Managing Main Memory Augmented with NOR Flash in Handheld Devices

Gurmeet Singh, Gautam Barua
Department of CSE, Indian Institute of Technology, Guwahati

***Abstract*—The growth of Smartphone applications in code and data requirements calls for increased memory sizes. In this paper, we propose the use of NOR flash memory as a part of main memory due to the ability of NOR flash to execute-in-place. By replacing a portion of RAM by a much larger NOR flash memory costing the same, the effective size of main memory can be dramatically increased. Memory management schemes for the NOR flash have been devised which try to reduce internal fragmentation and the number of erases and writes as writes have to take place in large units. To evaluate the system and memory management schemes, and to evaluate the effect of the slower access times of NOR flash memory, simulation has been carried out with the system comprising only RAM and with the system containing both RAM and NOR flash memory. The results of simulation show that for a large value of system parameters, the system with hybrid main memory performs better than that with only RAM.**

Key Words:

I. Introduction

As the applications for handheld devices grow in code and data requirements, there has been a need for increased memory size. To increase memory capacity, flash memory is used as a secondary storage device due to its properties like non-volatility, low power consumption, etc. NOR and NAND are the most popular types of flash memory. Due to sequential access architecture and long random latency of NAND flash, its use for code storage and execute-in-place (XIP) applications is not well suited. NOR flash memory (NOR flash for short from now on) on the other hand allows short random reads, and hence have been used for XIP applications.

NOR flash has the unit of erasure as a block which typically consists of many pages. This can lead to large internal fragmentation within blocks along with existing external fragmentation. Further, the lifetime of NOR flash is dependent on the number of erases and writes performed on it. NOR flash is also about half as slow as standard RAM. Of course NOR flash is much cheaper than RAM and so much larger memory can be placed at the same cost. Due to the above properties, it is clearly not feasible to replace RAM by NOR flash. But it is possible to have a hybrid main memory consisting of both RAM and NOR flash. For example, a system with 512KB of RAM and 8 GB of NOR flash will cost the same as a 1GB RAM-only system.

Our work focuses on proposing memory management schemes for the main memory RAM augmented with NOR flash, which we call as dual-natured main memory in the paper. The heuristics for managing memory have to look into the issues of reducing internal fragmentation and the number of erases performed in NOR. To reduce the number of erases, as well as to have code available in NOR flash when the application is run next time, the code is kept in NOR flash even after the process has finished execution. The schemes have been proposed for both systems having support of virtual memory and those without VM support.

This paper is structured as follows. The next section mentions about the existing work related to the use of NOR flash for execute-in-place applications. The following section describes the memory management schemes for the dual-natured main memory for systems without virtual memory support. Suggestions for a system with virtual memory are also made. The simulation results are discussed in section VI. We conclude with possible future work in section VII.

II. Review of Prior Work

The use of NOR flash to store code and execute-in-place has been mentioned by the research community in a number of papers. For instance [11]–[13] identify the capability of NOR flash to XIP, but do not work towards managing flash memory for its efficient use. Further, [8] mentions the use of flash memory as non-volatile RAM and making frequently used applications' code available in memory as a new way to exploit flash memory's capabilities in system design.

Some authors have also explored the implementation of XIP in NAND flash. [13] proposes a memory architecture for implementing XIP in NAND flash memory, made possible by using a pre-fetching cache to hide memory latency resulting from NAND memory access. But NOR flash is more suited for inclusion in main memory due to its short random reads compared to long sequential reads of NAND flash, as suggested by [11].

Along the lines of building main memory as a composition of RAM and NOR flash, and proposing memory management algorithms for the system, little work has been done so far. Recent work by Mogul et. al. [1] encourages use of flash memory in building a main memory as a hybrid between volatile DRAM and non-volatile memory and suggest management strategies for this hybrid memory. Their use of maintaining an erase-count for flash memory, to ensure endurance of flash memory, is similar to our approach of maintaining erase-count for flash memory and allocating memory uniformly throughout the NOR flash memory space. However, they do not directly address the smartphone scenario and NOR flash as the non-volatile component of its main memory, while our memory management strategies are particularly for the smartphone

environment with the use of NOR flash. Along with that, we present memory management strategies in detail, focusing on memory allocation and swapping/page replacement. We also present heuristics to manage memory when large read-only data needs to be kept in memory when running video or VoIP applications on smartphones.

III. Cost and Time Considerations

The prime considerations while proposing a system with different memory configurations are the effects on cost, power and access time of data. We intend to propose a system with nearly the same cost as that of RAM-only systems presently in existence. NOR flash has lower power consumption (both idle and active state power) as compared to RAM [4]. Although the access time of NOR flash is more than that of RAM, its effect can be reduced, as discussed later.

As mentioned in [6], typical use of XIP requires only 32 MB RAM and 48 MB flash compared to SnD (Store and Download) situation which needs 64 MB RAM and 48 MB flash for the same execution. The cost of extra RAM saved can be used to increase memory available as flash, in order to keep a larger pool of code of applications for faster start up of applications.The experiment suggests that if M is the size of RAM used in existing systems, then XIP can reduce RAM usage to $M/2$. While at the time of [6] the costs of memory were different, recent fall in prices of RAM and NOR flash allow us to increase the NOR flash memory beyond the value mentioned therein. The present prices of the memory components are around $28 per GB of mobile SDRAM and $17 for 16 GB of NOR flash. This suggests that a system with 1 GB RAM will cost around the same as that with 512 MB RAM and 8 GB NOR flash.

The other important factor is the memory access time. RAM and NOR flash have respective access times of approximately 40ns and 70ns. We assume only L1 cache is present in a smartphone configuration. The L1 instruction caches have access times of around 1ns and have a hit ratio of about 0.95, so the probability of main memory access for code is 0.05. This implies that the effective memory access time for instructions of a system is 2.95. If code is kept in NOR, then the effective access time becomes 4.45, a 50% increase. For data, the access times will be the same as it is assumed that they are in RAM in both situations and for simplicity, we assume that there is no data cache. For the system proposed with hybrid main memory, an important factor that plays a role in calculating effective memory access time is the ratio of amount of code and data in the processes. The benchmark programs used by authors in [14] have a probability of data (that is, the ratio of size of data segment to the total process size)ranging from a low of 0.016 for heap sort, to a high of 0.424 for the data intensive $10 * 10$ matrix multiplication. Firefox web browser's files were analysed in a linux-based system using pmap command the average probability of Data as 0.0165. For a Mozilla Thunderbird Mail Client it was found to be 0.293. All these values suggest that Probability of Data is in the range 0.01 to 0.4. If we assume that the access to data and code is directly proportional to the sizes of these segments, then this implies that effective memory access times will be as follows:

Probabilty of data = 0.01
for a RAM only system: 0.99*(2.95) + 0.01*(40) = 3.3205ns
for a hybrid system: 0.99*(4.45) + 0.01*(40) = 4.8055ns

Probabilty of data = 0.4
for a RAM only system: 0.60*(2.95) + 0.40*(40) = 17.77ns
for a hybrid system: 0.60*(4.45) + 0.40*(40) = 18.67ns

We can see that as the application becomes more data intensive, the performance improves. In any case, customer satisfaction for a smartphone user largely depends on how fast the applications start, and how easily the user can switch between applications. If a completed process residing in main memory is reused by a customer, then the application loads in no time. Moreover, the reduced number of swap-out of processes due to increased main memory will result faster switching between applications. So we intend to maximize the number of times a process residing in main memory is reused, which is our parameter of evaluating the proposed system later in the paper.

IV. Memory Management

Our work focuses on finding memory management schemes for this dual natured main memory composed of RAM and NOR. The standard method used is to do contiguous allocation of space (physically contiguous) and to use the first fit algorithm to find empty space. In our case, we have the following special issues:

1) Memory allocation has to be in units of "blocks" which are much larger than normal page sizes (64KB vs 4KB) and so there can be significant internal fragmentation besides external fragmentation if the standard techniques are used.
2) The NOR flash's longevity is dependent significantly on the amount of 'wear' it has, so the wear should be minimised and distributed over the entire memory space. The proposed scheme ensures that we handle these issues efficiently.

Memory allocation for a process is done using a small modification to the standard first fit algorithm. We use the circular-first-fit algorithm which starts the search for the first empty space in memory greater than the required space from the point where it last found empty space. This will ensure that the memory allocation is uniform over the memory space leading to less wear of each memory location. When a certain number of blocks are allocated to a process's code for execution, there can be significant amount of internal fragmentation in the last block, which, on an average, is 32KB. Since swapping out of a process means erasing its blocks from NOR, the part of the memory wasted by internal fragmentation can not be used for any other process but it can be utilised to store the read-only data of the same process, if any.

Another way to ensure longevity of flash memory is to keep a process's contents brought in NOR for longer duration than

required, so as to facilitate its reuse in near future without erasing it and without needing to erase some other process's contents before writing this process's contents. Further, this would also lead to faster start-up of the application when next run, a facility that will greatly increase user satisfaction.

The swapping algorithm for such a system should ensure that the processes which have completed and are in flash memory should be the preferred victims for swapping out as compared to those which are actively being executed. Among the completed processes are two broad classes. First one is of those which have completed execution but the application has not been closed by the user yet, for instance a song being played on the music system has finished but the music player is not closed by the user and hence is running in the background. Such processes should be less preferred victims over the second class of those processes whose parent application has been closed by the user. To account for this, we maintain a list of processes of closed applications called the Closed List and a list of active processes that are waiting called the Waiting List. This list includes the presently executing ones, and also those which have completed but the user has not closed the application and so can have a possible execution in the near future.

Now the task of the swapping algorithm is to find a victim process to be erased from NOR flash which can create enough space for the process that needs flash memory allocation. To minimise the number of time the swapping algorithm is run, and to minimise the number of processes swapped out from flash, we choose the process with the largest amount of memory allocated among the eligible processes. We first consider the Closed List, which is hence sorted on the basis of amount of memory allocated in flash to the process. The largest process in this list is chosen as the victim. In case there is a tie, we choose that process which resides in the location of memory that has been erased the least number of time in the past. If the size of the incoming process is larger than the largest process of the Closed List, we select the set of N processes from the List which together are of size larger than the incoming process's size and are the N largest processes of the Closed List. Again, to select among more than one process of the same size, we select the one residing at the least erased memory locations.

In case the Closed List does not have sufficient number of processes to account for the incoming process, we consider the processes in the Waiting List to be erased. First, we select all the existing processes in the Closed List. Next we select processes from the Running List to create enough space for the incoming process. These processes, in most cases, would be the ones that have completed their execution but the applications are not closed by its user. However some active processes may also become victims. If we sort the Waiting List on the basis of size, then active processes may become victims ahead of completed ones. We could have a separate list for completed processes, different from the Waiting List, and maintain it as a sorted on the size. But it is not easy to distinguish between a completed process and an active one as both may be waiting on user input. So we maintain a single list which is maintained as an LRU list. If the process which just executed is brought to the beginning of the list, the processes at the end of the list would eventually be the ones which have completed. This is because the scheduler will no more schedule them for execution and completed processes would gather at the end of the list. Now we find out the minimum number of processes K which can create the space required in flash memory.

V. Data Buffering in NOR Flash

Data buffering is important for streamed video and audio. In a smartphone these applications are likely to be used heavily. So current strategies in use presently hold the scope of improvement when we have large amounts of NOR flash available. According to [3] iPhone streams data till its buffer is filled, pauses download when buffered content exceeds a threshold, and then resumes download after the content falls below another threshold, hence leading to jitter. On the other hand, G2 from Android streams video every 10 seconds, and Samsung and Palm phones every 20 seconds. Our proposal of memory management schemes which gives larger buffer size will solve the problem of jitter in iPhone type systems as well as decrease the time interval after which G2, Samsung and Palm resume download.

Further observations were made on Nokia E-series smartphones. YouTube videos were streamed and in the middle of video being played the video was taken back to previous location in timeline. On doing this, the data contents preceding the present time by few seconds had to be buffered again. This suggests that in such systems once a part of video has been watched, its contents are lost. Further, such a scheme does not support replay of the video without re-buffering. With the larget memory available now, the buffer sizes can be significantly increased. This will reduce jitter and will allow replay of large chunks of streamed data.

We propose to keep the read-only data of video or VoIP based applications in the NOR flash component of the main memory. This idea is supported with the requirement of NOR to accommodate data which does not need to be modified, and needs to reside there for a sufficiently long time so as to reduce the number of erases. The incoming rate of such data is also relatively slow and so the slower NOR speeds will have no impact on user experience. For allocating memory, one of the options is having a part of the NOR memory reserved for read-only data (eg. video/VoIP etc.). This data section of NOR flash could, however, be used by code pages if required. Since the code being kept in data section would be susceptible to be removed on arrival of data in memory, the code should be for low priority applications and the direction of arrival of data should be opposite to that of the code in the section. In other words, when code partition of NOR is near full, incoming code pages of low priority applications are written in data section of NOR (from higher addresses to lower) and when in the future a video is streamed in, the data is brought in the section (from lower to higher addresses) replacing any existing code if needed. Also, when the read-only data pages of data partition of NOR are stale, they can be replaced by further incoming pages of low priority applications.

The other option is to treat code and data pages of NOR alike and keep them interleaved, wherever the space is available. Such a scheme, can result in replacing a code page (which will surely be used) for a data page which was used only once or perhaps just buffered but not used at all, like in the case of not watching a paused video which kept on buffering and taking up much memory.

Hence we plan to implement the first scheme of keeping the sections of data and code separate, but using them interchangeably for code and data, to provide flexibility as well as more efficient memory management. The flexibility is provided at two levels, keeping the code in data segment of NOR flash and placing data into the code segment if required.

- **Keeping code in data segment:** As already mentioned, the code of low priority applications can be brought into data segment of NOR flash in the direction opposite to that of allocation of data in the segment. Note that this is done only in the situation when the data segment has sufficient space, no buffering is taking place and the code segment is nearly full.
  This suggests that we need to guarantee some empty space in data segment of NOR at all instances to prevent erasure of code or data before more data can be buffered. Hence, we should erase the data pages (of other processes) which were brought into NOR the earliest (LRU, to provide the facility of going back in the video). Note that while going back in the video mandates availability of most recent data, replaying a video requires availability of least recent data of the process/video as well. The preceding discussion ensures the availability of most recent data, and that of least recent data to support Replay is an issue if the data being buffered is more than size of data segment. This is handled by keeping data in code segment as will be discussed later.
  Once the data segment is near full, it will lead to replacing pages from the segment. If a code page is removed from the segment it is simply erased if the process is closed, or put back to code segment. The code page can be required to be removed from data segment by either incoming data or more code into the segment. On the other hand if the page to be removed is data page, it is simply erased. The data page can also be required to be replaced by either a data page or code page, latter only if existing data is stale. This gives rise to the following four situations, explained assuming data enters the segment from the top of segment and code fills it from the bottom:
  1) *Code replaces code:* Code will replace the code pages which were brought earliest into the data segment. This is possible only when this process has been selected as victim to be swapped out and its code has filled the maximum amount of data segment memory allowed for code. Note that in this case the code pages being replaced need not be written to the code segment of NOR flash because this situation is analogous to running page replacement policy for the entire memory space. More incoming code will start erasing and replacing the pages from the bottom.
     In case the system does not support virtual memory, then the data segment would be having entire code of the process first brought into data segment at the bottom (contiguous allocation to support FIFO and LIFO policies), followed by the next one in order (above it in memory) and so on. Replacement would then hence erase all the code pages of first brought-in process the first, followed by the next ones in FIFO order, same as the allocation order.
  2) *Data replaces data:* Once the data segment has been filled by the data buffered in, more data will replace existing data pages again in FIFO manner in order to ensure that latest pages are available for going back in video, as discussed earlier. The data being buffered hence replaces the data pages at the top and continues downwards.
  3) *Data replaces code:* If the data segment is filled by code, incoming data will need to replace code pages. The code pages being replaced will be the ones which are the topmost among all code pages, hence implying LIFO policy. This is a favourable in that replacing the page brought in most recently to NOR would be an indication to the actively running application to not place its code into data segment any more and hence utilize code segment, after running a replacement algorithm there if needed. Similar to the case of *Code replaces code*, when there is no virtual memory support, when data needs to replace code from data segment, LIFO policy would erase all the pages of code of the last brought in process.
     Hence, when data is buffered in, it replaces all the existing data from top to bottom in FIFO manner and then existing code pages from top to bottom in LIFO manner to eventually continue from the top again using FIFO to replace the data.
  4) *Code replaces data:* If the incoming code needs to replace data pages of the segment due to unavailability of space, the pages which were brought in most recently will be the ones replaced. Although this seems counter-intuitive, but it becomes obvious once we enforce the condition that a code page can replace data page of data segment only when the data page is stale. Hence, being able to replace the most recent data page by a code page implies that all the data in data segment is stale enough to be replaced by incoming code. In case the most recent data page is not stale enough, the code segment can not replace data pages there allowing reuse of the data, and it should replace code pages either in the data segment or the code segment.

- **Keeping data in code segment:** Keeping read only data in NOR flash's code segment provides second level of flexibility in the design. Data can be allowed to reside in code segment if the code segment has sufficient occupiable space to accommodate data, and when the

the data being buffered has size larger than that of data segment of NOR flash. The reasoning of former condition is intuitive, as sufficient occupiable space, meaning the total of empty space and the space from applications in Closed list which can be safely removed, is necessary to avoid unnecessary replacing of code from the code segment itself. The latter condition follows from the fact that we wish to support Replay of a video without rebuffering. In case the requirement is met, that is the video size is larger than data segment size of NOR, yet data is not kept in code segment, then this would result in replacement of complete data segment everytime the video is replayed leading to failure of ! the scheme. So if video's size is more than the size of data segment, keep as much data in data segment as possible and then buffer remaining part of it in the code segment. Care needs to be taken not to fill entire code segment with data by forcing an upper limit to the amount of data which can be placed there.

The data buffered in the code segment should be immediate victim of replacement if some other data is buffered following this one (which means that there is no more an option of Replay with the user). Hence data in code segment should be replaced even before the lowest priority closed application's pages. If the video being buffered is smaller than data segment then data segment can itself handle the Replay request.

VI. Simulation Results

For evaluation of the proposed system having main memory augmented with NOR flash, we simulated the memory components of hand-held devices and memory management algorithms, and carried out a set of experiments which indicate increased performance of the proposed system. Two different systems were simulated- the system with RAM as the only main memory component, and system with both RAM and NOR as parts of main memory, of the same cost as the previous one. The former system is assumed to have memory allocated according to the first-fit algorithm mentioned above, and swapping carried out according to simple LRU algorithm with the help of a linked list. The latter system on the other hand performs allocation according to circular-first-fit algorithm while also considering erase-count of NOR blocks, and swapping according to the proposed algorithm mentioned earlier, involving a Closed List and a Waiting List.

Based on current market costs, we use a RAM only system with 1 GB of SDRAM, and a system augmented with NOR flash to have 512 MB SDRAM and 8 GB NOR flash . The swap space is taken to be NAND flash because it supports large sequential memory accesses, which due to low cost of about $1 per GB, is assumed to be able to accommodate all the processes. The purpose of the experiments is to test the support of two systems for a large number of smartphone applications, of a very large size, which are expected to flood the smartphone market in the near future. So, for the experiments we assume upto 30 processes and test the systems for changes in performance as we change system parameters like average process size and average process arrival rate.

Reducing the time to switch between applications is crucial for satisfaction of smartphone users, which is expected to increase with the advent of a large number of larger applications. Hence the performance parameter of our interest for the experiments is- the average number of times a process already existing in main memory is started. We term this performance parameter as '*Useful*' because it describes the usefulness of a system. The experiments observe the change in the value of *Useful* of the two systems as we vary the average process size and process inter-arrival time. Also, for the system having main memory augmented with NOR flash, we test the changes in the value of *Useful* as the part of process's contents occupied in RAM (which we assume is the writeable or editable part of process) increases and that occupied in NOR (which is the read-only part of process, mainly code section) decreases.

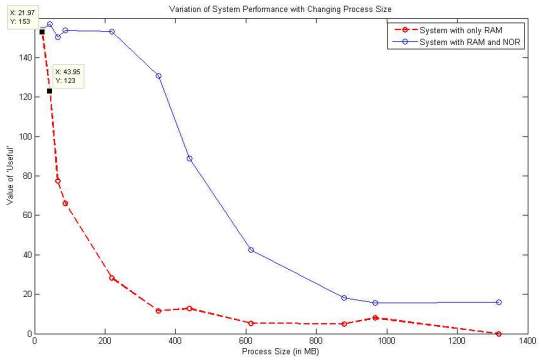


Fig. 1. Variation of performance of the two systems with changing average process size

The figure 1 shows that for small process sizes the performance of the two systems is similar. But as process size increases beyond 21MB, main memory augmented with NOR flash shows considerable increase in performance over a system with only RAM. At average process size of around 200MB, the *Useful* value of system with NOR is about 6 times more than that of system with only RAM. The performance of the two systems comes closer as the average process size grows very large, tending to a gigabyte of process. At around 1300 MB of process size, the system with only RAM is no longer able to accommodate process in the main memory without virtual memory support, whereas system with NOR flash still reports value of *Useful* to be around 18. Hence, with the advent of more number of large smartphone applications, the proposed system is expected to give increased performance and enhanced user satisfaction, which is dependent largely on switching between the applications on the device.

The next set of experiments observe the changes in performance parameters as the inter-arrival time of processes increases. These were performed for different average process sizes, and results for all process sizes are similar. Figure 2 shows the decrease in value of *Useful* as the inter-arrival time increases, for an average process size of 220 MB. Since the time for which simulation of systems was carried out remained constant for the experiment, the decrease in *Useful*'s value with increasing average arrival time is intuitive, as less often

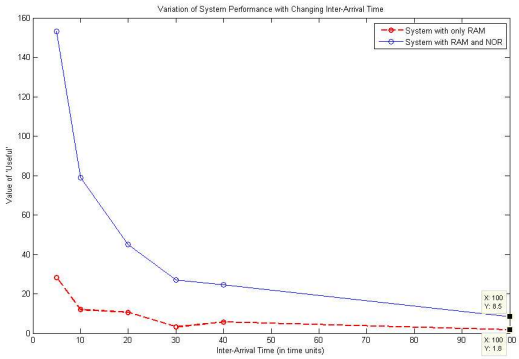


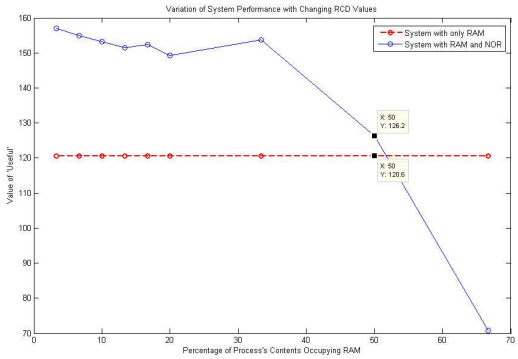
Fig. 2. Variation of performance of the two systems with changing average inter-arrival time

arriving processes would find code existing in main memory less number of times in a fixed time frame. The observation worth noting from the figure is that changing average inter-arrival time does not degrade the performance of the system with NOR as compared to the system with only RAM. The system with the two memory components performs better than the system with only RAM irrespective of the inter-arrival time.

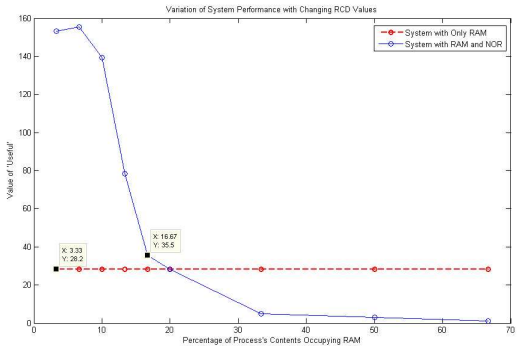
For the preceding experiments, the ratio of process's contents going to NOR and RAM was assumed to be constant-29:1. We term this ratio as *rcd*, that is ratio of code and data, as we propose to keep read-only part of process (mostly code) in NOR and the remaining contents in RAM. Since the assumed value need not be close to the actual rcd of processes, we experiment by changing the rcd values. To remove this assumption of rcd=29:1, and to observe the effect of changing rcd on value of *Useful*, we carried out the experiment of varying the percentage of process's contents going to RAM.

As can be seen from figure 3, increasing the amount of process's contents going to RAM decreases the value of *Useful*. Beyond a certain value of 'percentage of process's contents occupying RAM', the performance of the system with NOR performs worse than system with only RAM, whose performance is shown by a horizontal line in the graphs (because in case of the system with only RAM, all the contents of processes occupy RAM). So, as the *rcd* value becomes more in favour of data, the advantages of the proposed system go down. Further, comparison of figures 3(a), 3(b) and 3(c) indicates that this decline in advantages of the proposed system with increased data section (compared to code section) is more pronounced as the average process size increases.

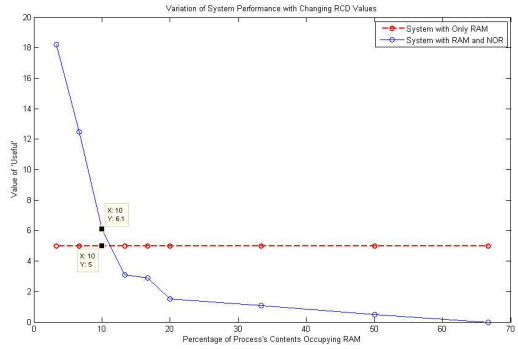
Since access time of RAM is less than that of NOR flash, increasing the percentage of contents of process occupying RAM is expected to decrease the execution time, but at the same time, it decreases the *Useful* value. Hence the amount of process's contents occupying RAM and that occupying NOR flash can be used as a trade-off between the two crucial performance parameters, opposing in this scenario- execution time and number of swaps in the system. In such a case, enhanced execution time decreases the availability of processes in main memory and increasing the availability will increase the execution time.



(a) 44 MB Process Size



(b) 220 MB Process Size



(c) 880 MB Process Size

Fig. 3. Variation of performance of system augmented with NOR flash with changing rcd values

VII. Conclusion and Future Work

In the report we propose that smartphones can have main memory which is a hybrid of volatile RAM and non-volatile NOR flash, and we present memory management algorithms for such a hybrid memory. For enhanced performance while data buffering, when running applications like YouTube, the read-only buffered data can also be accommodated in NOR flash, which mainly consists of code sections of processes. Hence we propose to have two sections of NOR flash memory space- data section and code section, which are dynamically adjustable according to amount of data and code available.

We simulated the two systems in Matlab for comparison of performance in various scenarios. It has been observed that

the system with NOR flash, due to an increased size of main memory, gives better performance in terms of availability of code of processes in memory when started next time. This is true mainly for processes of size more than about 20 MB, less than which there is not an observable performance enhancement by introduction of flash in main memory. It is also observed that even after changing the arrival rates the system with NOR flash gives better performance than that with only RAM. Hence, for a large values of system parameters, the proposed system augmented with NOR flash gives a better performance than system with RAM as the only main memory component.

Further work can be done along these lines by experimenting to observe the execution time of processes in the two systems, and finding out if the decrease in system performance with respect to execution time can be compensated by increasing the size of instruction cache. Also, the performance of the proposed memory management algorithms can be compared with that of other possible algorithms.

REFERENCES

[1] J. C. Mogul, E. Argollo, M. Shah, P. Faraboschi. *" Operating System Support for NVM+DRAM Hybrid Main Memory"*. In Hot Topics in Operating Systems (HotOS) (2009).

[2] Mohit Saxena, Michael M. Swift. *" FlashVM: Revisiting the Virtual Memory Hierarchy"*. In HotOS-XII (2009).

[3] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z. Morley Mao, Ming Zhang, Paramvir Bahl. *"Anatomizing Application Performance Differences on Smartphones"*. MobiSys10, June 1518, 2010, San Francisco, California, USA.

[4] Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim, Bumsoo Kim. *"Cost-Efficient Memory Architecture Design of NAND Flash Memory Embedded Systems"*. 21st International Conference on Computer Design (ICCD03).

[5] Earl Oliver. *"A Survey of Platforms for Mobile Networks Research"*. Mobile Computing and Communications Review, Volume 12, Number 4.

[6] Jared Hulbert, Justin Treon. *"Creating optimized XIP systems"*. CELF, ELC, 2006.

[7] Christian Forsberg. *"Effective Memory, Storage, and Power Management in Windows Mobile 5.0"*. http://msdn.microsoft.com .

[8] Dipert, Hebert. *"Flash Memory Goes Mainstream"*. IEEE Spectrum, October 1993.

[9] Hyojun Kim, Jihyun In, DongHoon Ham, SongHo Yoon, Dongkun Shin. *"Virtual-ROM: A New Demand Paging Component for RTOS and NAND Flash Memory Based Mobile Devices"*. in ISCIS, vol. 4263. Berlin, Germany: Springer-Verlag, 2006, pp. 677686.

[10] Website: *"http://www.samsung.com/global/business/semiconductor/products"*.

[11] Hyung Gyu Lee, Naehyuck Chang. *"Energy-Aware Memory Allocation in Heterogeneous Non-Volatile Memory Systems"*. ISLPED03, August 2527, 2003, Seoul, Korea.

[12] Chanik Park, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim. *"Energy-Aware Demand Paging on NAND Flash-based Embedded Storages"*. ISLPED04, August 911, 2004, Newport Beach, California, USA.

[13] Chanik Park, Jaeyu Seo, Sunghwan Bae, Hyojun Kim, Shinhan Kim, Bumsoo Kim. *"A Low-cost Memory Architecture with NAND XIP for Mobile Embedded Systems"*. CODES+ISSS03, October 1-3, 2003, Newport Beach, California, USA.

[14] M. Rebaudengo, M. Sonza Reorda, M.Violante. *"An Accurate Analysis of the Effects of Soft Errors in the Instruction and Data Caches of a Pipelined Microprocessor"*. Design, Automation and Test in Europe Conference and Exhibition, 2003.