 |
| Browsing the Web | 201744 | 70.90 | 38859 |
| Phone Start-Up | 26811 | 50.02 | 18400 |

TABLE I
DATA CHARACTERISTICS OF THE SCENARIOS

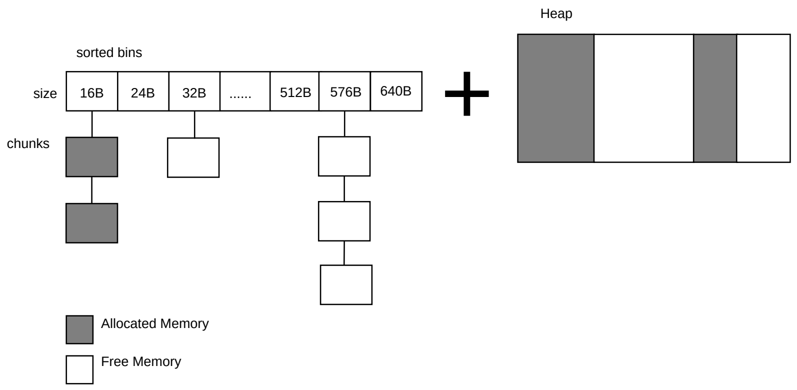

Fig. 4. Pool System plus Heap

| Sending and Receiving 5 MMS of 100KB | Free Block Metric− Average Size (Bytes) | Smallest−Biggest Block Metric (Bytes) | Cost Metric (normalized) | Internal Fragmentation Average Value (Bytes) |
|---|---|---|---|---|
| $heap\_bf$ | 16489.97 | 669876 | 0.51 | 16511.55 |
| $heap\_pool\_bf$ | 85265.69 | 327264 | 0.22 | 11299.17 |

TABLE II
METRICS AND STATISTICS FOR SENDING AND RECEIVING 5 MMS SCENARIO

| 5 MMS of 100KB | Performance Metric− Worst Case Allocation | Performance Metric− Average Scans Allocation | Performance Metric− Worst Case Deallocation | Performance Metric− Average Scans Deallocation |
|---|---|---|---|---|
| $heap\_bf$ | 192 | 69.12 | 198 | 56.31 |
| $heap\_pool\_bf$ | 30 | 9.89 | 16 | 6.20 |

TABLE III
PERFORMANCE SPEED, SENDING AND RECEIVING 5 MMS OF 100KB SCENARIO

pool data structure, a large part of the memory was allocated by the pool. The metric gave insights on the boundary between the heap and the pool system. By reducing the pool size, the Cost will decrease and therefore, the scenario will be able to run in a smaller memory footprint. On the otherhand, the tradeoffs with the other metrics described must be considered.

Looking at the performance speed of the two proposed dynamic memory managements in table III, we can see that the best-fit with a single heap layout is extremely slow if compared with the heap with segregated free lists. The worst-case allocation and deallocation performance was much higher in the best-fit single heap layout, and the worst-case value is important in real-time systems. The free lists in the segregated free lists were managed using a FIFO strategy, and since almost $90\%$ of the requests were small blocks, the advantages of the pool system were clearly demonstrated.

The experiment showed that a different heap layout has an important role in dynamic memory management. A heap plus segregated free lists was revealed to be a better solution, especially in situations where there are small patterns of allocations. However, multimedia features demand larger contiguous blocks and the single heap layout showed the best values for the metric SBBM. The increase in the progression of the largest block size allocated by multimedia features can be seen in table I. Therefore, in the future, with the introduction of new multimedia features in the handsets, the considerations made may be not valid anymore and a different dynamic memory management system could be adviced.

The conclusions above may seems obvious, however, only the assessment and the metrics extracted provided the evidence for the considerations made. The quantitative evaluation of the scenario illustrated the patterns of allocations of the system investigated and the metrics gave an understanding of the performance and the fragmentation.

## VI. Discussion

In this paper we presented an approach to the investigation of the problem of evaluating dynamic memory management systems. The scope of the research was not to show the best algorithms for dynamic memory management but to introduce a process of evaluating different strategies for real-time embedded systems. WE have presented a method and defined its steps; in addition, we have proposed metrics to evaluate different approaches of measuring the efficiency in terms of memory usage and speed performance.

The analysis is based on the significance of the scenarios. Therefore, significant scenarios for memory optimization must be defined well. The data are extracted from scenarios run in the targeted system, and the data is used as input for the simulation and subsequently for the analysis. Therefore, it is the job and primary responsibility of the performance engineer to define the scenarios that determine the scope of the analysis. The simulation planning phase ultimately refines the scope of the analysis and defines the objectives. The metrics must be representative and highlight the properties important to the scope of the analysis. The metrics that are defined here, help to evaluate the characteristics of dynamic memory management systems for memory efficiency and speed performance in the context of real-time embedded systems. No compaction of the memory is done in the system studied (and no garbage collector system is used) and the metrics reflect the issue: the best strategies are the ones that preserve a more compact memory with large free blocks; therefore, a less fragmented memory and the *Free Block Metric* express clearly this scope.

The cost metric can be used to understand and quantify the amount of memory to be allocated to dynamic memory in a mobile terminal and to evaluate the dynamic memory management system efficiency. We wanted to be able to determine and estimate the amount of dynamic memory to be placed in the handset. At the same time, memory efficiency is also evaluated by this metric: a dynamic memory management system that is able to work with less memory is more efficient and able to maintain a low level of memory fragmentation.

The metrics for memory fragmentation in the experiment are expressed in bytes; however, they can be normalized according to the size of the heap and then be used to express fragmentation without regard of the heap size used. Bytes were used because the *Smallest-Biggest Block Metric* and the *Free Block Metric* apply only if they can be compared with the largest block requested during the simulation. If the greatest memory requested in the simulated scenario is larger than the metric values, that request could lead to a failure during the system life-time.

Trace instrumentation was used and the data analyzed represented the run-time properties of the embedded real-time system studied. From the analysis emerged the importance of the patterns of data allocations and deallocations of the applications and the consequences to the memory management system. If many small blocks are allocated, a segregated free lists (pooling system) system is advised. On the other side, the allocation of dynamic memory to a segregate free lists data structure may negatively affect large blocks allocations which are served by a single heap data structure. A balance in the configuration of these data structures is needed.

The allocator strategies are important but the heap layout and the configuration of the parameters, such as the definition of the pool system, also plays an important role.

However, the pooling system presents some challenges as well. Configuring the pooling system in order to avoid overflows is not an easy task. The configuration of the pool should optimize memory usage so no overflow happens, and at the same time should have the minimum number of pre-allocated blocks so free blocks are not left unused. The right number of chunks for each bin unfortunately does not exist *a priori* but defining the significant and challenging scenarios once again is very important here. The fragmentation of the heap can then be checked using the simulation environment with different configuration parameters.

A trade-off between memory usage efficiency and performance speed must be evaluated by the performance analyst. The dynamic memory management system must perform fast enough for the correct functioning of the system and must use memory efficiently enough to be able to handle all the possible scenarios a user will encounter.

## VII. Conclusion and future work

In this paper we have presented a case study of the evaluation and the analysis of dynamic memory for real-time embedded systems. In addition we have presented a set of metrics to evaluated different dynamic memory management systems.

The approach is scenario-based and scenarios are used to specify the focus of the analysis. In the case study presented, real traces of memory allocations from one Nokia handset were used in the simulation environment. The simulation environment was built for the experiment and it was used to evaluate two dynamic memory management systems. In the analysis phase, the different alternatives were compared and evaluated.

Our aim in this paper is not to present a definitive study on the algorithms and data structures to be used but to present the method, the rationales and the tools for assessing embedded real-time systems in practice.

In future work we aim to extend the analysis of dynamic memory management by studying in depth the improvements to be made in every single phase of the method. For example, potential enhancements and challenges exist in the identification of key performance scenarios when embedded systems are part of a software product family. At the same time, different heap layouts can be introduced into the simulation and their performance can be measured against previous layouts. Moreover, configuration parameters are important and we are studying the use of genetic algorithms [25], [26], [27] to configure the pool system [28]. Dynamic memory management is a complex problem and must be tackled by considering all the aspects and not just analyzing the algorithms used. This approach establishes a framework and a starting point for future investigations for such an interesting and important issue in computer science as dynamic memory management.

## References

[1] Donald E. Knuth. *The Art of Computer Programming, Vol 1, Third Edition*. Addison Wesley, 1997.

[2] B. Randell and C. J. Kuehner. Dynamic storage allocation systems. *Communications of the ACM*, 11(5):297 – 306, May 1968.

[3] Paul W. Jr. Purdom and Stephen M. Stigler. Statistical properties of the buddy system. *Journal of the ACM (JACM)*, 17(4):683 – 697, October 1970.

[4] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623 – 624, October 1965.

[5] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM (JACM)*, 18(3):416–423, July 1971.

[6] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for java. *Software: Practice and Experience*, 30(3):199 – 232, March 2000.

[7] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.

[8] Emery D. Berger and M. Hertz. Quantifying the performance of garbage collection vs. explicit memory management. *OOPSLA*, http://www.cs.umass.edu/ẽmery/pubs/gcvsmalloc.pdf 2005.

[9] Paul Wilson. Uniprocessor garbage collection techniques. *International Workshop on Memory Management, Lecture Notes in Computer Science*, (637):1–42, November 1992.

[10] Doug Lea. A memory allocator. *http://gee.cs.oswego.edu/dl/html/malloc.html*.

[11] Jeffrey Ritcher. *Advanced Windows, 3rd edition*. Microsoft Press, 1997.

[12] Apache Foundation. Apache web server: http://www.apache.org.

[13] SGI. Standard template library: http://www.sgi.com/tech/stl/allocators.html.

[14] Free Software Foundation. Gcc home page: http://gcc.gnu.org/.

[15] Emery D. Berger and Kathryn S. Zorn, Benjamin G.and McKinley. Reconsidering custom memory allocation. *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, November 2002.

[16] Microsoft Corporation. *Microsoft Windows NT 4.0 Online Documentation*. Microsoft Corporation, Redmon, Washington, 1997.

[17] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? *Proceedings of the 1st international symposium on Memory Management*, 34(3), October 1998.

[18] John E. Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM*, 18(8):433 – 440, August 1975.

[19] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. *Proc. Int. Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

[20] D. Steward. Measuring execution time and real-time performance. *Embedded Systems Conference (ESC)*, April 2001.

[21] E. Metz and R. Lencevicius. Efficient instrumentation for performance profiling. *Proceedings of the ICSE Workshop on Dynamic Analysis, (WODA)*, May 3-11 2003.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[23] Chia-Tien Dan Lo, Witawas Srisa-an, and J. Morris Chang. The design and analysis of a quantitative simulator for dynamic memory management. *The Journal of Systems and Software*, 72(3):443–453, August 2004.

[24] James Noble and Charles Weir. *Small Memory Software: Patterns for system with limited memory*. Addison Wesley, 2001.

[25] F. Feitelson, Dror and Michael Naaman. Self-tuning systems. *IEEE Software*, 6(12):52–60, March-April 1999.

[26] H. Holland, J. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[27] E. Goldberg, D. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.

[28] Christian Del Rosso. Reducing internal fragmentation in segregated free lists using genetic algorithms. *Proceedings of the 2nd International ACM Workshop on Interdisciplinary Software Engineering Research*, pages 143 – 150, 2006.