Revisiting Read-ahead Efficiency for Raw NAND Flash Storage in Embedded Linux

Pierre Olivier
Univ. Europeenne de Bretagne
Univ. Bretagne Occidentale
UMR6585 Lab-STICC
F29200 Brest, France
+332 98 01 74 35
pierre.olivier@univ-brest.fr

Jalil Boukhobza
Univ. Europeenne de Bretagne
Univ. Bretagne Occidentale
UMR6585 Lab-STICC
F29200 Brest, France
+332 98 01 69 73
jalil.boukhobza@univ-brest.fr

Eric Senn
Univ. Europeenne de Bretagne
Univ. Bretagne Sud
UMR6585 Lab-STICC
F56100 Lorient, France
+332 97 87 46 03
eric.senn@univ-ubs.fr

ABSTRACT
The Linux Read-Ahead mechanism has been designed to bridge the gap between the secondary storage low performance and I/O read-intensive applications for personal computers and servers. This paper revisits the efficiency of this mechanism for embedded Linux using flash memory as secondary storage, which is the case for most embedded systems. Indeed, Linux kernel uses the same read-ahead mechanism whatever the application domain. This paper evaluates the efficiency of read-ahead technique for the widely used flash specific file systems that are JFFS2 and YAFFS2, in terms of response time and energy consumption. We used micro-benchmarks to investigate read-ahead effect on those metrics at a fine (system call) granularity. Moreover, we also study this impact at a higher application level using a macro-benchmark evaluating read-ahead effect on the SQLite DBMS read performance and power consumption. As described in this paper, disabling this mechanism can improve the performance and energy consumption by up to 70% for sequential patterns and up to 60% for random patterns.

Categories and Subject Descriptors
D.4.2 [**Operating Systems**]: Storage Management – s*econdary storage, main memory, storage hierarchies*. D.4.3 [**Operating Systems**]: File Systems Management – *access methods*. D.4.8 [**Operating Systems**]: Performance – *measurements, modeling and prediction*.

General Terms
Performance, Measurement, Design.

Keywords
Embedded Linux, Flash File System, JFFS2, YAFFS2, Linux Page Cache, Read-Ahead.

1. INTRODUCTION

Embedded Linux has become the de facto operating system for many embedded applications such as consumer electronics (smartphones, tablets, etc.), multimedia devices and set-top boxes. It is also integrated in many devices such as routers, video surveillance systems, and robots. With Linux, there is no special form of kernel dedicated to embedded systems. Instead, one unique (configurable) kernel is intended to be used for the widest range of devices. One of the peculiarities of embedded Linux is the use of a special subsystem to manage bare flash memory based storage systems through Flash File Systems (FFS).

The explosion of the NAND flash memory market has boosted many embedded system applications, especially consumer electronics, by providing efficient and relatively cheap Non-Volatile Memory (NVM). In fact, mobile memory (including both NOR and NAND flash, DRAM and embedded multimedia cards) market have experienced a growth of 14% in 2012 (as compared to 2011). However, NAND flash memory presents some specific constraints one should deal with when designing an embedded system: (1) Write/Erase granularity asymmetry: writes are performed on pages whereas erase operations are executed on blocks, a block being composed of pages. (2) Erase-before-write rule: one cannot modify data in-place. A costly erase operation must be achieved before data can be modified in case one needs to update data on the same location. (3) Limited number of write /erase (w/e) cycles: the average number is between 5000 and 10^5 depending on the used flash memory technology. After the maximum number of erase cycles is achieved, a given memory cell becomes unusable. Finally, (4) the I/O performance for read and write operations is asymmetric.

There are two possible ways to deal with the aforementioned flash memory constraints: (1) through a hardware/software controller included into the memory device itself named the Flash Translation Layer (FTL). This is the case for USB sticks, flash cards, solid state drives, etc; or (2) through some specific Flash File System (FFS) implemented at the embedded operating system level in a pure software solution. JFFS2 [1], YAFFS2 [2], and UBIFS [3] are the most popular FFS. All of them rely on a deep layer in the kernel that interfaces the file system with the flash memory driver: the Memory Technology Device (MTD). This software stack behaves differently from traditional block devices and file systems. Resulting interactions with the upper kernel file management layers are thus very different. In these layers is implemented the Linux read-ahead algorithm, which aims to prefetch data read from secondary storage to enhance a process I/O read performance.

EWiLi'14, November 2014, Lisbon, Portugal.

Copyright retained by the authors.

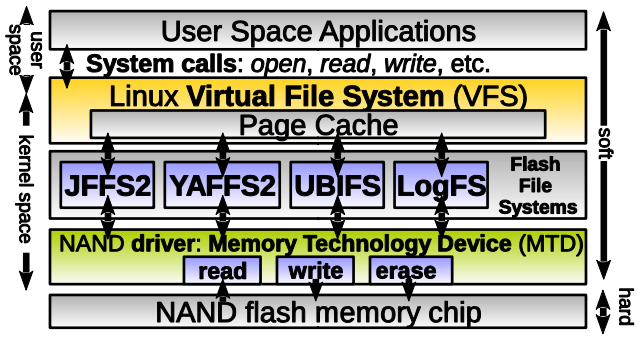

Figure 1. Flash-based storage software stack in Linux

This study analyzes the interaction between the flash based storage system on embedded Linux and the page cache read-ahead prefetching system. Conducted experimentations showed that a substantial performance drop is observed for both sequential and random I/O workloads. This leads to subsequent energy consumption overhead on both CPU and memory subsystem (RAM and flash memory). This paper tries to quantify this behavior.

The paper is organized as follows: the first section gives some background on flash memories. Section 3 describes the performance evaluation methodology. Section 4 discusses the obtained results. Section 5 gives some conclusions.

2. PRELIMINARIES

2.1 Background

To cope with the above-mentioned NAND flash memory constraints, some specific management mechanisms are implemented. (1) In order to avoid a costly in-place data update, a logical-to-physical address mapping mechanism allowing to perform *out-of-place* data modifications is used (update data in another location and invalidate the first copy). (2) As the number of write/erase (w/e) cycles is limited and because of spatial and temporal data locality of I/O workloads, some specific blocks containing "hot" data can wear out quickly. To avoid this issue, wear leveling mechanisms are implemented all with the mapping system in order to evenly distribute the erase operations over the whole memory surface. (3) Performing many update operations results in many invalidated pages/blocks that must be erased in order to be reused. A garbage collector is used to perform this task.

Figure 1 illustrates the FFS layer location inside the Linux storage hierarchy. User space processes access files using system calls, received by the Virtual File System (VFS). VFS role is to abstract the idiosyncrasies of the underlying file systems. Moreover, at the VFS level, Linux maintains several caches in RAM in order to speed up file operations. In particular, Linux page cache is dedicated to file data buffering. VFS maps system calls to FFS functions. To access the flash chip, the FFS uses a NAND driver, MTD.

Hard drives exhibit poor performance on random I/O requests. It is due to the presence of mechanical elements generating important latencies. Linux read-ahead [4] mechanism was designed to alleviate that problem. It allows to sequentially prefetch more data from the disk than it is requested by a process accessing a file. The size of prefetched data is based on complex heuristics. Their aim is to determine if the global read pattern of the process is sequential or random. On sequential access patterns, read-ahead prefetches large chunks of data. On random ones, it reads smaller amounts. Read-ahead is implemented at the VFS level: it is independent from the underlying file system. This mechanism enhances I/O performance on disk based storage systems because [4]: (1) it reduces mechanical movement by reading large data chunks; (2) it relies on I/O timeouts to asynchronously prefetch data. So, in case of a page cache hit, data are read directly from the page cache inferring no access to the secondary storage.

2.2 Motivational Example

Because flash memory is fundamentally different from hard disk drives, one could question about the usefulness of read-ahead for flash based storage. Read-ahead is enabled by default on Linux, unless the file system itself specifies not to use it. This may be done in the source code of the file system.

As a preliminary experiment, we used the popular storage benchmark Postmark [5]. Each time a file is read during the benchmark execution, we measured the time taken by the read() system calls, then we computed the mean read throughput. The throughput measurements were performed at 3 moments during each file read operation: when 25%, 50% and 100% of the file is read. The benchmark was launched on a hardware platform (TI OMAP based board described further in this paper), running Linux with (A) read-ahead enabled and (2) read-ahead disabled. Results are presented in Table 1.

Table 1. Postmark mean read throughput (MB/s)

Percentage of files size read	FFS: JFFS2		FFS: YAFFS2	
	RA ON	RA OFF	RA ON	RA OFF
25%	2.86	6.68	6.68	14.30
50%	4.77	7.63	9.54	14.30
100%	7.63	7.63	14.30	14.30

The difference in performance between JFFS2 and YAFFS2 may be due to YAFFS2 internal caching system [2], and JFFS2's waste of time in uncompressing random data composing Postmark files. Those results show that when reading a file, read-ahead causes a dramatic performance drop if the file is not entirely read In this paper we study and analyze in details this impact on the performance and power consumption for the popular FFS JFFS2 and YAFFS2.

2.3 Related Work

Based on the fact that flash memory read operation is pattern-agnostic (no difference between sequential and random reads), read-ahead was deactivated in UBIFS. Nevertheless, no experimental evaluation was published about this issue. In [6], authors present an algorithm for flash memory management in embedded systems for which read-ahead is disabled because of performance overhead. In [7], a novel read-ahead algorithm is proposed to enhance read-ahead performance in the context of demand-paging on compressed file system (CramFS) on flash memory.

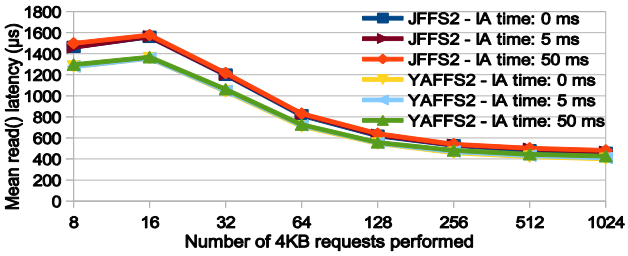

Figure 2. Inter arrival time variation

3. PERFORMANCE EVALUATION OF READ-AHEAD ON EMBEDDED LINUX

3.1 Methodology and Metrics

3.1.1 Micro-benchmarks

The performed experimentations consisted in measuring the performance and energy consumption of sequential and random I/O workloads on both JFFS2 and YAFFS2 with read-ahead enabled and disabled. We first modified JFFS2 and YAFFS2 source codes to disable read-ahead. Note that disabling this mechanism can also be done at the application level without modifying the kernel sources, by using the *posix_fadvise()* system call. This is a less intrusive but also less generic solution. We designed a simple C test program that performs read operations in a loop, on a target file stored in a JFFS2 or YAFFS2 flash memory partition. Read operations were performed with the read() system call and the target file was opened with the O_RDONLY flag. In the test program, several parameters were varied: the number of generated read requests, their size, the target file size, inter-arrival times, and the access pattern (random or sequential). Before each test, the Linux page cache was emptied to insure the same initial state.

Performance evaluation: The execution time of each read() system call was measured using the gettimeofday() system function, giving a microsecond precision. We also used Flashmon [8] to trace the number of flash I/O accesses or each test to compare the number of generated read operations with and without read-ahead mechanism.

Power consumption: In order to study how the read-ahead mechanism impacts the energy consumption, we measured during read accesses the power on both the (1) CPU and (2) RAM + flash memory power rails (RAM and flash share the same power rail on our hardware test platform). It allowed us to build a simple power consumption model to estimate read-ahead effect on the I/O energy consumption. The basic idea behind the model is to multiply the mean power measured during read accesses by the execution time of a given experimentation to obtain the energy consumption.

3.1.2 SQLite Macro-benchmark

SQLite [9] is a relational database engine operated by the SQL query language. One of its specificities is the fact that a SQLite database is fully contained in a single file on top of a regular file system. Moreover, the SQLite DBMS is available in various formats, in particular in the form of a C library which can be directly embedded in (i.e. compiled with) an application, eliminating the need for any external dependency. Because of that portability, its relatively low CPU load /memory footprint, and its tolerance to sudden system shutdown, SQLite is widely used in embedded systems. In particular, SQLite is the DBMS used for managing system and user databases in the Android embedded operating system [10].

We used for the macro-benchmark experimentation a SQLite database containing a single table. As in [10], we used a schema reproducing the contacts database of an Android operating system. The contacts table contains 17 fields, 12 of which being integers, and 5 being text fields. One of the integer fields is a unique record identifier (primary key). The database was created on previously erased JFFS2 and YAFFS2 flash partitions. The database was filled with 1000 records containing random data. We created two versions of the database, one with each one of the 5 text fields filled with 64 bytes strings, the other with 128 bytes strings. The reported size of the database files was then of 518 KB for the first version, and 1 MB for the second. In the rest of this paper we refer to the version 1 as *small database*, and *large database* for version 2.

Performance evaluation: We created a C application integrating the SQLite library and performing select operations on the database. Using the application, we performed selection runs, each one consisting of one or several record selections from the database in a loop. A run can be performed in sequential or random mode, according to the read record identifier order (record identifier being assigned incrementally during the table creation). The Linux page cache is dropped before each run and the execution time of each run is measured with *gettimeofday()*. Experiments are ran with read-ahead enabled and disabled.

Power consumption: We used an adapted version of the power consumption model created in the micro-benchmark phase to estimate read-ahead impact on energy during SQLite selections.

3.2 Hardware and Software Experimental Configuration

We used a Mistral Omap3evm board embedding an OMAP3530 (720 MHz ARM Cortex A8) CPU, 256 MB of RAM, and 256 MB of Micron SLC (Single Level Cell) NAND flash. The NAND chip datasheet reports a latency of 130 µs to read a 2048 bytes flash page (internal read operation plus transfer on the I/O bus). As stated earlier, the RAM and flash share the same power rail. The Linux kernel 2.6.37 was used with a standard embedded kernel configuration. For the power consumption measurements, we used the Open-PEOPLE (Open-Power and Energy Optimization PLatform and Estimator) platform [11][12] equipped with a National Instruments PXI-4472 module. For the macro-benchmark we used the version 3.7.15.2 of SQLite.

4. RESULTS AND DISCUSSION

4.1 Micro-benchmarks: Read system call performance

4.1.1 Impact of Inter-arrival Times: why read-ahead performs badly with FFS

Using disks, the read-ahead mechanism can be launched asynchronously during I/O timeouts to optimize I/O response times. In contrast, the Linux NAND driver (MTD) used by FFS is a fully synchronous software stack. We tested this feature by replaying the same experimentations, sequentially reading part of a 5MB file, and inserting inter-arrival times between reads.

Figure 2 shows the mean read latency when varying inter-arrival times with read-ahead enabled under sequential workload. Response times are not impacted by the inter-arrival time increase, which confirms the synchronous nature of flash operations. In fact, when read-ahead is enabled, the prefetching

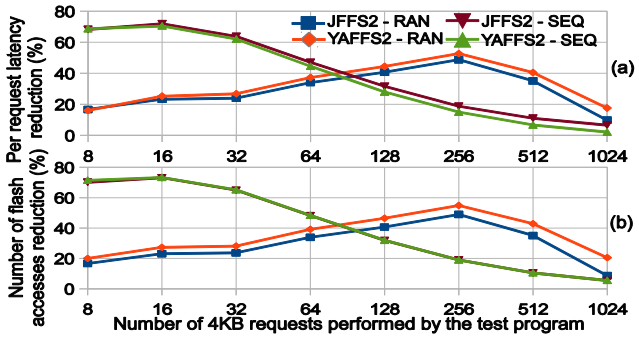

Figure 3. Number of requests variation

requests triggered by the page cache are served synchronously, thus delaying the application read response times. In other words the I/O latency due to data prefetching is added to the *read()* system call execution time which triggered the read-ahead pass. Because with MTD prefetching is done synchronously, read-ahead cannot mask I/O latencies from the calling process. Therefore, in the best case (when all prefetched data are used), read-ahead do not enhance the process read performance. Moreover, when prefetched data are not used, read-ahead can only have a negative impact on performance. For that reason, most of the curves presented in this paper show the improvement in performance / power consumption gained from disabling read-ahead.

4.1.2 Impact of Request Number

We varied the request number from 8 to 1024 on a 5MB target file for a fixed 4KB request size with no inter-arrival times between I/O requests. For the highest request number, most of the file is read. Figure 3-a shows the improvement rate on the total I/O response time when *disabling* read-ahead.

One can observe that disabling read-ahead always improves the performance. In addition, the behavior of both JFFS2 and YAFFS2 is similar as they behave the same under sequential and random workloads. In fact, read-ahead impact does not depend on the used file system, as the technique is implemented at the (upper) VFS level. Read-ahead just asks the file system for more data, which leads to more flash memory reads. This can be verified in Figure 3-b showing the results of the same experimentations in terms of number of flash memory reads generated (traced with Flashmon).

On sequential workloads, we can notice that the smaller the number of requests, the better the improvement when disabling read-ahead (we can observe more than 50% flash operations reduction when performing less than 64 requests on the same file). In fact, read-ahead is very active under sequential workloads, and issues many prefetching requests. As the size of the prefetching requests is stable (proved to be around 128KB for our experimentations), when the number of application read requests is small, the overhead of prefetching is relatively high (as all the prefetched data are not necessarily used). On the other hand, when the number of requests is high, a larger part of prefetched data is used inducing less overhead.

Under random workloads, one can also observe a performance improvement when disabling the read-ahead mechanism. It means that read-ahead is actually activated for random workloads. We can notice that the improvement is less impressive for small number of requests (20% of improvement for up to 32 requests). Indeed, read-ahead does not prefetch large data chunks. The improvement increases up to 50% for 256 requests. Under random workload, prefetched data is not likely to be accessed in a near future, unless the total accessed space is large enough (as compared to the target file size) to reveal temporal locality. This is what happens starting from 256 requests up to 1024: when the total addressed space is large, we observed many page cache hits.

A last common observation one can draw is that we get a minimal performance improvement from disabling the read-ahead mechanism when reading the whole file: this is because there is no waste in the prefetched data (all the prefetched data are read by the application).

4.1.3 Impact of Request Size

On a 32 MB target file, we varied the number of requests and the request size. Results are presented on Figure 4-a. Under

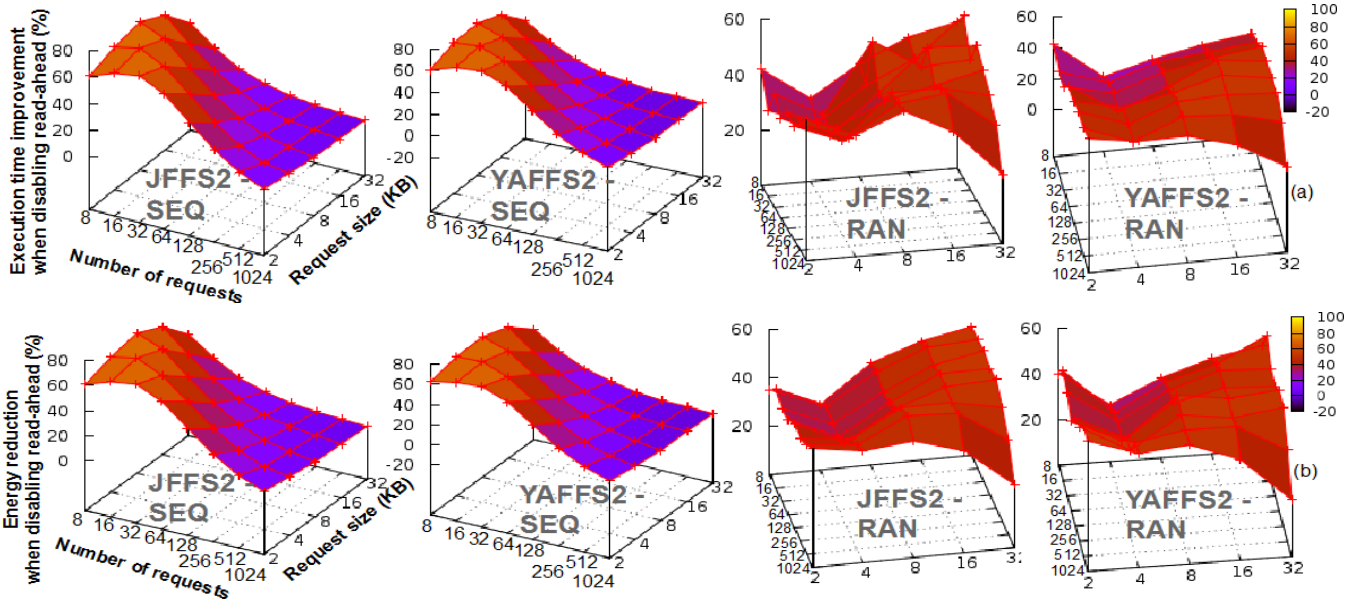

Figure 4. Performance (top) and power consumption (bottom) results, focusing on request number and size variation.

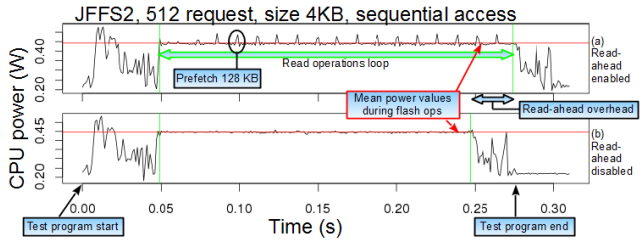

Figure 5. Example of power consumption measurement on the CPU power rail

sequential workloads, for a fixed small number of requests, we observe that the improvement when disabling read-ahead decreases with the increase of the request size. This behavior is similar to the observations made when varying the number of requests. Indeed read-ahead impact depends on the total space read (number of request multiplied by the request size).

Under random workloads the performance optimization is relatively stable, apart from large addressed spaces (1024 requests of 32 KB). This is caused by the large number of page cache hits generated when the addressed space converges to the total file size. On small addressed spaces the enhancement is lower as read-ahead does only prefetch a small amount of data for small random workloads. Finally, one can observe a slight enhancement for 2 KB requests as compared to 4KB size (size of a memory page). This is probably due to memory alignment issues.

4.2 Micro-benchmarks: Energy Consumption Estimation

We observed that the parameters impacting the power on the CPU are the file system type and the fact that read-ahead is enabled or disabled. In addition to these parameters, the memory power consumption is also affected by the access pattern. Results are presented in Table 2.

Table 2. Energy model constants

Com-ponent	File system	Access pattern	Read-ahead	Mean Power Cost (W) during read operations
CPU	JFFS2	no impact	ON	0.223
			OFF	0.223
	YAFFS2		ON	0.211
			OFF	0.214
Memory (RAM + flash)	JFFS2	SEQ	ON	0.029
			OFF	0.024
		RAN	ON	0.023
			OFF	0.023
	YAFFS2	SEQ	ON	0.028
			OFF	0.028
		RAN	ON	0.024
			OFF	0.026

The power cost is obtained by subtracting the idle power (power measured when executing the sleep command) from the power measured during reads. We observed that these values were stable enough to extract a simple energy model.

As one can see in Figure 5, the energy consumption is represented by a square area. Whether read-ahead is enabled or not, the magnitude of the power does not change. Thus the energy consumption is only impacted by the response time value. The energy model can be approximated as follows:

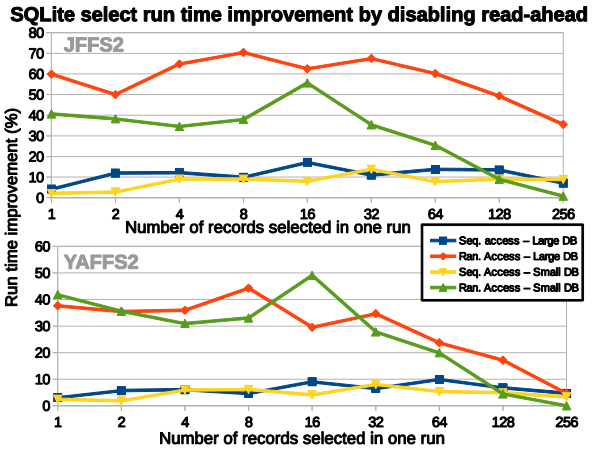

Figure 6. SQLite SELECT performance improvement for JFFS2 (top) and YAFFS2 (bottom)

$$E_{total} = t_{exp} * (P_{CPU} + P_{Mem})$$

E_{total} is the total energy consumed by the read operations in the experimentation, t_{exp} is the execution time of the read operations, and P_{CPU} and P_{Mem} are estimations of the mean power cost during read operations.

Results on energy consumption are presented on Figure 4-b, they exhibit a very similar behavior to the performance results. In fact, execution times of read requests are the main factor impacting the energy consumption. Indeed, as shown in Table 2, the variation of the mean power is very slight.

4.3 Read-Ahead impact on SQLite database read performance and power consumption

In this section we measured the impact of read-ahead on SQLite database performance, and the previously presented model is adapted to compute an estimation of read-ahead impact on the database accesses power consumption.

4.3.1 Performance impact measurement

As for the micro-benchmarking results, disabling read-ahead always reduces the execution time of the SQL SELECT runs, regardless of the file system type, access pattern, and size of the database (small / large). So once again, we plot the performance improvement gained by disabling the prefetching mechanism. Performance results are presented on Figure 6.

One can see that the performance improvement on sequential pattern is relatively stable and stays around 10%. Counting both small and large DB results, mean values for the improvement are 9.5% for JFFS2, and 5% for YAFFS2. We did not observe the important performance improvement on sequential workloads measured in the previous section. It may indicate that the majority of the prefetched data are actually used. It is also important to note that sequentially selecting records do not necessarily translates into sequential file read operations: this behavior depends on the way the records were inserted / updated, and more generally on SQLite internal algorithms.

On random workloads, one can see that read-ahead generates an important overhead: the performance improvement is up to 70% for JFFS2, and 50% for YAFFS2. Mean values are 44% and 25% for JFFS2 and YAFFS2 respectively. With random patterns, when the number of records read in one run increases, the performance improvement decreases. In that case, because of the

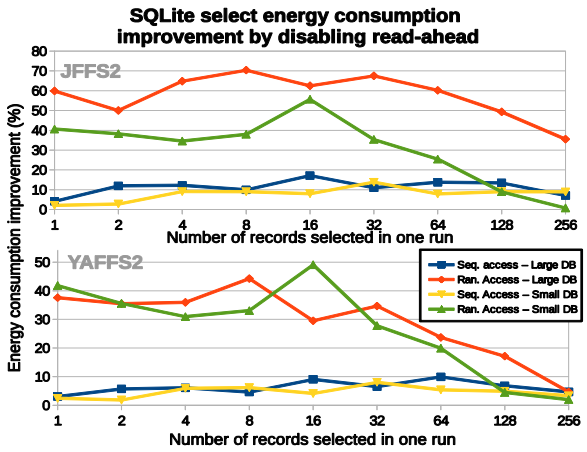

Figure 7. SQLite SELECT energy consumption improvement for JFFS2 (top) and YAFFS2 (bottom).

nature of the access, a large part of the file is read and prefetched (by small amounts) from the flash memory. Thus, the number of page cache hits increases and read-ahead calls are less frequent, minimizing its impact. This depends on the number of records selected in the run, and on the total size of the database file.

4.3.2 Power consumption impact estimation

Using the same methodology as in the micro-benchmarking phase, we measured the mean power during a large run of SELECT operations on both JFFS2 and YAFFS2 file systems, with read-ahead enabled / disabled. Results are presented in Table 3. We did not vary the access pattern as it was previously proven to have a small impact on power consumption.

Table 3. SQLite energy model constants

Com-ponent	File system	Read-ahead	Mean Power Cost (W) during SELECT operations
CPU	JFFS2	ON	0.24565
		OFF	0.23925
	YAFFS2	ON	0,245814
		OFF	0,245296
Memory (RAM + flash)	JFFS2	ON	0,030188
		OFF	0,036863
	YAFFS2	ON	0,033325
		OFF	0,036319

The mean power cost was multiplied by the execution time measurements of SQLite SELECT runs presented in the previous section. Figure 7 represents the estimated energy savings obtained by disabling read-ahead. Once again, as the various power values measured exhibit few variations, the execution time is the main influencing factor in the energy equation. Thus, the energy savings plots are very similar to the run time improvements presented in Figure 6. Taking both large and small DB configurations, the mean energy savings for JFFS2 are 8.6% (seq.) and 41% (ran.). For YAFFS2, these numbers are 5.3% and 25.5%.

5. CONCLUSION

Because of the synchronous nature of the NAND driver on embedded Linux, read-ahead has a negative impact on both performance and power consumption of raw NAND flash based embedded storage systems. We studied that effect on JFFS2 and YAFFS2 flash file systems. According to micro-benchmarking results, the main parameters influencing read-ahead impact were proved to be the access pattern, and the proportion of the file read by the process. Disabling the technique improved performance and energy consumption by up to 70% on our test platform. We also found that deactivating read-ahead lead to strong performance and power consumption improvements in the context of SQLite SELECT operations, particularly on random workloads.

Kernel patches for disabling read-ahead with JFFS2/YAFFS2 can be retrieved at the following URL: http://syst.univ-brest.fr/~pierre/Files/ra_ffs.zip.

6. REFERENCES

[1] Woodhouse, D. 2001. JFFS: The journalling flash file system. *In Proceedings of the Ottawa Linux Symposium* (Ottawa, Canada, July 25 – 28, 2001).

[2] Wookey. 2004. YAFFS2: a NAND flash file system. *UK's Unix & Open Systems User Group Linux Tech. Conf* (Leeds, United Kingdom, August 5 – 8, 2004).

[3] Hunter A. 2008. *A Brief Introduction to the Design of UBIFS*. www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf (online, accessed 07/2014).

[4] Wu F., Xi H., Li J., and Zou N. 2007. Linux readahead: less tricks for more. *In Proceedings of the Linux Symposium* (Ottawa, Canada, July 27 – 30, 2007), vol. 2, pp. 273–284.

[5] Katcher J. 1997. *PostMark: A New File System Benchmark*. Technical Report TR3022, Network Appliance.

[6] Park S., Jung D., Kang J., Kim J. and Lee J. 2006. CFLRU: a replacement algorithm for flash memory. *In Proceedings of the 2006 international conference on Compilers, Architecture and Synthesis for Embedded Systems* (Seoul, Korea, October 23 – 25, 2006), CASES, pp. 234–241.

[7] Ahn S., Hyun S., and Koh K. 2009. Improving demand paging performance of compressed filesystem with NAND flash memory. *In Selected Papers of the Seventh International Conference on Computational Science and Applications* (Yongin, Korea, June 29 – July 2, 2009), ICCSA'09, 84–88.

[8] Olivier P., Boukhobza J. and Senn E. 2014. Flashmon v2: monitoring raw flash memory accesses for embedded Linux. *ACM SIGBED Review vol. 11, issue 1*, Special issue of the Embed With Linux (EWiLi) International workshop, Toulouse, France, 2013.

[9] Hipp D.R. and Kennedy D. 2007. SQLite. www.sqlite.org (online, accessed 07/2014).

[10] Kim J.M. and Kim J.S. 2012. AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices. *In Frontiers in Computer Education Advances in Intelligent and Soft Computing Volume 133,*Springer, pp 667-674.

[11] Senn E., Chillet D., Zendra O., Belleudy C., Bilavarn S., Atitallah R. B. , Samoyeau C. and Fritsch A. 2012. Open-PEOPLE: open power and energy optimization platform and estimator. *In Proceedings of the 2012 15th Euromicro Conference on Digital System Design* (Cesme, Izmir, Turkey, September 5 – 8, 2012), DSD, pp. 668–675.

[12] Benmoussa Y., Senn E. and Boukhobza J. 2014. Open-PEOPLE, a collaborative platform for remote and accurate measurement and evaluation of embedded systems power consumption. *In 22nd IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.*