Reducing Memory Fragmentation with Performance-optimized Dynamic Memory Allocators in Network Applications

Stylianos Mamagkakis[1], Christos Baloukas[1], David Atienza[2], Francky Catthoor[3], Dimitrios Soudris[1], José Manuel Mendias[2] and Antonios Thanailakis[1]

[1] VLSI Design Center-Democritus University of Thrace, 67100 Xanthi, Greece
{smamagka, cmpalouk, dsoudris, thanail}@ee.duth.gr
[2] DACYA - Univ. Complutense de Madrid, 28040 Madrid, Spain
{datienza, mendias}@dacya.ucm.es
[3] IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium
Also professor at K.U.Leuven, Belgium.
catthoor@imec.be

Abstract. The needs for run-time data storage in modern wired and wireless network applications are increasing. Additionally, the nature of these applications is very dynamic, resulting in heavy reliance to dynamic memory allocation. The most significant problem in dynamic memory allocation is fragmentation, which can cause the system to run out of memory and crash, if it is left unchecked. The available dynamic memory allocation solutions are provided by the real time Operating Systems used in embedded or general-purpose systems. These state-of-the-art dynamic memory allocators are designed to satisfy the run-time memory requests of a wide range of applications. Contrary to most applications, network applications need to allocate too many different memory sizes (e.g. hundreds different sizes for packets) and have an extremely dynamic allocation and de-allocation behavior (e.g. unpredictable web-browsing activity). Therefore, the performance and the de-fragmentation efficiency of these allocators is limited. In this paper, we analyze all the important issues of fragmentation and the ways to reduce it in network applications, while keeping the performance of the dynamic memory allocator unaffected or even improving it. We propose highly customized dynamic memory allocators, which can be configured for specific network needs. We assess the effectiveness of the proposed approach in two representative real-life case studies of wired and wireless network applications. Finally, we show very significant reduction in memory fragmentation and increase in performance compared to state-of-the-art dynamic memory allocators utilized by real-time Operating Systems.

1 Introduction

In the last years networks have become ubiquitous. Modern portable devices are expected to access the internet (e.g. 3G mobile phones) and communicate with each other wirelessly (e.g. PDAs with 802.11b/g) or with a wired connection (e.g. Ethernet). In order to provide the desired Quality of Experience to the user, these

systems have to respond to the dynamic changes of the environment (i.e. network traffic) and the actions of the user as fast as possible. Additionally, they need to provide the necessary memory space for the network applications dynamically at run-time. Therefore, they have to rely on dynamic memory allocation mechanisms to satisfy their run-time data storage needs. Inefficient dynamic memory (DM from now on) allocation support leads to decreased system performance and increased cost in memory footprint due to fragmentation [1].

The standard DM allocation solutions for the applications inside the Terminals, Routers or Access Points are activated with the standardized malloc/free functions in C and the new/delete operators in C++. Support for them is provided by Real Time Operating Systems (e.g. uClinux [8]). These O.S. based DM allocators are designed for a variety of applications and thus can not address the specific memory allocation needs of network applications. This results in mediocre performance and increased fragmentation. Therefore, custom DM allocators are needed [7, 12] to achieve better results. Note that they are still realized in the middleware and usually not in the hardware. In our case we propose never to use hardware but instead use only a library (system layer) just on top of the (RT)OS in the middleware.

In this paper, we propose a systematic approach to reduce memory fragmentation (up to 97%) and increase performance (up to 97%), by customizing a DM allocator to be used especially for the network application domain. The major contribution of our work is that we explore exhaustively all the available combinations of de-fragmentation techniques and explain how our custom DM allocator can decrease fragmentation and improve performance at the same time in network applications. The remainder of the paper is organized as follows. In Sect. 2, we describe some related work. In Sect. 3, we analyze fragmentation. In Sect. 4, we show the de-fragmentation techniques and their trade-offs. In Sect. 5, we describe our exploration and explain the effect of each de-fragmentation technique in the network application domain. In Sect. 6 we present the simulation results of our case studies. Finally, in Sect. 7 we draw our conclusions.

2 Related Work

Currently, there are many O.S. based, general-purpose DM allocators available. Successful examples include the Lea allocator in Linux based systems [5], the Buddy allocator for Unix based systems [5] and variations of the Kingsley allocator in Windows XP [11] and FreeBSD based systems. Their embedded O.S. counterparts include the DM allocators of Symbian, Enea OSE [9], uClinux [8] and Windows CE [10]. Other standardized DM allocation solutions are evaluated in [6] for a wide range of applications (without evaluating performance). In contrast to these 'off-the-shelf' DM allocation solutions, our approach provides highly customized DM allocators, fine tuned to the networking applications for both low memory fragmentation and high performance.

Also, in [12], the abstraction level of customizable memory allocators has been extended to C++. Additionally, the authors of [7] propose an infrastructure of C++ layers that can be used to improve performance of general-purpose

allocators. Finally, work has been done to propose several garbage collection algorithms with relatively limited performance overhead [13]. Contrary to these frameworks, which are limited in flexibility, our approach is systematic and is linked with our tools [2], which automate the process of custom DM allocator construction. This enables us to explore and validate the efficiency of our customized DM allocators, combining both memory de-fragmentation and performance metrics.

3 Memory Fragmentation

When the application requests a memory block from the DM allocator, which is smaller than the memory blocks available to the allocator, then a bigger block is selected from the memory pool and allocated. This results in wasted memory space inside the allocated memory block. This is called internal fragmentation, which is common in requests of small memory blocks [5]. When the application requests a memory block from the DM allocator, which is bigger than the memory blocks available to the allocator, then these smaller memory blocks are not selected for the allocation (because they are not contiguous) and become unused 'holes' in memory. These 'holes' among the used blocks in the memory pool are called external fragmentation.

We measure the level of both internal and external fragmentation (we use the same cost function with [6]). Thus, we express fragmentation in terms of percentages over and above the amount of live data, (i.e. increase in memory usage), not the percentage of actual memory usage that is due to fragmentation. Therefore, we measure the maximum amount of memory requested by the application relative to the maximum amount of memory used by the DM allocator:

$$Fragmentation = \frac{Memory_{alloc.}}{Memory_{req.}} - 1$$

$$Memory_{alloc.} = Memory_{req.} + Memory_{Int.Fragm.} + Memory_{Ext.Fragm.}$$

4 Memory De-fragmentation Techniques and Trade-offs

We are going to analyze the de-fragmentation techniques and their trade-offs. All of the techniques are well known [5] but their trade-offs (when used in conjunction) have never been evaluated up to now:

1.-The most common technique to prevent internal memory fragmentation is the use of *freelists*. The *freelists* are lists (i.e. double or single linked lists) of memory blocks, which were no longer needed by the application and, thus, they were freed by the DM allocator. This technique can reduce internal fragmentation significantly and improve performance in most cases. The trade-off is that it increases external fragmentation, because the freed blocks are not returned in the main memory pool, where they can be coalesced with a neighboring free block to produce a bigger contiguous memory space.

2.-Another technique to prevent internal memory fragmentation is the use of specific fit policies. The two most popular fit policies are the *first fit policy*

and the *best fit policy*. On the one hand, the *first fit policy* allocates the first memory block that it finds that is bigger than the requested block. On the other hand, the *best fit policy* searches a part (or even 100%) of the memory pool in order to find the memory block closest to the size of the requested block. Therefore, there will be the least memory overhead per block and, thus, the least internal fragmentation. The trade-off is that the performance of the DM allocator decreases, while it spends more time trying to find the best fit for the requested block.

3.-An additional technique to decrease internal fragmentation is the use of the *splitting mechanism*. When the DM allocator finds a block bigger than the requested block, then it can split it in two. The block can be split precisely to fit the request and, thus, produce zero internal fragmentation. The trade-off of this mechanism is that it reduces performance considerably. The mechanism itself needs a lot of time perform the splitting, plus it generates one more block inside the pool per split.

4.-Finally, a technique to decrease external fragmentation is the use of the *coalescing mechanism*. When the DM allocator frees a block, which has an adjacent memory address with another free memory block, then it can coalesce them to produce a single bigger block. In this way, external memory fragmentation can be reduced significantly. A positive by-product of the *coalescing mechanism* is that it results in one less block inside the pool per coalesce. This in turn reduces significantly the time needed to traverse all the blocks inside the pool to find a best or first fit. On the other hand, the trade-off of this mechanism is that it reduces some performance, because the mechanism itself needs some time to perform the coalescing.

It is obvious that these four different de-fragmentation techniques have contradicting effects on performance, internal and external fragmentation (e.g. an increase of usage of the *splitting mechanism* decreases internal fragmentation but also decreases performance). To make things even more complicated it appears that the efficiency of the techniques is interdependent (e.g. the performance of the *best fit policy* decreases when the usage of the *splitting mechanism* increases). So a Pareto trade-off exploration is necessary. In order to evaluate which techniques should be used to decrease fragmentation and how much they should be applied, we have explored exhaustively all the available combinations of de-fragmentation techniques in various levels of usage (ranging from full usage to no usage of the technique at all).

5 Customization of DM Allocators for Network Applications

For the purposes of the exhaustive exploration of the different de-fragmentation techniques we have used our powerful profiling tool (described in more detail in [2]). Our tool automates the process of building, implementing, simulating and profiling different customized DM allocators. Every one of these customized DM allocators implements a different combination of de-fragmentation techniques with a different combination of usage level for each technique. About 10 levels of

usage have been used for each de-fragmentation technique. The total exploration effort took 45 days using 2 Pentium IV workstations. On average, there have been explored about 10.000 different customized DM allocator implementations for each one of two different networking applications: DRR scheduling and buffering in Easyport. Finally, 3 to 7 real network traffic trace inputs (of wired and wireless networks) have been used for each application to make sure that our exploration strategy is valid for a wide range of dynamic behavior scenarios.

In Fig. 1, a custom DM allocation exploration example for the Easyport buffering application can be seen (a network traffic trace of various real ftp sessions was used as input). Each dot in the figure is the simulation results for performance and memory footprint allocated by one out of the 10.000 explored custom DM allocators. The results were heavily pruned and (out of the 10.000 custom DM allocator implementations) only a handful with the best performance and lowest fragmentation were selected (as seen in the upper right corner of Fig. 1). The same procedure has been used for the other applications and for each one of the available inputs (i.e. network traffic traces).

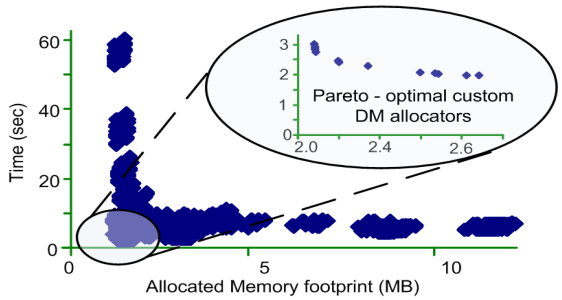

Fig. 1. Custom DM allocation exploration example for the Easyport buffering application and pareto-optimal DM allocators

Our simulations show that the limited list of resulting 'Pareto-optimal' custom DM allocators share some common characteristics, which favor particular de-fragmentation techniques at certain levels of usage:

1.-Contrary to most application domains (where about 6 different memory sizes amount for more than 90% of the total requested memory sizes [6]), in networking applications just 2 memory sizes amount for 30-70% of the total requested memory sizes (an example of this bimodal distribution can be seen in the histograms of Fig. 3). These 2 object sizes are around the size of the *Acknowledgement* (or ACK) packet and the *Maximum Transmission Unit* (or MTU) packet of each network [3]. The rest of the requested memory sizes are evenly distributed between these 2 extreme sizes. Our exploration results show that custom DM allocators, with just 2 freelists of these 2 extreme memory sizes, managed to reduce considerably internal fragmentation and improve performance, without increasing much the external fragmentation. All five of the

O.S. based DM allocators, which use from 6-64 different freelists, manage to do the same, but with a very high cost in external fragmentation.

2.-Contrary to most application domains (where memory usage comes in the form of very thin spikes and 10% of the memory sizes are freed back to the main memory heap or pool [5] [6]), in networking applications the memory usage form varies greatly [3] (in the upper 3 traces of Fig. 2 we can see thin and fat spikes, in the lower left trace of Fig. 2 we can see plateaus and in the lower right trace of Fig. 2 we can see a ramp). Additionally, about 30-70% of the memory sizes are returned to the main memory pool. This means that blocks are not always freed fast (this is the case of thin spike usage forms only) and that the main memory pool accommodates a huge number of memory blocks. It also means that the *best fit policy* used in all the O.S. based DM allocators (except Windows XP and CE) is extremely slow because it has to traverse too many blocks in order to find a good fit. Our exploration results show that custom DM allocators, which use *first fit policy* in combination with full usage of the *splitting mechanism* and the *coalescing mechanism*, increase performance dramatically and suffer only minimal internal fragmentation overhead.

3.-Contrary to most application domains (where about 38 different memory sizes constitute 99% of the total requested memory sizes [6]), in networking applications 30-70% of the total requested memory sizes are attributed to 700-1500 different memory sizes (an example of this fact can be seen in Fig. 3). This produces exceptionally high values of internal fragmentation, which is different from what is observed in other application domains. All the O.S. based DM allocators (except Linux) have a very low usage level of the *splitting mechanism* and therefore suffer massively from internal fragmentation. Actually, our exploration showed that this is the major contributor to fragmentation generally in network applications. Our exploration results show us that the only way to really decrease fragmentation is with the full use of the *splitting mechanism*.

4.-Finally, a common characteristic shared among the networking and the other application domains, is that objects allocated at the same time tend to die and get de-allocated at the same time. This temporal locality of the allocated objects is something common in both wired and wireless networks. The reason is that the traffic structure is imposed implicitly by the tasks initiated by Internet users at the application layer (e.g. a file or a Web page download). Therefore, allocated objects are not independent and isolated entities; rather they are part of a higher-layer logical flow of information [3]. This temporal locality can easily be converted to spatial locality of the memory freed, if we pursue high usage levels of the *coalescing mechanism*, thus reducing external fragmentation. All the O.S. based DM allocators (except Linux) have an extremely low usage level of the *coalescing mechanism* and thus can not take advantage of the locality effect. Our exploration results have shown, that with full usage of the *coalescing mechanism*, external fragmentation in networking applications can be eradicated.

These favorable common characteristics are a combination of just two *freelists*, *first fit policy*, full usage of the *splitting mechanism* and full usage of the *coalescing mechanism*. Therefore, this is the custom DM allocator that we propose to use for network applications.

6 Case Studies and Simulation Results

We have applied the proposed custom DM allocator to two real case studies:

The first case study, is the Deficit Round Robin (or DRR) [14] application, which is a scheduling algorithm implemented in many routers and WLAN Access Points today [15]. In the DRR algorithm, the scheduler visits each internal non-empty queue, increments the variable deficit by the value *Quantum* (e.g. 9 Kbytes are used in most Cisco Routers) and determines the number of bytes in the packet at the head of the queue. If the variable deficit is less than the size of the packet at the head of the queue (it does not have enough credits), then the scheduler moves on to service the next queue. If the size of the packet at the head of the queue is less than or equal to the variable deficit, then the variable deficit is reduced by the number of bytes in the packet and the packet is transmitted on the output port. The scheduler continues this process, starting from the first queue each time a packet is transmitted. If a queue has no more packets it is destroyed. The arriving packets are queued to the appropriate node and if no such exists then it is created.

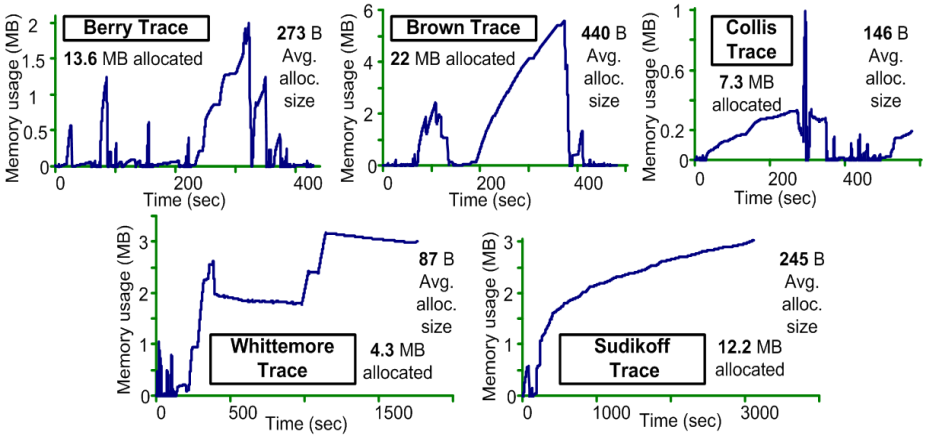

Fig. 2. Real memory usage of the DRR application for wireless traffic traces of different buildings [4] (50.000 packets)

It is important to stress that the simulation results of the DRR application were taken for 5 real wireless traffic traces. These traces represent the traffic of 5 different buildings in Dartmouth University Campus [4]. As noted in [6], a randomly generated trace is not valid for predicting how well a particular DM allocator will perform on a real program. The reason is that for different inputs there will be different dynamic allocation behaviors and allocation sizes(as shown in Fig. 2 and 3 respectively). The effect of the different dynamic behaviors can be seen in the variation of the simulation results (as shown in Table 2).

After an exhaustive exploration of all the custom DM allocators (as explained in the previous section), we use an instantiation of the proposed parameterized

setup that is described at the end of section 5. Namely, we use 2 *freelists* for memory blocks of 16 Bytes and memory blocks of 1476 Bytes and we fully apply the *coalescing mechanism*, the *splitting mechanisms* and the *first fit policy*. Note that although in the packet traces the ACK packet has zero size and the MTU packet has a size of 1460 Bytes, 16 Bytes more are allocated per objects to store some application-specific data (e.g. like *Quantum*). From our exhaustive exploration we have concluded that the aforementioned custom DM allocator is the most balanced, giving both low fragmentation and good performance (other custom DM allocators give only good performance or only low fragmentation).

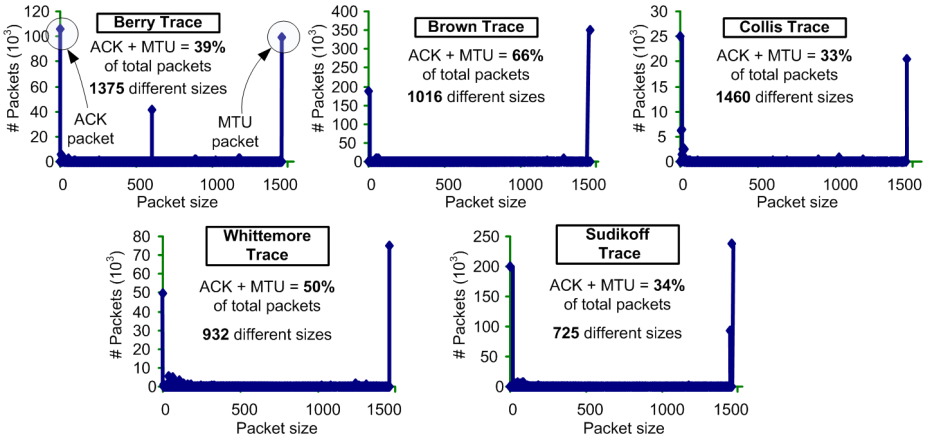

Fig. 3. Histograms of memory allocation requests of the DRR application for wireless traffic traces of different buildings [4]

Table 1. Simulation results for the DRR scheduling algorithm running for 50.000 packets per trace (lower fragmentation and execution-time is better)

DM Allocators	Fragmentation						Performance (execution-time sec.)					
	Avrg.	Ber.	Br.	Col.	Sud.	Whit.	**Avrg.**	Ber.	Br.	Col.	Sud.	Whit.
Windows CE	**83%**	70%	13%	59%	251%	20%	**1.78**	0.36	0.78	0.34	5.17	2.27
Windows XP	**142%**	169%	21%	183%	256%	80%	**1.69**	0.28	0.58	0.31	5.25	2.03
Linux	**66%**	35%	8%	59%	206%	23%	**2.19**	0.33	0.79	0.50	6.58	2.74
Enea OSE	**93%**	62%	8%	100%	212%	86%	**7.91**	8.40	10.88	8.55	8.04	3.67
uClinux	**152%**	93%	49%	153%	350%	117%	**2.40**	0.13	0.51	0.33	6.68	4.34
Avrg. Alloc.	**107%**	**86%**	**20%**	**111%**	**255%**	**65%**	**3.19**	**1.90**	**2.71**	**2.01**	**6.34**	**3.01**
Proposed Alloc.	**55%**	20%	1%	35%	183%	35%	**1.62**	0.17	0.46	0.24	5.13	2.09

Then we simulate and compare our customized DM allocator with O.S. based DM allocators for 5 different network traces (note that the very bad fragmentation and performance results of all the allocators for the Sudikoff trace are

attributed to the ramp form of its memory usage, i.e. too much network traffic results in DM allocation bottleneck). We observe that for the average of all the traces our custom DM allocator is both faster and has less fragmentation than any O.S. based DM allocator. In fact, it can achieve memory fragmentation reductions up to 97.82% (48.39% on average) and execution time reductions up to 97.20% (49.22% on average).

The second case study presented is the Easyport wireless network application produced by Infineon [16]. Easyport features packet and ATM cell processing functionality for data and voice/data Integrated Access Devices (IADs), enterprise gateways, access routers, and Voice over IP (VoIP) gateways. Easyport allocates dynamically the packets it receives from the Ethernet channels in a memory before it forwards them in a FIFO way. To run simulations of Easyport, we used 3 typical packet traffic traces provided by Infineon (mainly ftp sessions).

Table 2. Simulation results for the Easyport buffering algorithm running for 4.200 packets per trace (lower fragmentation and execution-time is better)

DM	Fragmentation				Performance (execution-time sec.)			
Allocators	**Avrg.**	Trace 1	Trace 2	Trace 3	**Avrg.**	Trace 1	Trace 2	Trace 3
Windows CE	**46%**	30%	75%	34%	**0.51**	0.62	0.33	0.60
Windows XP	**49%**	33%	78%	37%	**0.49**	0.59	0.31	0.59
Linux	**40%**	29%	62%	28%	**0.59**	0.69	0.37	0.71
Enea OSE	**46%**	30%	70%	36%	**1.20**	1.42	0.81	1.39
uClinux	**60%**	40%	97%	42%	**0.87**	1.02	0.62	0.97
Avrg. Alloc.	**48%**	**32%**	**77%**	**35%**	**0.73**	**0.86**	**0.48**	**0.85**
Proposed Alloc.	**37%**	20%	61%	30%	**0.47**	0.52	0.32	0.59

After an exhaustive exploration of the all the custom DM allocators (as explained in the previous section), we again use an instantiation of the proposed parameterized setup that is described at the end of section 5. Namely, 2 *freelists* for memory blocks of 66 Bytes and memory blocks of 1514 Bytes and we fully apply the *coalescing mechanism*, the *splitting mechanisms* and the *first fit policy*. Again this specific custom DM allocator is the most balanced. We observe that for the average of all the traces our custom DM allocator is both faster and has less fragmentation than any O.S. based DM allocator. In fact, it can achieve memory fragmentation reductions up to 50.16% (23.0% on average) and execution time reductions up to 63.39% (35.62% on average).

7 Conclusions

Dynamism is an important aspect of wired and wireless network applications. Therefore, the correct choice of a Dynamic Memory Allocation subsystem becomes of great importance. Within this context, memory fragmentation must be minimized without a performance reduction. In this paper we have presented a novel approach to explore exhaustively the combinations of the de-fragmentation

techniques in custom DM allocator implementations. The results achieved with the use of our approach in real wired and wireless network applications show that our customized DM allocator solution can reduce memory fragmentation up to 97% and improve performance up to 97% compared to state-of-the-art, O.S. based DM allocators.

8 Acknowledgements

This work is partially supported by the European founded program AMDREL IST-2001-34379 and the Spanish Government Research Grant TIC2002/0750. We want to thank Matthias Wohrle (Advanced Systems and Circuits group, Infineon Technologies, Munich, Germany) and Arnout Vandecappelle (IMEC, DESICS group, Leuven, Belgium) for their help and support in the simulation of the Easyport application.

References

1. D. Atienza, S. Mamagkakis, F. Catthoor, et al. Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications. In *Proc. of IEEE/ACM DATE 2004*.
2. D. Atienza, S. Mamagkakis, et al. Modular Construction and Power Modelling of Dyn. Mem. Managers for Embedded Systems. In *Proc. of LNCS PATMOS 2004*.
3. C. Williamson. A Tutorial on Internet Traffic Measurement In *Proc. of IEEE Internet Computing, Vol. 5, No. 6, 2001*
4. D. Kotz, et al. Analysis of a campus-wide wireless network. In *Dartmouth CS Technical Report TR2002-432*.
5. P. R. Wilson, et al. Dynamic storage allocation, a survey and critical review. In *Int. Workshop on Mem. Manag.*, UK, 1995.
6. M. Johnstone, et al. The Memory Fragmentation Problem: Solved? In *Proc. of Intnl. Symposium on Memory Management 1998*
7. E. D. Berger, et al. Composing high-performance memory allocators. In *Proc. of ACM SIGPLAN PLDI*, USA, 2001.
8. Dyn. Allocation in uClinux RTOS. `http://linuxdevices.com/articles/AT7777470166.html`
9. Dyn. Allocation in Enea OSE RTOS. `http://www.realtime-info.be/magazine/01q3/2001q3_p047.pdf`
10. Dyn. Allocation in MS Windows CE. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50conheaps.asp`
11. Dyn. Allocation in MS Windows XP. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngenlib/html/heap3.asp`
12. G. Attardi, et al. A customizable memory management framework for c++. *Software Practice and Experience*, 1998.
13. D. Bacon, et al. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Proc. of SIGPLAN 2003*
14. M. Shreedhar, et al. Efficient Fair Queuing using Deficit Round Robin. In *Proc. of SIGCOMM 1995*
15. M. Gerharz, et al. A Practical View on Quality-of-Service Support in Wireless Ad Hoc Networks. In *Proc. of IEEE ASWN 2003*
16. Infineon Easyport. `http://www.itc-electronics.com/CD/infineon%2010063/cd1/html/p_ov_33433_-9542.html`