Cache Based Dynamic Memory Management For GPS

David Livshits, Yair Wiseman
Computer Science Department
Bar-Ilan University
wiseman@cs.biu.ac.il

***Abstract* — Operating Systems of Embedded Devices can manage application data more effectively. This directed us to consider a new dynamic memory management approach for GPS and similar embedded systems. The uniqueness of the suggested approach is shifting elements of the application to the Operating System with the aim of manage application data more effectively. A key component of an application is caching its data with the purpose of preventing excessive memory accesses and allocations. The suggested approach called "Cache Based Dynamic Memory Management" – CBDMM proposes to move the caching component from the application to the Operating Systems. We present CBDMM design, discusses its advantages and show encouraged benchmarking results.**

I. Introduction

The use of GPS (Global Positioning System) turns out to be the standard of rent a car companies, taxis and actually it grows to be the standard of many conventional car. The adaption of all-purpose Operating Systems like Linux for GPS devices is an up-and-coming field [1]. A GPS is an embedded and Real-Time device; therefore the management of its resource is done differently [2].

An embedded/Real-Time environment adds additional constrains on the OS (Operating System), such as a lack of resources (memory, storage) and performance concerns. Most of the embedded devices do not have a permanent storage device like a hard disk. The reason is the high cost of the storage device or the necessary size that prevents the manufacturer from including the device in relatively small embedded systems. CPU that is used by an embedded device is usually much cheaper and as a result much slower than a regular PC CPU. In addition, there are predefined restrictions imposed on RTOS (Real Time Operating Systems) such as a deterministic time response in most of RTOS possible service requests. In this environment memory management is always tensed between two main constraints: a lack of memory for dynamic allocations and a deterministic time response for a memory allocation request. The survey below shows that RTOSes are divided into two main categories:

- RTOSes preferring deterministic memory allocations over a possible fragmentation.
- RTOSes prefferring a minimal fragmentation over deterministic memory allocations.

Another important issue is the performance - allocation policy producing a minimal fragmentation can consume too much resources and CPU cycles.

II. Related Works

The following survey reviews the basic dynamic memory allocation methods of the most common RTOSes.

- VxWorks[3]: The memory management of VxWorks RTOS was replaced several times. The early versions of VxWorks implemented the first-fit allocation policy. Later versions of VxWorks implemented the best-fit allocation policy that according to some benchmarking causes less memory fragmentation and a better time response.
- µC/OS-II[4]: Each task creates a pool of memory partitions. Each partition is planned for block allocations of a specific size. When a task requests to allocate a specific size of blocks the allocation should be done from the planned partition. This fixed block size allocation scheme helps to avoid fragmentation, because available memory is typically small in embedded applications. Allocations and deallocations of these fixed-sized memory blocks are done in a constant time and thus deterministic.
- Nucleus[5]: The dynamic allocation is made out of the memory pool that is defined by the application. The dynamic allocation policy is a standard linear "first-fit". A block split mechanism will be applied if the first fitted block is significantly larger than the requested size. An application can allocate any number of pools. Due to a possible external fragmentation, the first-fit based block allocation process is not deterministic, but the deallocation is deterministic.
- FreeRTOS[6]: The memory allocation algorithm simply subdivides a single array into smaller blocks as a dynamic memory allocator does and uses the best-fit allocation policy. An allocation of a block is always deterministic i.e. it always takes the same time to allocate a block).
- RT-Linux[7]: The original version of RT-Linux did not provide any mechanism for dynamic memory allocation, because RT-Linux is non-determinism, but later versions of RT-Linux 2.2 have some dynamic memory allocation facilities which are based on predefined pools of blocks with fixed size (the size is configurable). The RT-Linux

memory allocator searches for the most appropriate preallocated block with a size that is not bigger than twice of the requested size. If all the preallocated blocks are bigger than twice of the requested size, the big block is splitted into pieces sized as the requested block and the first piece is returned for this request. This approach is pretty fast and reliable but is not deterministic. Also this approach is not suitable for the stack [8]. There was another try to implement dynamic memory management in RT-Linux based on the TLSF [1] algorithm with worst case allocation complexity of O(1) [9].

All the above methods have no cache mechanism for freed blocks.

III. CBDMM

It can be concluded from the above survey that usually RTOSes almost do not use virtual memory mechanism because its resource consumption is quite high and it requires a storage device that in most of the systems does not exist, and as a result there is no memory caching on memory management level. Even Operating Systems that implement a virtual memory approach have an internal cache logic that works on the memory pages level and manages application pages but does not involve in the application logic. The common concept is that the application logic is out of scope of OSes and particularly RTOSes.

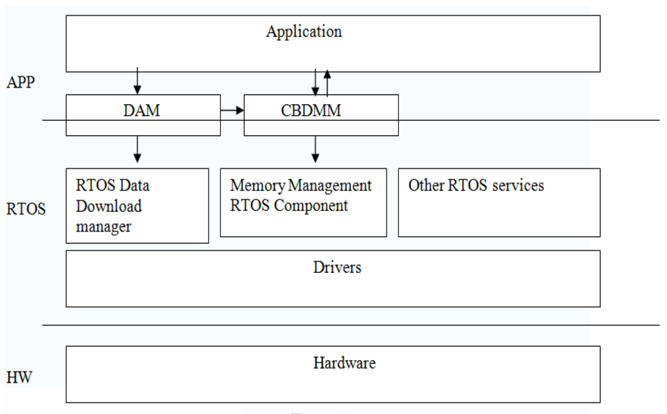


Figure 1.

Figure 1 demonstrates the separation of application and RTOS. When an application reads data from any asynchronous read only source (IP, DVB Broadcast, DSM-CC carousels, etc.) dynamic memory allocation may be required. The application requires the data using a data acquisition manager. In some system the data cycle-time is too long, so when some data is received, the application will try to save the currently unnecessary portions of data in the memory (if possible) for a later use, because most embedded systems do not have any storage device as was mentioned before or the storage device is too slow and the overlapping suggested in [10] will not be suitable. When there is no more room in the RAM for additional allocations, the application will free the data i.e. it is removed from memory until the next application access. It means that always when a new dynamic memory allocation fails, the application should release a data block which is part of some unused blocks list that the application needs to manage, and try to do the new allocation again.

Another problem is that the data download manager is not aware of the application logic, so when an application requests some data, the application should translate its request into the download manager language that works on packets/module level. So even if the dynamic memory manager is able to cache the unnecessary application, the data download manager must be aware of this caching so it will not reread the data from the source.

The main idea of the paper is adapting application logic to the dynamic memory management. Thus application can choose/provide an allocation policy for its dynamic memory allocation and also choose/provide the caching mechanism for caching/releasing allocated memory blocks. The new proposed dynamic memory management preferably should be the part of the RTOS and not a part of the application. Besides, the proposed scheme suggests making use of some kind of data acquisition layers that will manage cached memory blocks upon an application data request. The acquisition manager is only a sample of how the proposed system can work. It shows in the best way the advantages of the new approach and can be either a part of the application or fully integrated entity in the OS.

The following new components are defined:

Data acquisition manager (DAM) – is a client of the CBDMM and also an application client, the responsibilities of DAM are:

- Receiving data requests from an application and translating it into download manager requests.
- Managing application data requests with its statuses. When application decides to release the requested data it is done also using the DAM. In this case DAM marks the released data as cached and keeps it till the next allocation request.
- DAM is responsible for the dynamic memory allocations required for the data download, here DAM interacts with CBDMM
- DAM serves as repository for all downloaded data.

Cache based dynamic memory manager (CBDMM) – Memory manager. At each memory allocation request the CBDMM receives from DAM several figures: the list of required allocations, allocation policy, application callback, cache block list and cache policy application callback. After allocation transaction is succeeded, CBDMM will return to DAM the status of the cache list. Allocations and cache policies are managed by the application.

Figure 1 demonstrates this model.

How does it actually work?

- The Application requires the data from DAM using a predefined protocol or a wild card/regular expression mechanism. For example download file with name "account_info.xml" from the remote server using tcp/ip protocol.

- If the data is already loaded (in cache), DAM takes it from the cache and returns the requested data immediately (in synchronous way), otherwise builds new download request. Sequence diagrams on figure 2, 3, 4 describe these cases.

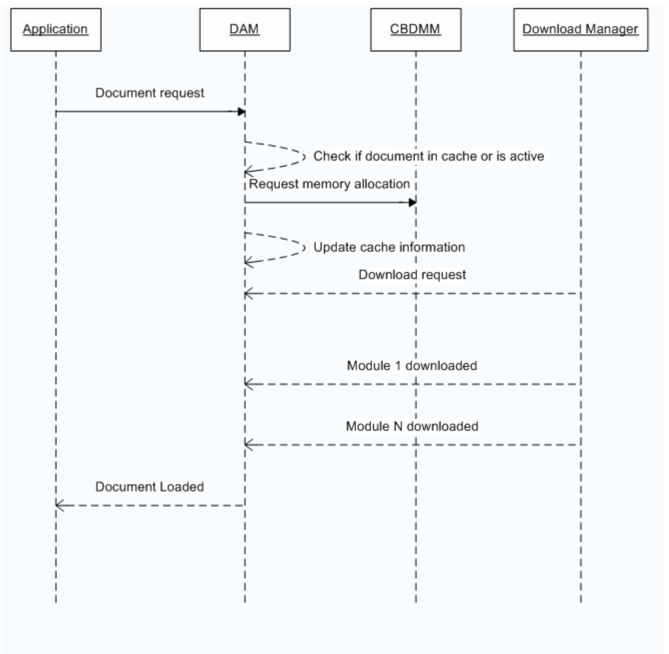


Figure 2. Shows the data request in case the data is not in cache

- New download request contains the following steps:
    - Allocation of the new memory blocks for the request – here DAM is responsible to notify CBDMM about all unused (cached) by application memory blocks that CBDMM can free in case of need, so during the Dynamic memory allocation request DAM passes the list of all cached memory blocks to the CBDMM and waits for result.
    - After memory allocation transaction is succeeded DAM updates the its cache list (some of the cached modules could be freed during new allocation request) and sends request to the download manager to download requested content into preallocated memory.
- DAM waits until all data's modules/parts are loaded and notifies the application that download transactions are done.
- If the application frees the data, DAM will keep it in the cache. Figure 4 shows the sequence diagram for this use case.

The CBDMM system includes two sub-systems:

- MM - Subsystem for memory allocation. Implements a number of allocation policies that application can choose from and/or uses allocation policies provided by an application.
- MC - Subsystem for caching the memory blocks. The cache policy is implemented by CBDMM client (DAM, Application) and can be chosen by an application.

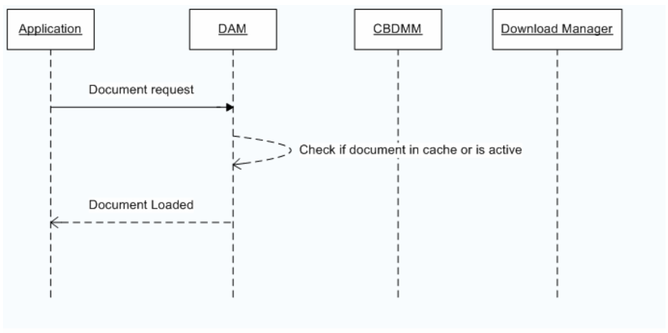


Figure 3. Shows the data request in case the data is in cache

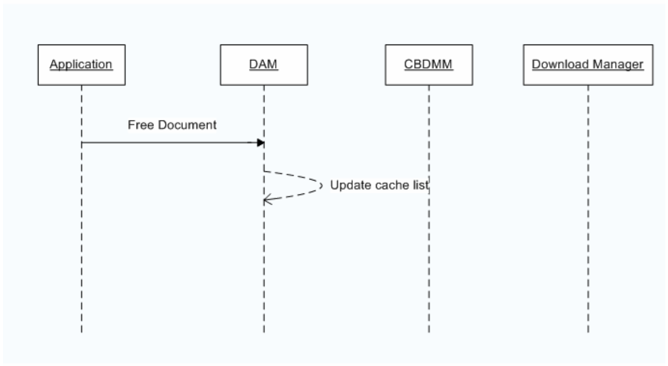


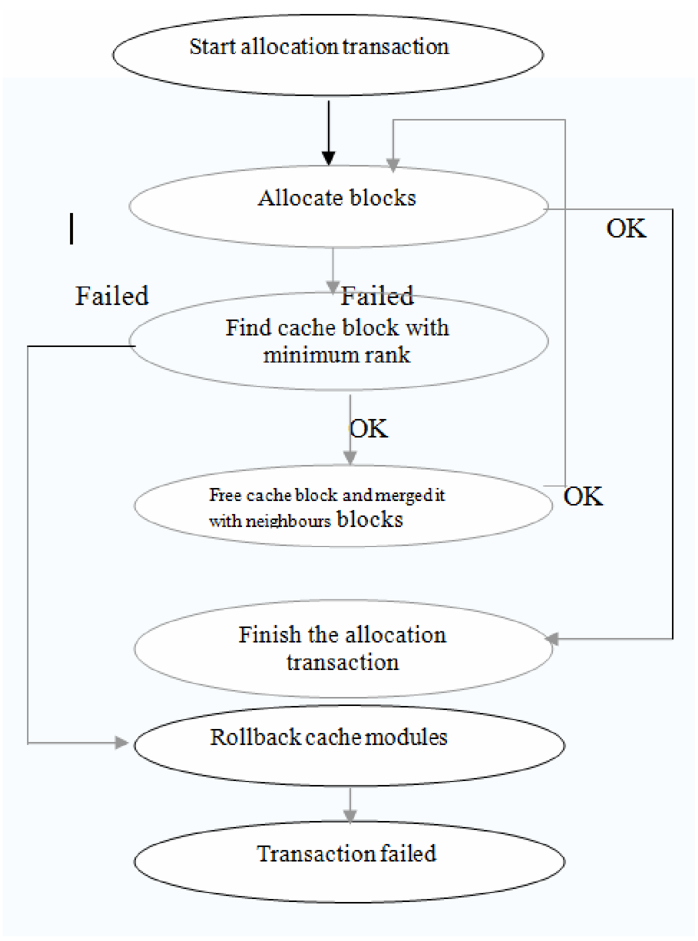
Figure 4. Shows the way application frees downloaded document. DAM continues to keep it in the RAM just marked it as cache.

Both sub-systems, together with the API user system, provide the following functionality:

- Efficient memory allocation— reducing the likelihood of memory fragmentation, by increasing the number of modules resident in the memory (MM).
- Efficient module acquisition—reducing the time required to fetch a module by caching redundant modules in memory when possible (MC).

Both MM and MC should reduce the module request time by reusing the cached modules. MC is effective only when the CBDMM user is aware of the need to release unused modules as soon as possible. If the allocation function cannot allocate the memory that is required, it will begin to release cached modules. In order to prevent an arbitrary release of cache modules, (or when there is a higher probability that certain cached modules will be in use in the future than others) each cached module contains special cache data that defines the rank of the module with regard to release. Modules will be released according to their rank, in low to high order.

In order to determine which module has the lowest rank, the CBDMM queries the cache policy and, according to the module's rank, releases the module. If releasing some non-required modules does not assist in providing the required space for the new module allocation, the release process will continue until all cache modules are released. If sufficient space is not freed the allocation transaction will roll back (i.e. will be cancelled). If sufficient space is freed, the module will be allocated and the next allocation transaction begins. Figure 5 demonstrates this flow.



Start allocation transaction
Allocate blocks
OK
Failed
Failed
Find cache block with minimum rank
OK
Free cache block and merged it with neighbours blocks
OK
Finish the allocation transaction
Rollback cache modules
Transaction failed

Figure 5. Block diagram for allocation transaction

IV. BENEFITS

The benefits of the proposed model are obvious: Applications have accessibility to their data in a maximal optimized way, because all the allocation policies and the cache policies are provided/can be choosen by the application that can predict the memory map better. This way the application can optimize the data accessibility. It should be noted that this approach can be combined with predictive cache mechanism, but this is out of scope of this proposal. Both DAM and CBDMM can be either a part of RTOS/Middleware [11] (preferable) or fully integrated into the application.

The survey above shows that nowadays there is no RTOS that has a cache mechanism for the dynamically allocated block scheme. In addition, the allocation policies that are used by RTOSes are not changeable or configurable by the application. So applications cannot be tuned to produce the minimal external fragmentation. In our suggested system if an application runs on different platforms, the tuning process can be done only once.

Some other interesting approaches have been developed during the years e.g. [12,13], where a particular cache mechanism is applied to the memory block size level, without any relation to the data that the memory block contains. These approaches try to use some heuristics to find which block size is more frequently used. The disadvantage of these approaches vs. CBDMM proposal is that the cached data of the cache blocks is usually irrelevant for further allocation requests and as a result new download request is required even in case the returned cached block already contains the requested data. In this case only time for memory allocation is saved and not time for downloading the data.

V. POSSIBLE DISADVANTAGES

The main disadvantage of this proposal is the overhead. According to this proposal the application can manage the memory allocation policy, so if the policy is not implemented by the RTOS, it should be implemented by the application itself. It can be less reliable than an OS implementation and may not meet performance requirements. The same relates to the cache policy which is also might be implemented either by an application or by DAM, but managing the cache block list still remains in DAM that can be part of an OS like [14] that handles page lists in the OS.

Another issue is the concept of application and OS logic separation. CBDMM breaks the border between the OS and the application and passes some OS logic to the application. Hopefully this feature will not place problems like an application which tries to handle some OS functionality that actually cannot be passed to the application even theoretically.

VI. EVALUATION

The benchmarking of system is based on some web browser like applications that have multiple presentation pages and links. Each page requires downloading a data. Page presentation (on the screen) time will be a benchmarking parameter. The benchmarking has been made on a GPS, but even a PC simulation could be enough. The simulation compares application performance based on the original RTOS and the RTOS with the described DAM/CBDMM model.

Some servers broadcast the data using IP multicast. Data is broadcasted with some predefined cycle time (depends on type of data and target application). There are handy embedded devices that receive data using a WIFI receiver and process it by certain applications. This function is fully portable [15,16]. The device does not have any internal storage unit and a browser application can consume only 512k RAM for its internal needs.

On the device site there is a browser using “Weather forecast” application which has to present 4 different screens with the forecast information. Each screen will be presented when the application enters a proper state. There are 4 equivalent states for 4 screens. Figure 6 shows the acceptable state switch.

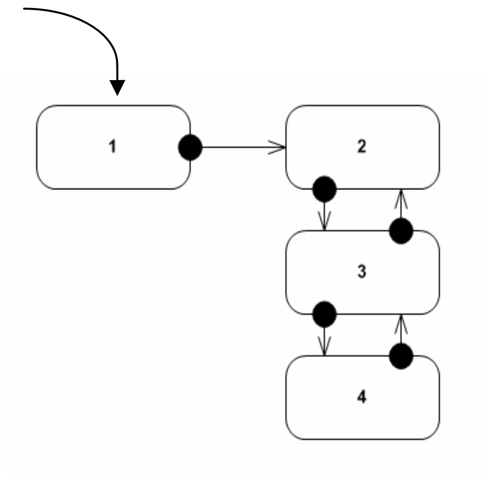


Figure 6. Application states

Each state requires a data for presentation. Table 1 shows the dependency between state and data modules:

State	Data modules
1	resources.mod presentation_1.mod text.mod ads.mod
2	resources.mod presentation_2.mod metadata_2.mod text.mod
3	resources.mod presentation_3.mod metadata_2.mod text.mod presentation_2.mod
4	resources.mod presentation_3.mod metadata_2.mod text.mod presentation_2.mod ads.mod

Table 1: Dependency between state and data modules

Table 2 defines data modules cycle time:

Module name	**Cycle Time (ms)**	**Size (bytes)**
resources.mod	1000	128000
presentation_1.mod	1000	128000
text.mod	1000	25000
ads.mod	1000	35000
presentation_2.mod	1000	130000
metadata_2.mod	1000	100000
presentation_3.mod	1000	30000

Table 2: Data modules cycle time

Application Test Plan (ATP) is a document that provides QC team the rules for testing the application. There is a test from the “Weather Forecast Application”. The ATP that has been used for benchmarking is “Weather Forecast Application” PC simulator based on CBDMM.

#	Test description	Time (ms)
1	Launch the Whether application	4087
2	Using the yellow button navigate to screen 2	2168
3	Using the arrows navigate to screen 3	1154
4	Using the arrows navigate to screen 4	2184
5	Using the arrows navigate to screen 3	47
6	Using the arrows navigate to screen 2	31
7	Using the arrows navigate to screen 3	31
8	Using the arrows navigate to screen 4	47

Table 3: Performance results of the first test

#	Test description	Number of Requests	Cache hit count
1	Launch the Whether application	4	0
2	Using the yellow button navigate to screen 2	4	2
3	Using the arrows navigate to screen 3	5	4
4	Using the arrows navigate to screen 4	6	5
5	Using the arrows navigate to screen 3	5	5
6	Using the arrows navigate to screen 2	4	4
7	Perform steps 3,4,5,6 20 times	400	400

Table 4: Cache hit ratio of the first test

#	Test description	Time (ms)
1	Launch the Whether application	4134
2	Using the yellow button navigate to screen 2	4134
3	Using the arrows navigate to screen 3	5132
4	Using the arrows navigate to screen 4	7191
5	Using the arrows navigate to screen 3	5148
6	Using the arrows navigate to screen 2	4119
7	Using the arrows navigate to screen 3	5132
8	Using the arrows navigate to screen 4	7191

Table 5: Performance results of the second test

#	Test description	Number of Requests	Cache hit count
1	Launch the Whether application	4	0
2	Using the yellow button navigate to screen 2	4	0
3	Using the arrows navigate to screen 3	5	0
4	Using the arrows navigate to screen 4	6	0
5	Using the arrows navigate to screen 3	5	0
6	Using the arrows navigate to screen 2	4	0
7	Perform steps 3,4,5,6 20 times	400	0

Table 6: Cache hit ratio of the first test

The first test uses CBDMM approach and an OS simulator which implements the test. The OS has been configured to use WorstFit allocation policy and LRF cache policy. When a new screen is requested all previous screen modules are moved to the cache list automatically and released (if required) using LRF policy. The data is requested correspondently to the specific state.

The results of the first test are shown in Table 3 and Table 4. The cache hit ratio for all the test steps is 420/428 = 0.981.

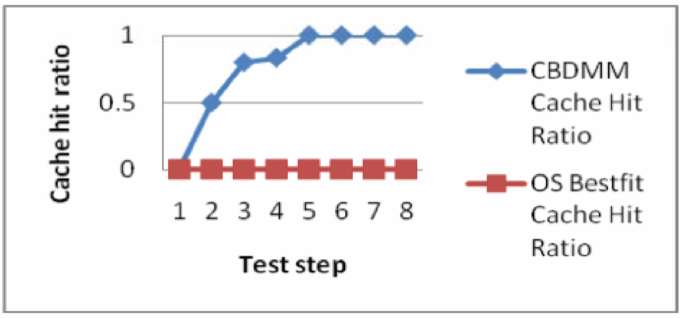


Figure 7. Cache hit ratio graph

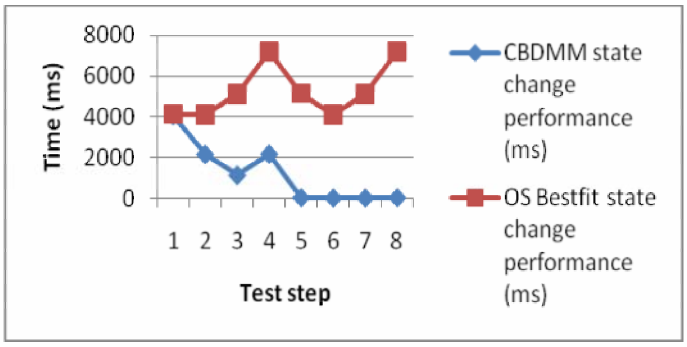


Figure 8. Performance graph

The second test uses only the Operating System dynamic memory management system calls. When a new screen is requested all previous screen data will be deleted using system free and a new memory space will be allocated with malloc(BestFit). The data is requested correspondently to the state. The results of the second test are shown in Table 5 and Table 6. The cache hit ratio of all the test steps is 0/428 = 0.

Figure 7 compares cache hit ratio results for general OS memory management and CBDMM. Certainly, a high cache hit ratio affects the data request's performance. Figure 8 compares performance results in milliseconds for general OS data requests and CBDMM.

VII. CONCLUSION

Memory management of navigation systems is typically done in the application level [17,18]. We showed in this paper that moving components of the memory management to the Operating System can enhance the performance of a conventional GPS. Particularly, we showed that the caching management mechanism should be the part of the RTOS and not part of the application. Benchmarking tests (Figure 7 and Figure 8) explicitly demonstrate that the CBDMM approach obtains a higher cache hit ratio and as a result a significant increase of data request's performance by choosing proper allocation and caching policies. In addition, the manipulation of allocation policies generates a better fragmentation of the heap. This flexibility will not be available when using standard Operating System memory management and data acquisition mechanisms which are completely separated in an all-purpose Operating System. In the future we consider adapting parallel approaches [19,20] so as to make the suggested system suitable for parallel processing.

REFERENCES

[1] P. Hongyan, H. Hong, J. Hengtian, "Drive design for ship GPS navigation equipment based on Linux operating system", International Conference on Educational and Network Technology (ICENT), pp. 384-388, Qinhuangdao, China, June, 2010.
[2] N. Chadil, A. Russameesawang, P. Keeratiwintakorn, "Real-time tracking management system using GPS, GPRS and Google earth", 5th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, pp. 393-396, 2008.
[3] Z. Laszlo, "Memory Allocation in VxWorks 6.0", Wind River Systems, pp. 2-3, 2005.
[4] J. J. Labrosse, "µC/OS-II The Real-Time Kernel, Published by CMP books", pp. 273-285, 2002.
[5] Nucleus PLUS Reference Manual, Accelerated Technology, Inc., pp. 39-52, 1999.
[6] http://www.freertos.org, October 2007.
[7] V. Yodaiken, and M. Barabanov, "A real-time Linux", Proceedings of the Linux Applications Development and Deployment Conference, 1997.
[8] Y. Wiseman, J. Isaacson and E. Lubovsky, "Eliminating the Threat of Kernel Overflows", Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008), Las Vegas, Nevada, pp. 116-121, 2008.
[9] L. Wang, C. Yang, X. Wang, "RTL-IO: An Extension of RTLinux I/O", Eighth Real-Time Linux Workshop, Lanzhou, Gansu, China, Oct. 2006.
[10] Y. Wiseman and D. G. Feitelson, "Paired Gang Scheduling", IEEE Transactions on Parallel and Distributed Systems, Vol. 14(6), pp. 581-592, 2003.
[11] Y. Wiseman, K. Schwan and P. Widener, "Efficient End to End Data Exchange Using Configurable Compression", Proc. The 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, pp. 228-235, 2004.
[12] S. G. Meier, "Dynamic memory allocation suitable for stride-based prefetching", Advanced Micro Devices., US Patent 6,295,594, 2000.
[13] Christopher Cyll, Williams College, Advisor: Duane Bailey, Cache conscious dynamic memory allocation, Consortium for Computing Sciences in Colleges, pp. 1-2, 2004.
[14] M. Itshak and Y. Wiseman, "AMSQM: Adaptive Multiple SuperPage Queue Management", International Journal of Information and Decision Sciences (IJIDS), Vol. 1, No. 3, pp. 323-341, 2009.
[15] I. Grinberg and Y. Wiseman, "Scalable Parallel Collision Detection Simulation", Proc. Signal and Image Processing (SIP-2007), Honolulu, Hawaii, pp. 380-385, 2007.
[16] M. Geva and Y. Wiseman, "Distributed Shared Memory Integration", Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2007), Las Vegas, Nevada, pp. 146-151, 2007.
[17] J. Courbon, Y. Mezouar, L. Lequievre, L. Eck, "Navigation of urban vehicle: An efficient visual memory management for large scale environments", IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2008), pp. 1817-1822, Nice, France, 2008.
[18] P. Crowley, J. Jaugilas, A. Nash, S. Natesan and D. S. Lampert, "Memory management for navigation system", US Patent 6,073,076, 2000.
[19] S. T. Klein and Y. Wiseman, "Parallel Lempel Ziv Coding", Journal of Discrete Applied Mathematics, Vol. 146(2), pp. 180-191, 2005.
[20] S. T. Klein and Y. Wiseman, "Parallel Huffman Decoding with Applications to JPEG Files", The Computer Journal, Oxford University Press, Swindon, UK, Vol. 46(5), pp. 487-497, 2003.