Available online at www.sciencedirect.com

ScienceDirect

Computer Communications 29 (2006) 2612–2620

computer communications

www.elsevier.com/locate/comcom

Reducing memory fragmentation in network applications with dynamic memory allocators optimized for performance ☆

Stylianos Mamagkakis [a,*], Christos Baloukas [a], David Atienza [b], Francky Catthoor [c,1], Dimitrios Soudris [a], Antonios Thanailakis [a]

[a] VLSI Design and Testing Center, Democritus University of Thrace, 67100 Xanthi, Greece
[b] LSI/EPFL 1015-Lausanne, Switzerland and DACYA/UCM, 28040 Madrid, Spain
[c] IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium

Available online 20 March 2006

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 on 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 three 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.

Keywords: Memory fragmentation; Dynamic memory allocator; Network application; Performance optimized; Operating systems; Customization

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

☆ Part of this work has been submitted to WWIC 2005.
* Corresponding author. Tel.: +30 6976861165; fax: +30 2541079545.
E-mail addresses: smamagka@ee.duth.gr (S. Mamagkakis), cmpalouk@ee.duth.gr (C. Baloukas), david.atienza@epfl.ch, datienza@dacya.ucm.es (D. Atienza), catthoor@imec.be (F. Catthoor), dsoudris@ee.duth.gr (D. Soudris), thanail@ee.duth.gr (A. Thanailakis).
[1] Present address: Also Professor at ESAT/K.U. Leuven, Belgium.

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 [9]). These OS 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 [8,14] 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 98%) 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 Section 2, we describe some related work. In Section 3, we analyze fragmentation. In Section 4, we show the de-fragmentation techniques and their trade-off. In Section 5, we describe our exploration and explain the effect of each de-fragmentation technique in the network application domain. In Section 6, we present the simulation results of our case studies. Finally, in Section 7 we draw our conclusions.

2. Related work

Currently, there are many OS based, general-purpose DM allocators available. Successful examples include the Lea allocator in Linux based systems [6], the Buddy allocator for Unix based systems [6] and variations of the Kingsley allocator in Windows XP [13] and FreeBSD based systems. Their embedded OS counterparts include the DM allocators of Symbian [11], Enea OSE [10], uClinux [9] and Windows CE [12]. Other standardized DM allocation solutions are evaluated in [7] 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 [14], the abstraction level of customizable memory allocators has been extended to C++. Additionally, the authors of [8] 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 [15]. 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.

Finally, in contrast to our previous work [1] and [2], which focused on reducing the memory footprint and power consumption for multimedia and network applications, in this paper we focus on de-fragmentation and performance improvement. We also show that the latter are not improving in the same direction as the memory footprint or memory access cost functions. So they form important complementary objective functions for the optimization problem that we want to tackle. Additionally, in this paper we explore exhaustively all the combinations of de-fragmentation techniques for custom DM allocator implementations, instead of just giving general guidelines to achieve low memory footprint. Finally, we compare our proposed custom DM allocator with three more DM allocators of embedded real time OSs, on top of the two general purpose DM allocators that we use in [1] and [2].

3. Memory fragmentation

Memory fragmentation can be divided in internal and external fragmentation:

1. 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 (as shown in the upper part of Fig. 1). This results in wasted memory space inside the allocated memory block. This space is not used to store the application's data and can not be used for a

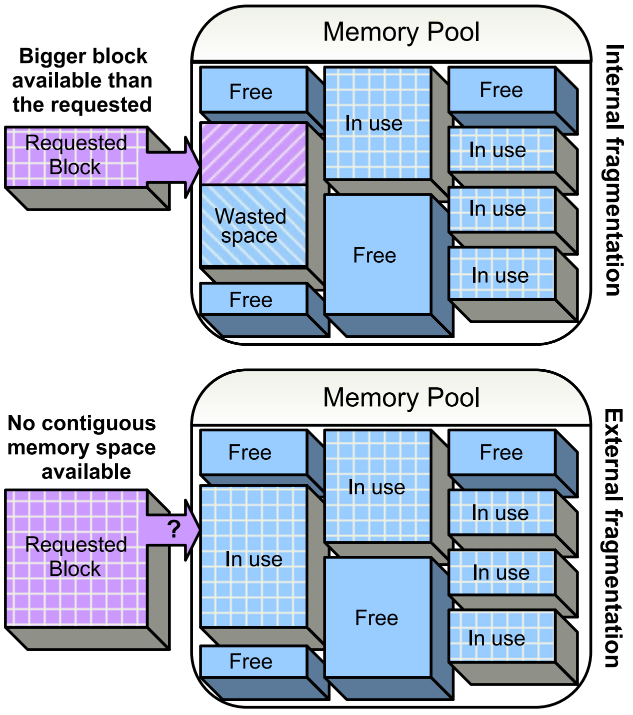


Fig. 1. Internal and external memory fragmentation.

future memory request. This is called internal fragmentation, which is common in requests of small memory blocks [6]. It can be prevented or reduced with the use of *freelists*, the *best fit policy* and the *splitting mechanism*, which will be analyzed in detail in the next section.

2. 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 (as shown in the lower part of Fig. 1). These 'holes' among the used blocks in the memory pool are called external fragmentation. If they become too small, then they can not satisfy any request and they remain unused during the whole execution time of the application. External fragmentation becomes more evident in requests of big memory blocks [6]. It can be reduced with the use of the *coalescing mechanism* and is increased as a by-product of the *freelists*, which will be explained in detail in the next section.

We measure the level of both internal and external fragmentation (we use the same cost function with [7]). 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:

$$\text{Fragmentation} = \frac{\text{Memory}_{\text{alloc.}}}{\text{Memory}_{\text{req.}}} - 1,$$

$$\text{Memory}_{\text{alloc.}} = \text{Memory}_{\text{req.}} + \text{Memory}_{\text{Int.Fragm.}} + \text{Memory}_{\text{Ext.Fragm.}}$$

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 [6] 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 (Table 1).

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

Table 1
Usage of de-fragmentation techniques in OS based dynamic memory allocators)

OS	De-fragmentation techniques of DM allocator
Windows CE	Windows CE use a memory heap for all the free blocks. The blocks within the heap are singly linked in a LIFO way. To decrease internal fragmentation, Windows CE use a first-fit algorithm. Free heap blocks are merged on every allocation or free cycle to decrease external fragmentation
Windows XP	Windows XP's dynamic memory allocation implementation uses 127 freelists of 8-byte aligned blocks ranging from 8 to 1024 bytes and a memory heap, which holds blocks greater than 1024 bytes in size, doubly linked FIFO list. To decrease internal fragmentation, Windows XP use a first-fit algorithm. It also provides full support for coalescing and splitting operations
Linux	In Lea 2.7.2, various levels of coalescing and splitting operations are supported (ranging from 0% for small blocks to 100% for bigger blocks). This is a best fit allocator, which can utilize up to 128 freelists according to the application memory block requests
Enea OSE	In Enea OSE, 8 freelists are used. This is a best fit allocator, which uses just these 8 freelists and no main memory heap. Coalescing and splitting operations are not supported
uClinux	In uClinux, the dynamic memory allocator uses a power-of-two allocator for allocations up to 4 kbyte. Then, for bigger blocks, it allocates memory rounded up to 4 kbyte. Two freelists are supported but no coalescing or splitting operations

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 three different networking applications: DRR scheduling, buffering in Easyport and URL-based context switching (presented in Section 6). Finally, 3–7 real network traffic trace inputs (of wired [5] and wireless [4] 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. 2, 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. 2). The same procedure has been used for the other applications and for each one of the available inputs (i.e., network traffic traces).

Our simulations show that the limited list of resulting 'optimal' custom DM allocators share some common characteristics, which favor particular de-fragmentation techniques (they are seen with bold letters in Table 2) at certain levels of usage. These common characteristics are a combination of two or three *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.

1. Contrary to most application domains (where about 6 different memory sizes amount for more than 90% of the total requested memory sizes [7]), 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. 4). 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 OS based DM allocators, which use from 6 to 128 different freelists, manage to do the same, but with a very high cost in external fragmentation.

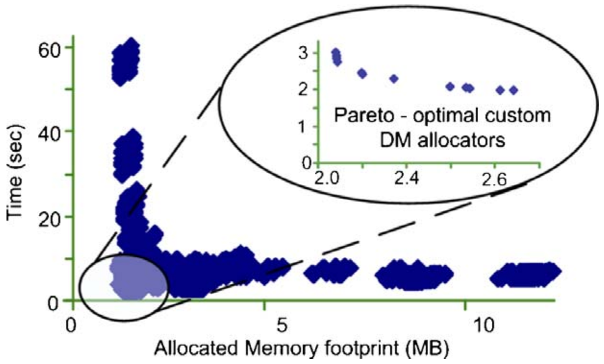


Fig. 2. Custom DM allocation exploration example for the Easyport buffering application and pareto-optimal DM allocators.

Table 2
Effect of de-fragmentation techniques in networking applications

Defrag. Techniques	Int. Fragmentation	Ext. Fragmentation	Performance
Freelists	– –	**+**	**+ +**
No Freelists	+ +	–	– –
Best fit	– –	None	– –
First fit	**+ +**	**None**	**+ +**
Split +	– –	**None**	– –
Split –	+ +	None	+ +
Coalesce +	**None**	– –	**+**
Coalesce –	None	+ +	–

+ means increase, – means decrease and None means no effect.

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 [6,7]), in networking applications the memory usage form varies greatly [3] (in the upper 3 traces of Fig. 3 we can see thin and fat spikes, in the lower left trace of Fig. 3 we can see plateaus and in the lower right trace of Fig. 3 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 OS 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 dramatically performance 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 [7]), 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. 4). This produces exceptionally high values of internal fragmentation, which is different from what is observed in other application domains. All the OS 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*.

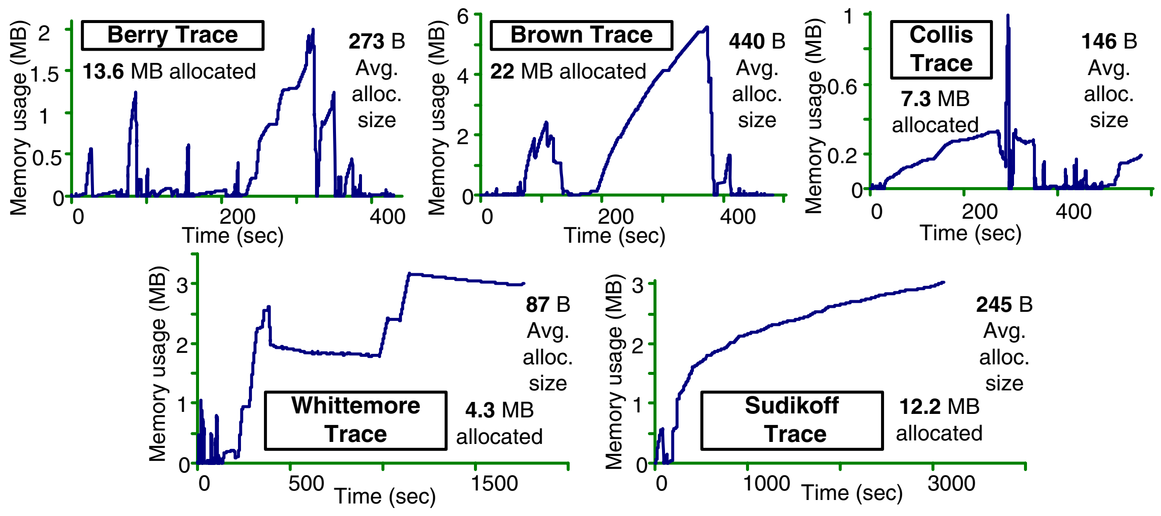


Fig. 3. Real memory usage of the DRR application for wireless traffic traces of different buildings [4] (50.000 packets).

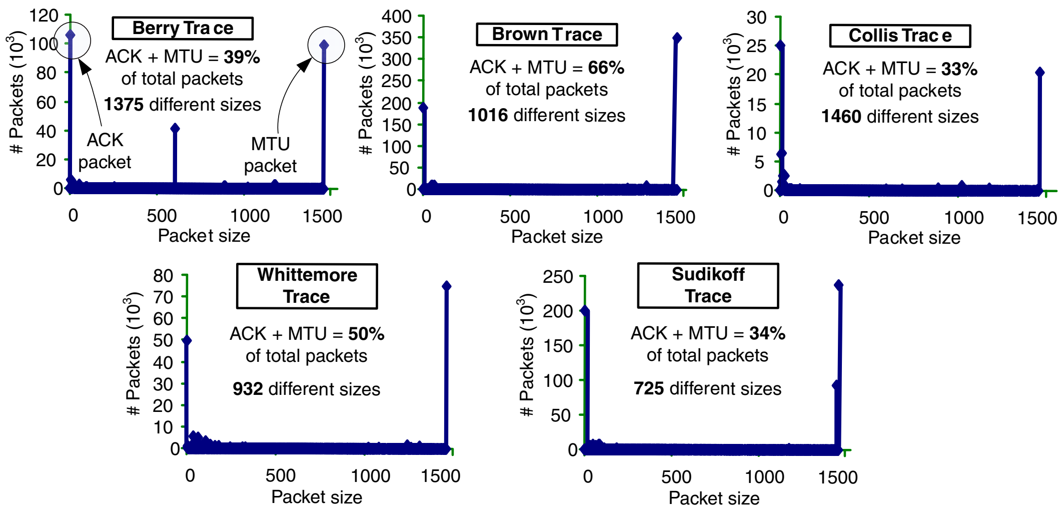


Fig. 4. Histograms of memory allocation requests of the DRR application for wireless traffic traces of different buildings [4].

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 transfer 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 OS 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 completely.

6. Case studies and simulation results

We have applied the proposed custom DM allocator to three real case studies that represent the wired and wireless network application domains. The first case study is Deficit Round Robin (or DRR) [16], a scheduling algorithm from the wired and wireless network domain [17]. The second case study is Easyport from Infineon [18], a buffering algorithm from the wireless network domain. The third case study is the URL-based switching algorithm [19] from the wired network domain. The real wireless and wired traffic input traces were obtained from [4] and [5], respectively.

The simulation results have been obtained using a Pentium IV at 2.4 GHz, with 1 Gbyte of SDRAM. To measure fragmentation we have used the cost function presented in Section 3. To measure performance we have evaluated the execution-time overhead caused by the DM allocator.

The first case study is the Deficit Round Robin (or DRR) [16] application, which is a scheduling algorithm implemented in many routers and WLAN Access Points today [17]. 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.

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 [7], 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 Figs. 3 and 4, respectively). The effect of the different dynamic behaviors can be seen in the variation of the simulation results (as shown in Table 3).

After an exhaustive exploration of the all the custom DM allocators (as explained in the previous section), we select to use 2 *freelists* for memory blocks of 16 bytes and memory blocks of 1476 bytes and we apply fully 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).

Then we simulate and compare our customized DM allocator with OS based DM allocators for 5 different network traces [4] (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

Table 3
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 (s))					
	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

our custom DM allocator is both faster and has less fragmentation than any OS 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 [18]. 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 with ftp sessions) (Table 4).

After an exhaustive exploration of the all the custom DM allocators (as explained in the previous section), we select again to use 2 *freelists* for memory blocks of 66 bytes and memory blocks of 1514 bytes and we apply fully 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 OS 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).

The third case study presented is the URL-based switching wired network application from the Netbench benchmarking suite [19], which is a commonly used context-switching mechanism. The algorithm works as follows: a table with patterns is formed. During the initialization phase, each pattern corresponds to a specific kind of data (e.g., to be served by a particular server). Therefore, there is a route indicating where packets containing similar patterns should be forwarded to. This route is also written to the table of patterns. During the normal execution phase of the application, the header of each packet is parsed and searched to find possible similarities to a pattern in the aforementioned table. Then, the packet is forwarded to the route corresponding to the pattern match that was found (Table 5).

After an exhaustive exploration of the all the custom DM allocators (as explained in Section 5), we select to use 3 *freelists* for memory blocks of 8, 11 and 20 bytes and we apply fully the *coalescing mechanism*, the *splitting mechanisms* and the *first fit policy*. In this case, we have chosen to add a third *freelist* instead of using just two, because the dominant memory blocks have a small size, so they would not contribute greatly to external fragmentation (as explained in Section 4). This specific custom DM allocator is the most balanced and operates without sacrificing fragmentation or speed (i.e., some OS allocators manage to be slightly faster but with big fragmentation overhead). We observe that it can achieve memory fragmentation reductions up to 98.13% (88.64% on average) and execution time reductions up to 7.42% (1.64% 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

Table 4
Simulation results for the Easyport buffering algorithm running for 4.200 packets per trace (lower fragmentation and execution–time is better)

DM allocators	Fragmentation				Performance (execution–time (s))			
	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

Table 5
Simulation results for the URL-based switching algorithm running for 50.000 packets per trace (lower fragmentation and execution–time is better)

DM allocators	Fragmentation					Performance (execution–time (s))				
	Avrg. (%)	BWY (%)	COS (%)	UFL1 (%)	UFL2 (%)	**Avrg.**	BWY	COS	UFL1	UFL2
Windows CE	**5**	8	2	8	2	**3.23**	3.27	3.21	3.20	3.27
Windows XP	**14**	16	16	16	10	**3.37**	3.39	3.36	3.37	3.39
Linux	**25**	24	25	25	26	**3.03**	3.05	3.01	3.01	3.05
Enea OSE	**75**	86	78	65	73	**3.11**	3.14	3.1	3.09	3.14
uClinux	**99**	96	107	96	97	**3.00**	3.04	2.97	2.96	3.04
Avrg. Alloc.	**44**	**46**	**46**	**42**	**42**	**3.15**	**3.17**	**3.13**	**3.12**	**3.17**
Proposed Alloc.	**5**	8	2	8	2	**3.13**	3.15	3.12	3.12	3.15

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 98% and improve performance up to 97% compared to the state-of-the-art, OS based DM allocators.

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.

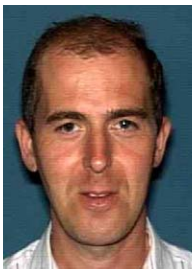
References

[1] D. Atienza, S. Mamagkakis, F. Catthoor, J. Manual Mendias, D. Soudris. Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications, in: Proceedings of IEEE/ACM DATE 04, France, 2004.
[2] D. Atienza, S. Mamagkakis, F. Catthoor, J. Manual Mendias, D. Soudris. Modular Construction and Power Modelling of Dyn. Mem. Managers for Embedded Systems, in: Proceedings of LNCS PATMOS'04, Greece, 2004.
[3] C. Williamson, A tutorial on Internet traffic measurement, Proc. IEEE Internet Comput. 5 (6) (2001).
[4] D. Kotz, K. Essien, Analysis of a campus-wide wireless network, In: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking. 107118. Revised and corrected as Dartmouth CS Technical Report TR2002-432.
[5] National Laboratory for Applied Network Research, <http://www.nlanr.net/>.
[6] P.R. Wilson, M.S. Johnstone, M. Neely, D. Bowles, Dynamic storage allocation, a survey and critical review, in: International Workshop on Memory Management, UK, 1995.
[7] M.S. Johnstone, P.R. Wilson. The memory fragmentation problem: Solved? in: Proceedings of the International Symposium on Memory Management, 1998.
[8] E.D. Berger, B.G. Zorn, K.S. McKinley, Composing high-performance memory allocators, in: Proceedings of ACM SIGPLAN PLDI, USA, 2001.
[9] Dynamic Allocation in uClinux RTOS, <http://linuxdevices.com/articles/AT7777470166.html/>.
[10] Dynamic Allocation in Enea OSE RTOS, <http://www.realtime-info.be/magazine/01q3/2001q3_p047.pdf/>.
[11] Dynamic Allocation in Symbian RTOS, <http://www.symbian.com/developer/techlib/v70docs/sdl_v7.0/doc_source/reference/cpp/MemoryAllocation/RHeapClass.html#%3a%3aRHeap/>.
[12] Dynamic Allocation in MS Windows CE, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50conheaps.asp/>.
[13] Dynamic Allocation in MS Windows XP, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngenlib/html/heap3.asp/>.
[14] G. Attardi, T. Flagella, P. Iglio, A customizable memory management framework for c++, Software Pract. Exper. 28 (11) (1998).
[15] David F. Bacon, Perry Cheng, V.T. Rajan. A Real-time garbage collector with low overhead and consistent utilization, in: Proceedings of SIGPLAN 2003, pp. 285–298.
[16] M. Shreedhar, G. Varghese, Efficient fair queuing using deficit round robin, in: Proceedings of SIGCOMM 1995, pp. 231–242.
[17] M. Gerharz, C. de Waal, M. Frank, P. James, A practical view on quality-of-service support in wireless ad hoc networks, in: Proceedings of IEEE ASWN 2003, Switzerland, 2003, pp. 185–196.
[18] Infineon Easyport, <http://www.itc-electronics.com/CD/infineon%2010063/cd1/html/p_ov_33433_-9542.html>.
[19] G. Memik, W.H. Mangione-Smith, W. Hu, NetBench: a benchmarking suite for network processors, in: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, pp. 39–42.

Stylianos Mamagkakis: Stylianos Mamagkakis received his Diploma in Electrical and Computer Engineering from the Democritus University of Thrace, Greece, in 2002. He is currently a senior PhD candidate in the VLSI Design and Testing Center in the Democritus University of Thrace. His research interests include optimizations in dynamic memory management on wired and wireless network applications for low power and high performance, embedded systems using high-level design optimizations. He has published more than 14 papers in international journals and conferences. He was investigator in four research projects funded from the Greek Government and Industry as well as the European Commission. He is a member of the IEEE.

Christos Baloukas: Christos Baloukas received his Diploma in Electrical and Computer Engineering from the Democritus University of Thrace, Greece, in 2004. He is currently a post graduate student in the VLSI Design and Testing Center in the Democritus University of Thrace. His research interests include dynamic access and storage optimization on communications applications for low power and high performance, embedded systems and high-level design optimizations.

David Atienza: David Atienza received the M.Sc. and PhD degrees in Computer Science from Complutense University of Madrid (UCM), Spain in June 2001 and June 2005, respectively. Currently he is Post-Doc at the Integrated Systems Laboratory at EPFL, Switzerland. He also holds the position of invited Assistant Professor at the Computer Architecture and Automation Department (DACYA) of UCM. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on dynamic memory management on embedded systems, flexible Networks-On-Chip (NoC) interconnection paradigms for Multi-Processors System-on-Chip, design automation and low-power design. In these fields, he is reviewer and co-author of various publications in prestigious journals and international conferences: ACM TODAES, IEEE Trans. on VLSI Systems, VLSI Journal, Journal of Embedded Systems, DATE, DAC, etc. Also, he is part of the Technical Program Committee of the IEEE/ACM DATE conference.



Francky Catthoor: Francky Catthoor received a PhD in El. Eng. from the K.U. Leuven, Belgium in 1987. Since then, he has headed several research domains in the area of architectural methodologies and system synthesis for embedded multimedia and telecom applications. His current research activities mainly belong to the field of system-level exploration, with emphasis on data storage/transfer and concurrency exploitation, both in customized and programmable (parallel) instruction-set processors. All this within the DESICS division at IMEC, Leuven, Belgium where he is currently a research fellow. He is also professor at the K.U. Leuven. He has (co-)authored over 500 papers in international conferences and journals, and has worked on 8 text books in this domain. He was the program chair and organizer of several conferences including ISSS'97 and SIPS'01.

Dimitrios Soudris: Dimitrios Soudris received his Diploma in Electrical Engineering from the University of Patras, Greece, in 1987. He received the PhD Degree in Electrical Engineering, from the University of Patras in 1992. He is currently working as Ass. Professor in Department of Electrical and Computer Engineering, Democritus University of Thrace, Greece. His research interests include low power design, parallel architectures, embedded systems design, and vlsi signal processing. He has published more than 140 papers in international journals and conferences. He was leader and principal investigator in numerous research projects funded from the Greek Government and Industry as well as the European Commission (ESPRIT II-III-IV and 5th and 6th IST). He has served as General Chair and Program Chair for the International Workshop on Power and Timing Modelling, Optimisation, and Simulation (PATMOS). Recently, received an award from INTEL and IBM for the project results of LPGD #25256 (ESPRIT IV) and two awards in ASP-DAC 05 and VLSI 05 for the project AMDREL (5th IST-2001-34379). He is a member of the IEEE, the VLSI Systems and Applications Technical Committee of IEEE CAS and the ACM.

Antonios Thanailakis: Antonios Thanailakis was born in Greece on August 5, 1940. He received B.Sc. degrees in physics and electrical engineering from the University of Thessaloniki, Greece, 1964 and 1968, respectively, and the M.sc. and PhD Degrees in electrical engineering and electronics from UMIST, Manchester, UK in 1968 and 1971, respectively. He has been a Professor of Microelectronics in Department of Electrical and Computer Eng., Democritus University of Thrace, Xanthi, Greece, since 1977. He has been active in electronic device and VLSI system design research since 1968. His current research activities include microelectronic devices and VLSI systems design. He has published a great number of scientific and technical papers, as well as ve textbooks. He was leader for carrying out research and development projects funded by Greece, EU, or other organizations on various topics of Microelectronics and VLSI Systems Design (e.g., NATO, ESPRIT, ACTS, STRIDE).