International Journal of Advanced Smart Convergence Vol.8 No.1 9-17 (2019)
http://dx.doi.org/10.7236/IJASC.2019.8.1.9

IJASC 19-1-2

Application-Adaptive Performance Improvement in Mobile Systems by Using Persistent Memory

Hyokyung Bahn *

Department of Computer Engineering, Ewha University, Korea
bahn@ewha.ac.kr

Abstract

In this article, we present a performance enhancement scheme for mobile applications by adopting persistent memory. The proposed scheme supports the deadline guarantee of real-time applications like a video player, and also provides reasonable performances for non-real-time applications. To do so, we analyze the program execution path of mobile software platforms and find two sources of unpredictable time delays that make the deadline-guarantee of real-time applications difficult. The first is the irregular activation of garbage collection in flash storage and the second is the blocking and time-slice based scheduling used in mobile platforms. We resolve these two issues by adopting high performance persistent memory as the storage of real-time applications. By maintaining real-time applications and their data in persistent memory, I/O latency can become predictable because persistent memory does not need garbage collection. Also, we present a new scheduler that exclusively allocates a processor core to a real-time application. Although processor cycles can be wasted while a real-time application performs I/O, we depict that the processor utilization is not degraded significantly due to the acceleration of I/O by adopting persistent memory. Simulation experiments show that the proposed scheme improves the deadline misses of real-time applications by 90% in comparison with the legacy I/O scheme used in mobile systems.

Keywords: *Real-time application; Mobile application; Smartphone; Persistent memory.*

1. Introduction

With the wide diffusion of mobile platforms as well as the rapid improvement of smartphone hardware technologies, application domains covered by smartphones are becoming increasingly wider. In particular, not only non-real-time applications like web browsers, but also real-time applications like video players are supported by a smartphone. Nevertheless, current smartphone platforms do not support these heterogeneous applications efficiently. For example, Android is a well-known software platform for smartphones, but it is developed on the Linux kernel, which is not a real-time operating system. Fig. 1 depicts the basic structure of

Manuscript Received: Jan. 3, 2019 / Revised: Jan. 11, 2019 / Accepted: Jan. 18, 2019
Corresponding Author: bahn@ewha.ac.kr
Tel: +82-2-3277-2368, Fax: +82-2-3277-2306
Department of Computer Engineering, Ewha University, Korea

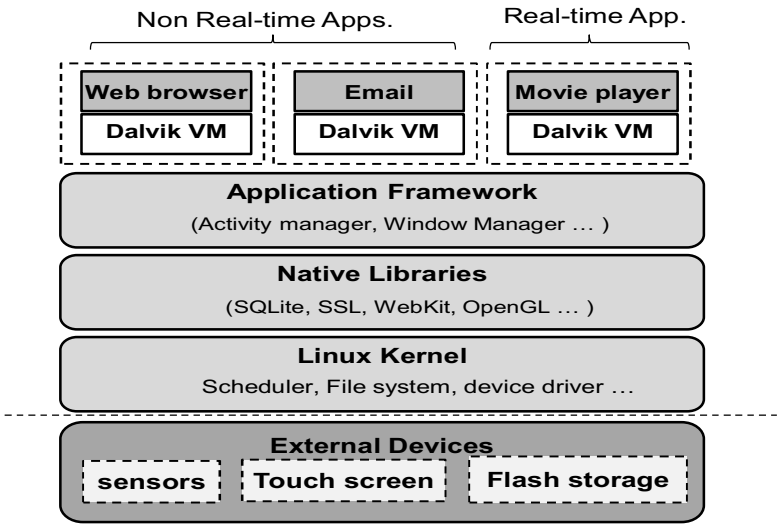


Figure 1. Basic structure of Android mobile software platform and applications supported by it.

an Android mobile software platform. As Linux is a general-purpose operating system, which has been developed for managing time-sharing applications, it does not guarantee the deadline of real-time applications. In particular, it is difficult to guarantee the deadline of real-time applications due to the unpredictable latency in the I/O path of Linux. Under the Linux and Android systems, a process is blocked when it requests an I/O and it wakes up when the requested I/O is completed. After waking up, however, even a high priority process like real-time applications cannot run in a processor immediately as current I/O schedulers like CFS (completely fair scheduler) do not support preemption in processor scheduling. Due to this reason, even a real-time application should wait in the processor queue until the remaining time slice of the process currently running in the processor expires, making it difficult to guarantee the deadlines of real-time applications [1].

Flash storage is another source of unpredictable latency in the I/O path of mobile systems. Due to its good features like energy efficiency, shock resistance, and small size, flash storage is commonly adopted in the storage system of smartphones. However, flash storage has some critical weaknesses to be a storage system of real-time applications. Specifically, flash needs an erase operation before re-writing to the same location, which necessitates the garbage collection procedure. Note that garbage collection makes free space by moving scattered valid data to a certain area and erasing them [2]. This garbage collection procedure is not activated periodically and it needs an order of magnitude longer latency than regular read/write operations, causing unpredictable I/O latency. Moreover, the duration of garbage collection significantly varies depending on the internal state of flash storage, resulting in fluctuation of I/O latency.

In this article, we aim at removing the unpredictable sources of latency in order to guarantee the deadline of real-time applications by two novel enhancements. The first is that we use persistent memory as the supplementary storage media of real-time applications. Persistent memory such as PC-RAM (phase-change RAM) or STT-MRAM (spin torque transfer magnetic RAM) is recently manufactured and can be adopted as an auxiliary component of mobile devices in order to improve system performances [3]. Unlike flash storage, persistent memory ensures predictable I/O latency for real-time applications as it does not need garbage collection and its performance is faster than flash storage. Although we adopt persistent memory as the storage of real-time applications, unpredictability can still occur due to the blocking I/O mechanism and the scheduler adopted in current Android and Linux systems. To alleviate this problem, this article presents a

new processor scheduler, which assigns one processor core to a real-time application, if there is a pending real-time application ready to run. Furthermore, our scheme performs I/O requests of a real-time application without blocking. As this allows the real-time application to have a processor control even during its I/O execution, it can perform its execution in processor core right after the completion of I/O.

This non-blocking I/O has a weakness in that processor cycles can be wasted during the I/O execution time of real-time applications. Apparently, this is not efficient in low-end storage devices like HDD, in which a storage access needs millions of processor cycles. Unlike this situation, however, persistent memory is several orders of magnitude faster than HDD, and thus non-blocking I/O does not degrade the utilization of processors significantly. Moreover, as a processor core for real-time applications can be free if real-time applications are not active, degradation of the processor utilization is not significant. The efficiency of the proposed scheme is assessed via trace-driven simulation studies. Our evaluation results depict that the proposed scheme improves the deadline misses of real-time applications by 90% on average in comparison with the I/O scheme currently adopted in Android platform.

The remainder of this article is organized as follows. Section 2 explains a mobile software architecture and presents the proposed algorithms therein. Section 3 presents the performance evaluation results to assess the effectiveness of the proposed algorithms. Finally, Section 4 concludes this article.

2. Application-Adaptive Utilization of Persistent Memory

2.1 Mobile Software Platform and I/O Mechanism

The Android mobile software platform is composed of 4 main components; an application framework, run-time libraries like a Dalvik machine, native libraries, and the Linux kernel. Fig. 1 depicts the basic structure of Android. The application framework consists of a set of system servers, which allows applications to use Android services. The native libraries consist of a set of libraries written in C/C++ like the standard C library, graphic engines and multimedia codecs, which are executed between Java APIs and the Linux kernel. Whenever an application issues a storage I/O request, it calls a specific framework API to access the storage. The API then delivers the I/O request to kernel via a system call, and the kernel forwards the I/O request to storage via device drivers and controllers. After that, kernel blocks the application that has invoked the I/O request and passes the processor control to other processes during the I/O execution. Once the I/O is finished, the device controller informs it to the kernel through an interrupt. Then, kernel delivers the requested data to the buffer space and wakes up the blocked process.

Although the aforementioned I/O mechanisms are effective with respect to the processor utilization, it does not guarantee the deadline of real-time applications because of the unpredictable latencies in the I/O path. In particular, the completely fair scheduler used in Android does not perform preemptive scheduling. That is, processors cannot be preempted until the time slice of the process currently being executed in a processor core expires [1]. Due to this reason, although a real-time application finishes its I/O operation, it should wait for the remaining time slice of the process currently being executed in the processor. The time slice is usually set to one millisecond, which is very long when compared to the access latency of persistent memory, tens of nanoseconds [4]. Moreover, as this latency can be varied depending on the remaining time slice of the current process, unpredictable delays can happen to real-time applications.

Flash storage also provides unpredictable delay due to garbage collection. In particular, when free space in flash storage becomes small, garbage collection is activated, which converts obsolete spaces to free spaces by erasing them. This garbage collection procedure makes the prediction of I/O time even more difficult as it can be triggered sporadically and takes long time because of the large number of erase and copy operations. As Table 1 depicts, an erase operation takes an order of magnitude longer time in comparison with read/write

operations.

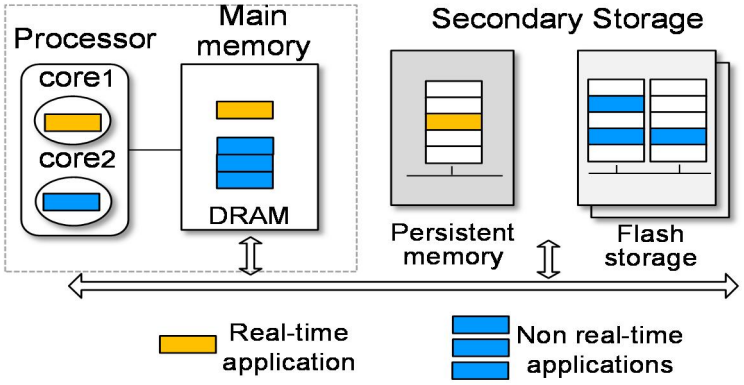
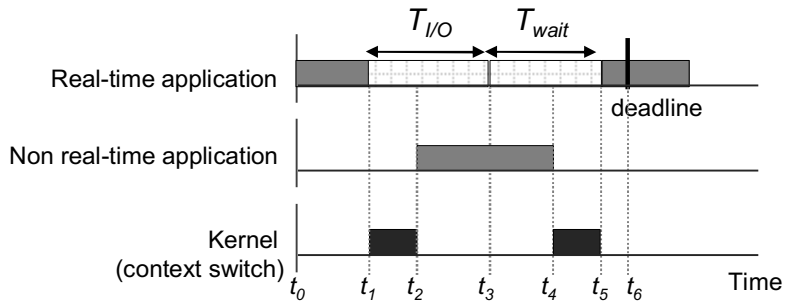


Figure 2. System structure with persistent memory.

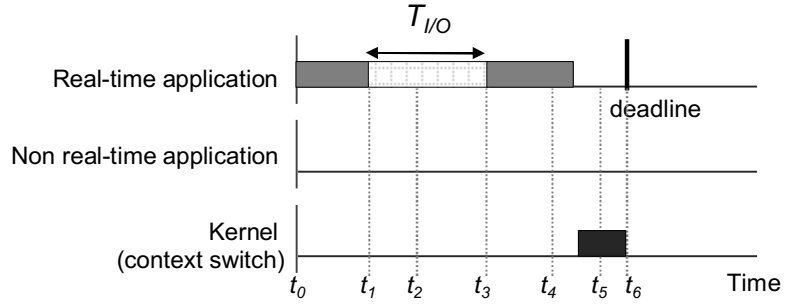
2.2 The Proposed Scheme

This article removes all sources of unpredictable latency while executing real-time applications by utilizing persistent memory and a novel scheduling scheme. Specifically, our scheme makes use of persistent memory as the storage of real-time applications and their data. Unlike flash storage, persistent memory does not need garbage collection but provides uniform access time [5], and hence it can be used for the storage of deadline-guaranteed real-time applications. Fig. 2 depicts the memory and storage structures of the proposed scheme. As shown in the figure, persistent memory and flash storage form hybrid storage media. Persistent memory is utilized as a dedicated storage for real-time applications and their data, while all the other applications and data are stored in flash storage. This structure removes unpredictable I/O latency of flash storage for real-time applications, and also reduces the I/O time incurred due to the bus contention. The processor part of our system is composed of multiple cores, of which one core is exclusively allocated for a real-time application. Nevertheless, when no real-time application is active in the system, the reserved core can be allocated to other applications. In our architecture, the first level cache memory is private to each processor core and the second level cache memory is shared across different cores.

Main memory consists of DRAM like conventional system architectures as shown in the figure. As persistent memory is byte-addressable and has good performances in comparison with DRAM, there is a bright prospect that persistent memory can be used as main memory as well as storage, leading to a unified memory architecture [6]. When considering this prospect, a real-time application can be executed in persistent memory without moving to DRAM, allowing more accurate prediction of execution time in real-time applications. However, this is a challenging architecture and the operating system itself should be designed again to support such functionalities. Due to this reason, our scheme assumes the persistent memory as storage only. Note that our architecture can also relieve the scheduling latency caused by the blocking I/O. The rationale behind the blocking I/O is to improve the utilization of processor by executing other processes when I/Os are performed due to the large I/O latency in conventional storage systems. However, blocking I/O cannot allow for the prediction of I/O time for real-time applications due to the large and unpredictable I/O latencies caused by reordering and merging while I/O scheduling. When persistent memory is used as its underlying storage, however, non-blocking I/O can be considered as the I/O time is very small and predictable. Thus, our scheduling scheme performs the non-blocking I/O for real-time applications, which enables the instant execution of the real-time application when it returns to the processor queue after the completion of I/O, removing long latency caused by blocking and waking up. Although a certain utilization of processor's time can be degraded when adopting non-blocking I/O, it is not large



(a) Process scheduling with blocking I/O



(b) Process scheduling with non-blocking I/O

Figure 3. Process scheduling with blocking and non-blocking I/Os.

because the access latency of persistent memory is very fast compared to other storage media like flash storage or HDD.

Figs. 3(a) and 3(b) depict the execution path of applications when blocking and non-blocking I/Os are adopted. In this example, we assume that the real-time application begins its execution at time t0, and requests an I/O at t1. In case of blocking I/O, the real-time application is blocked upon the I/O request time and a context-switch happens to allocate the processor core to other applications. Although the I/O is completed at t3, the real-time application cannot use the processor core immediately and needs to wait until t4 when the other application completes its execution due to a timer interrupt or a voluntary return. Then, a context-switch happens again and the real-time application resumes its execution at t5. The problem in this process is that the real-time application does not satisfy its deadline as the waiting time and the context-switch overhead makes unacceptable delays.

On the contrary, as depicted in Fig. 3(b), non-blocking I/O allows the real-time application to hold the processor core even when it is performing storage I/O. As a result, the real-time application returns to the processor core at time t3, right after the I/O is completed. Since this non-blocking I/O causes neither scheduling delay nor context-switch overhead, the real-time application can finish its execution earlier than its deadline. The processor core is wasted while the real-time application performs I/O, that is t1 to t3, but the saved context-switch time can compensate this as persistent memory is fast enough. Moreover, as the core allocated to the real-time application can be utilized by other applications when it becomes inactive, degraded utilization of processors due to non-blocking I/O is not large.

3. Performance Evaluations

To evaluate the performance of the proposed scheme, trace-driven simulations have been performed. Our experimental platform is ODROID A4, a kind of Android reference platform. I/O traces were collected by a system profiler developed on Linux and a video player, which plays 24 frames per second, is used as the real-time application. In each period, a video player reads data from a video file, decodes them, and then displays frames. Once the mission of the current period is completed, it sleeps until the next period begins. We developed a simulator to evaluate the completion time of the movie player for each period. We set the context-switch time of this experiment to 15 us [7]. We assume PC-RAM as the persistent memory of our architecture as it has already been manufactured by some persistent memory vendors [3]. The flash storage and DRAM access latency is set as shown in Table 1.

Fig. 4 depicts the deadline miss ratio of the real-time application for the blocking I/O with flash storage and the non-blocking I/O with our scheme. Blocking I/O always performs context-switch when an I/O

Table 1. Characteristics of memory and storage devices

	DRAM	PC-RAM	Flash	HDD
Read latency	50ns	50-100ns	60us	10ms
Write latency	50ns	400-500ns	800us	10ms
Ease latency	N/A	N/A	1.5ms	N/A

request arrives, while in our scheme the real-time application keeps the processor context even when I/O is being processed. We varied the cache capacity widely from 10% to 100% to evaluate the effectiveness of our scheme. When the cache capacity is small, the real-time application incurs more I/O requests. As depicted in Fig. 4, our scheme improves the deadline misses by 90% on average in comparison with the blocking I/O. In particular, the proposed scheme keeps the 0 percent of deadline misses if the cache capacity is 80%, whereas the blocking I/O incurs 50% of deadline misses with the same cache capacity. The reason is that our scheme removes the unpredictable delays of processor scheduling by using non-blocking I/O in case of real-time applications. The saved context-switch time and fast access latency of persistent memory also contribute to the enhancement of real-time application's deadline misses.

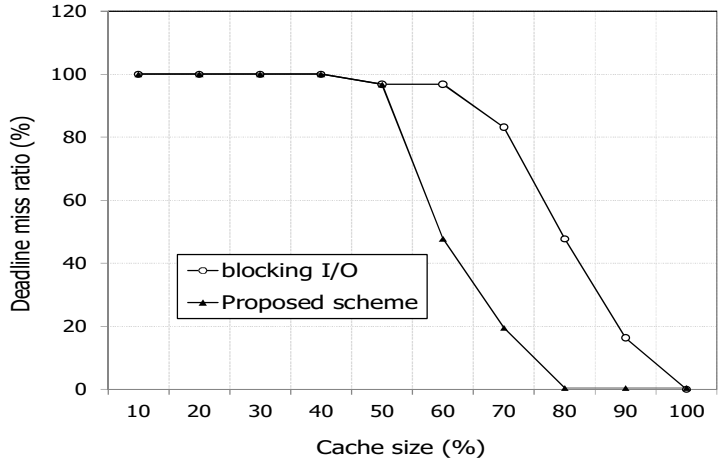


Figure 4. Deadline misses as the cache capacity is varied.

Fig. 5 depicts the deadline misses of the real-time applications as the waiting time in the processor queue is varied. In this experiment, the waiting time of one hundred percent implies the case that a process in its execution uses the core for its full time slice when the real-time application returns to the processor queue. Note that the time slice is set to one millisecond in our experiments by following the conventional setting [4]. As depicted in the figure, the deadline misses of blocking I/O degrades seriously as the waiting time in the processor queue becomes longer, while our scheme exhibits consistently good results regardless of the waiting time. In case of blocking I/O, the deadline misses becomes 84% even when the waiting time in the processor queue is only 20%. Based on these observations, we can conclude that prohibiting interferences among other applications will be efficient in order to guarantee the deadline of real-time applications, specially when the number of applications executed simultaneously becomes large.

Fig. 6 depicts the completion time of the real-time application in each period as time goes on. As shown in the figure, our scheme met the deadlines of all periods and completed the real-time application 60% earlier than its deadline on average. In case of blocking I/O, 78 times of deadline misses happen while playing 150 frames and delayed the completion time by 22% on average. One weakness of our scheme is that the core allocated to real-time applications can be wasted, degrading the performance of other applications. That is, as the real-time application does not release its core while it performs I/O, the utilization of processors can be degraded. To see the effect of this, the utilization of processors is monitored when our scheme is adopted. For a comparison purpose, we also observed the utilization of processors when our scheme is used with flash storage and HDD rather than persistent memory. Table 1 lists the detailed performance features of storage media used in our experiments. As depicted in Fig. 7, the utilization of processors with our scheme is degraded by only 4% in comparison with blocking I/O when persistent memory is adopted. On the contrary, our scheme degrades the utilization of processors by 38% and 49% in flash storage and HDD, respectively. When using slow storage media, I/O processing needs more processor cycles, and thus holding a core while processing I/O causes significant performance degradations. In contrast, with persistent memory, the wasted processor cycles are not significant and can also be compensated by the saved context-switch overhead. As a result, the proposed scheme supports the deadline guarantee of a real-time applications, while providing reasonable performances for non-real-time applications.

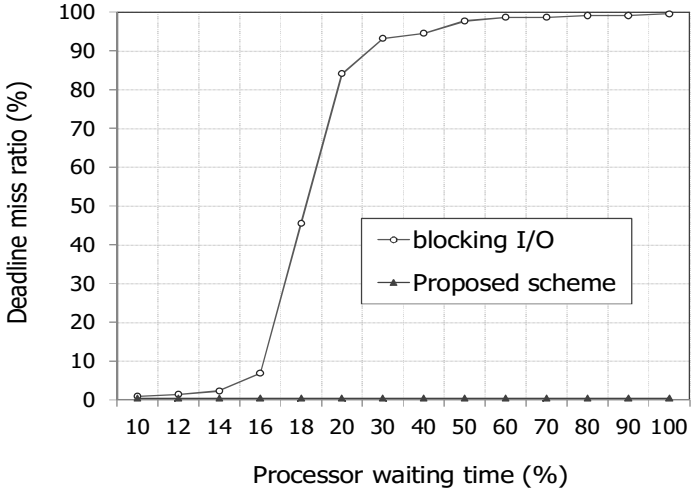


Figure 5. Deadline misses as the waiting time in processor queue is varied.

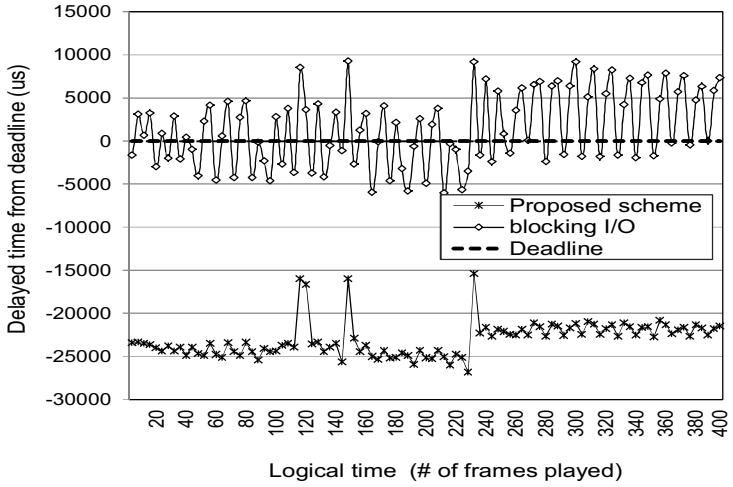


Figure 6. Completion time of the real-time application as time goes on.

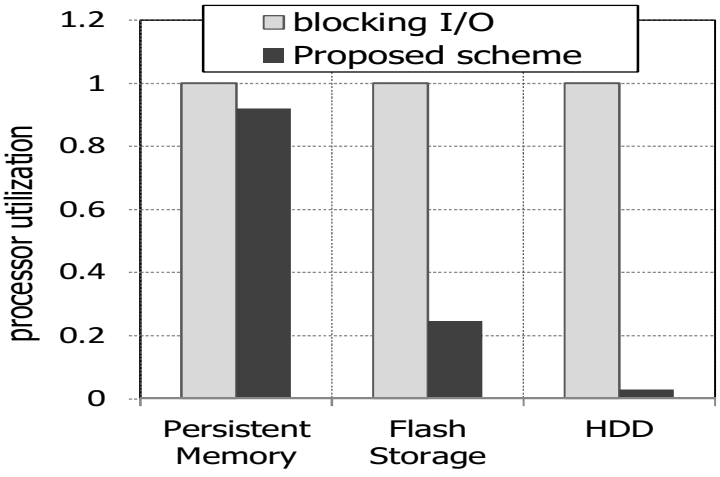


Figure 7. Utilization of the processor as the storage device is varied.

4. Conclusion

We proposed an application-adaptive performance improvement of mobile applications by adopting persistent memory. To relieve the unpredictable delay caused by the I/O path in mobile systems, we use persistent memory as the storage of real-time applications, and perform non-blocking I/O. Simulation experiments with real traces showed that the proposed scheme improves the deadline misses of real-time applications by 90% on average. Nevertheless, the performance degradation of non-real-time applications was negligible.

Acknowledgement

A subset of this article was presented at the 5th Int'l Joint Conference on Convergence (IJCC 2019), Jan. 22-26, 2019, Taipei. This work was supported by the Basic Science Research program through the National

Research Foundation (NRF) grant funded by the Korea government (MSIP) (No. 2016R1A2B4015750).

References

[1] S. Wang Y. Chen, W. Jiang, P. Li, T. Dai, and Y. Cui, "Fairness and interactivity of three CPU schedulers in Linux," in *Proc. IEEE RTCSA*, 2009.

[2] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho, "A Space efficient Flash Translation Layer for CompactFlash Systems," *IEEE Trans. Consumer Electron.*, Vol. 48, No. 2, pp. 366-375, 2002.
DOI: https://doi.org/10.1109/TCE.2002.1010143

[3] E. Lee S. Yoo, J. Jang, and H. Bahn, "WIPS: a write-in-place snapshot file system for storage-class memory," *Electronics Letters*, Vol.48, No.17, pp. 1053–1054, 2012.
DOI: https://doi.org/10.1049/el.2012.1016

[4] B. Brandenburg, J. Calandrino, and J. Anderson, "On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study," in *Proc. IEEE RTSS*, 2008.

[5] S. Lee, H. Bahn, and S. Noh, "Characterizing Memory Write References for Efficient Management of PCM and DRAM Memory," in *Proc. IEEE MASCOTS*, 2011.

[6] X. Wu and A. Reddy, "SCMFS: a file system for storage class memory," in *Proc. IEEE Conference on Supercomputing (SC)*, 2011.

[7] H. Choi and H. Yun, "Context Switching and IPC Performance Comparison between uClinux and Linux on the ARM9 based Processor," in *Proc. SAMSUNG Tech. Conference*, 2005.