Embedded Databases on Flash Memories: Performance and Lifetime Issues, the case of SQLite

Jalil Boukhobza, Pierre Olivier,
Univ. Bretagne Occidentale
UMR 6285, Lab-STICC
F-29200 Brest, France
{boukhobza, pierre.olivier}@univ-brest.fr,

Loic Plassart
TréflévéNet
F-29800 Tréflévénez
loic.plassart@treflevenet.fr

Hamza Ouarnoughi, Ladjel Bellatreche
LIAS/ISAE ENSMA
Futuroscope, France
hamza.ouarnoughi@univ-brest.fr,
bellatreche@ensma.fr

Abstract — Databases are more and more used in embedded system applications and especially in consumer electronics. This comes from the need to structure user and/or system data to be more efficiently managed and accessed. SQLite is one of the most used database applications. This paper presents a micro benchmarking methodology and results for SQLite database requests on embedded flash specific file systems. Indeed, flash file systems behavior are very specific to flash memory intricacies and the objective of this study is to highlight the interactions between flash memory, flash file systems, and SQLite based applications.

Keywords Embedded databases, NAND Flash memory, Flash file systems, SQLite, JFFS, UBIFS, Performance, Lifetime.

I. INTRODUCTION

Embedded systems actors are nowadays experiencing the golden age of Non Volatile Memories (NVM). Indeed, according to Market Research [1], the NVM market will have an annual growth of 69% up to 2015. This tendency concerns several NVM technologies such as Ferroelectric RAM (FeRAM), Phase Change RAM (PCRAM), Magneto-resistive RAM (MRAM). Flash memory is however the most mature and disseminated NVM as its use is boosted by the ever-growing demand of smartphones and tablets market. In fact, mobile memory (including both NOR and NAND flash, DRAM and embedded multimedia cards) market has experienced a growth of 14% in 2012 (as compared to 2011) [2]. Moreover, NAND flash memory revenues have hit a new record as of $5.6 billion (a growth of 17%) in the fourth quarter (as compared to the third quarter of 2012). Flash memory has just celebrated its 25th birthday (1988-2013) after its creation in Toshiba labs and it is already shipped with almost 8 times more gigabytes than DRAM (in 2011). It became the process technology leader for memory fabrication and miniaturization.

Flash memory success is due to many attractive features such as good I/O performance, energy efficiency, shock resistance, small size and weight. These advantages come with some limitations/constraints the designers must deal with in order to maximize both lifetime and I/O performance. NAND flash memory is an EEPROM (Electrically Erasable and Programmable Read only Memory) on which one can perform operations at two different granularities. The smallest data unit used for read and/or write (program) operations is the page, while the data unit used for the erase operation is the block, with a block being composed of a given number of pages.

NAND flash memory constraints can be summarized as follows: (1) Write/erase granularity asymmetry: writes are performed on pages while erase operations are executed on blocks. (2) Erase-before-write rule: one of the most important constraints as 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 (and erase) operations is asymmetric.

Embedded and critical systems typically use flash memories as storage support. In addition, the implementation of databases is becoming increasingly popular for that kind of systems to replace simple files for data management and thus meet the growing use and complexity of information management.

The purpose of the encompassing project from which this study is a part is threefold: (1) to study the impact of unitary database requests on embedded flash memory storage systems from a performance and lifetime points of view (for instance: how many read, write, and erase operations are performed). (2) To model the behavior of all the software stack impacted by such requests for different flash file systems (achieving comparative studies). Finally, (3) to optimize embedded database systems according to the results obtained in (1) and (2). The system optimizations can be declined in, at least, three different solutions: (a) optimize the existing file systems to better manage database requests. (b) If the first solution is unfeasible, one might think of developing a "database aware" flash file system. Indeed, current flash file systems are primarily designed to support simple file data management. (c) Another solution one can think about is the design of optimization techniques (such as buffers) that can be implemented at different levels such as the database management system or at lower operating system layers. This paper focuses on the first part of the study that is the impact of database requests on I/O performance and lifetime of embedded flash memory based storage systems.

II. Background on Flash Memories

A. Some Basics on Flash Memory

Flash memories are based on floating gate transistors and can be divided mainly into two types according to the logical gate used as the basic component: (1) NOR flash memories support random data access, are more reliable, have a lower density, and a higher cost (as compared to NAND flash memory). NOR flash memory is mainly used for storing executed code as a replacement of DRAM (e.g. in mid to low range mobile phones). (2) NAND flash memories are block addressed, offer a high storage density for a lower cost and are used as secondary storage. This paper only concerns NAND flash memory.

Three different NAND flash memory technologies exist: (1) Single Level Cell (SLC), (2) Multi Level Cell (MLC) and (3) Triple Level Cell (TLC). In SLC, one bit is stored in each memory cell, while in MLC, two bits can be stored, and 3 bits for TLC (highest number of voltage states). From a bit density and cost per bit points of view, TLC is better than MLC that outperforms SLC. From a performance and reliability points of view, SLC performs better than MLC that is better than TLC. From the application point of view, TLC is used for low end media players, mobile GPS, and more generally non critical data applications that do not require frequent data updates. MLC and SLC are used for more data intensive appliances such as SSDs, mobile phones, and memory cards.

B. Flash Memory Management

To cope with the above-mentioned NAND flash memory constraints, some specific management mechanisms are implemented. (1) In order to avoid costly in-place updates (block erase and data write operations), a logical-to-physical address mapping mechanism is used, allowing to perform *out-of-place* data modifications (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, some specific data blocks containing "hot" data can wear out quickly. To avoid this problem, some 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 used. A garbage collector is generally used to perform this task.

Mapping mechanism, wear leveling and garbage collection services can be implemented either in hardware or in software.

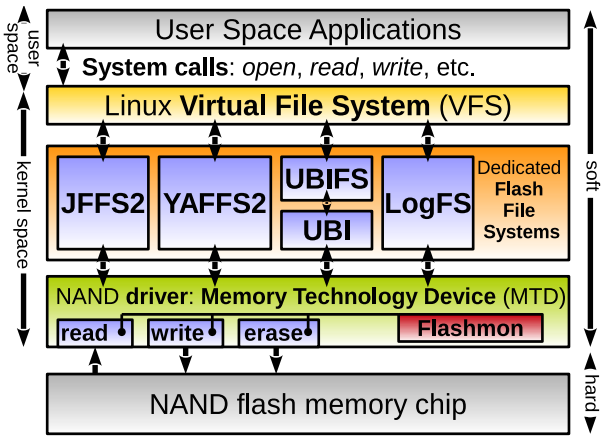

Figure 1: I/O request software stack and Flashmon location in the Linux NAND storage hierarchy

In hardware, a Flash Translation Layer (FTL) is implemented to perform the aforementioned services [3]. This is the case of some devices such as USB sticks, compact flash, and SSDs. Bare flash chips that can be found in most embedded systems such as smartphones, tablets, personal navigation devices, and video camcorders are generally managed by the operating system through some specific Flash File Systems (FFS). In addition to flash specific management services, FFS should achieve all traditional file system tasks: file and directory hierarchy organization, user access rights, etc.

C. Dedicated Flash File Systems

In embedded systems equipped with raw flash chips, NAND flash storage is mainly managed using dedicated Flash File Systems (FFS). Such hardware platforms only embed a simple NAND controller used to perform basic flash operations. The flash management is achieved at the FFS level, which is a software layer included in the operating system. Then, FFS can be viewed as a purely software flash management method. The Linux OS implements today's most popular FFS: JFFS2 [4], YAFFS2 [5] and UBIFS [6].

Figure 1 illustrates the FFS layer location inside the Linux NAND storage software stack. User space application access files using system calls. System calls are received by the Virtual File System (VFS), which role is to abstract the specifications of all the actual file systems supported by Linux. Moreover, at the VFS level, Linux maintains several caches in RAM in order to speed up files access. In particular, the Linux page cache (formerly buffer cache) is dedicated to file data buffering. VFS maps system calls to FFS functions. The FFS determines the NAND operations to perform, according to its state and algorithms. To access the flash chip, the FFS uses a NAND driver called the Memory Technology Device (MTD) layer. It is a generic driver for all kinds of NAND chips supported with Linux.

The above-mentioned FFSs present common features in their implementation. Files are divided into data nodes. When files are updated, old nodes are invalidated and new nodes are created. Invalid nodes are recycled through garbage collection which is generally performed asynchronously through the execution of a background kernel thread. As FFS implements out-of-place updates, nodes can be at a variable location on the flash media. To keep track of the nodes locations, FFS use indexing mechanisms which related metadata are also stored on flash.

In this study we chose to perform our tests on two FFS: JFFS2 and UBIFS. JFFS2 is a very mature and widely spread FFS, used for more than a decade (mainlined since Linux 2.4.10 in 2001). UBIFS, for its part, is a relatively recent FFS with a growing usage. UBIFS was created to address several of JFFS2 design issues strongly impacting file system performance [7].

III. Embedded Databases

Today, the database functionality is needed to provide support for storage and manipulation of data in embedded systems [8]. Many different database management systems (DBMS) can be used to better meet the requirements of current embedded systems. In most cases, the implementation of embedded databases must rely on limited resources. Embedded DBMS must also exhibit robustness towards sudden power cuts. Among the existing solutions, three should particularly be considered: Berkeley DB, Firebird and SQLite.

Berkeley DB [9] is a database engine compatible with several operating systems. Since version 2.0, Berkeley DB is available under both a free OSI-certified license and a commercial license. It comes in the form of a C-written library providing an API. Connectors exist for many programming languages such as C (native interface), C++ and Java. A Berkeley DB database is only composed of records whose format is freely determined by the calling program. There is no table concept, and the database cannot be used with a data manipulation language such as SQL. Berkeley DB is considered simple to use and supports concurrent accesses by multiple users.

Firebird project [10] originally started as the Borland InterBase database. Firebird is an open source SQL relational database management system that runs on MS Windows, Linux and various UNIX flavors. Firebird is written in C++. A native API is provided to connect applications to Firebird databases. A Firebird database is operated using SQL language.

SQLite [11] is an open source C library providing a relational database engine that can be operated by the SQL query language. SQLite is a lightweight relational database engine that can be integrated directly into an application. It is supposed to improve data storage and management potentials of embedded applications [12][13]. SQLite stores each database in one file, and each file consists of a given number of SQLite pages (the default value for the page size is 1KB). SQLite is widely applied for data management of embedded environment, such as smartphones, industrial control, etc. [14]. Today, SQLite is probably the most widely used DBMS for embedded applications and systems.

For the sake of this study we chose to concentrate on SQLite DBMS. This choice is motivated by the large adoption of SQLite. Moreover, the fact that SQLite is open source, in addition to the large available documentation are criteria that can help in explaining tests results. The developed methodology can however be applied to whatever DBMS on embedded system that supports the SQL language.

IV. Benchmarking Methodology and Tools

A. Global Benchmarking Methodology

As discussed above, the objective of this study is threefold: (1) to evaluate (measure) the impact of unitary database requests on flash memory storage system from a performance and lifetime points of view. (2) to achieve comparative studies on different FFS and model the behavior of each of them for the tested SQL requests, and finally, (3) to optimize embedded database systems according to the obtained results.

For the sake of this paper, we focused on the first step of the study that is the performance evaluation and interpretation of the impact of SQL requests for embedded databases on flash memories based storage systems with a focus on two specific FFSs: JFFS2 and UBIFS.

Figure 2 describes the global methodology followed in our work. One can observe on the left hand side the tested platform. As discussed earlier, we relied on SQLite database engine for creating our databases and issuing SQL requests. SQLite has been installed on an embedded Linux platform

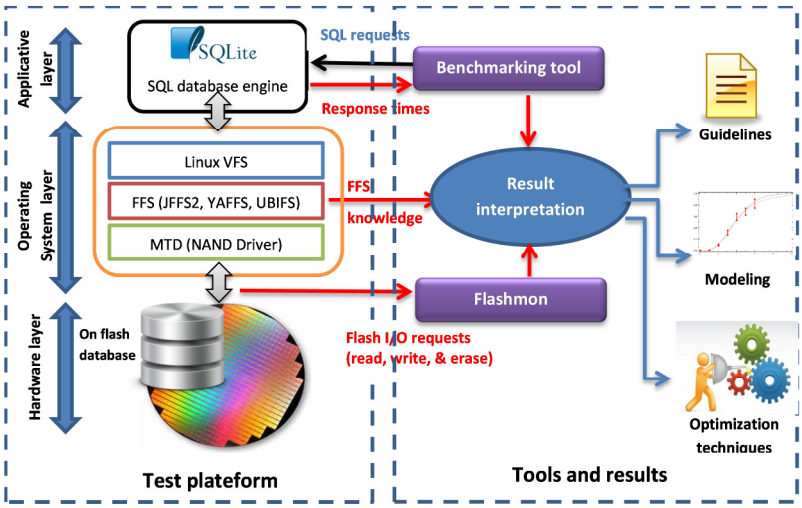

Figure 2: Embedded database performance evaluation methodology on flash based storage systems

executing a FFS on top of an embedded flash memory chip. A benchmarking tool relying on the SQLite API has been developed to automatically create the database, generate unitary SQL requests and measure the completion time of each request. In parallel to that, a tool, Flashmon [15][15], that monitors all flash memory accesses (read, write, and erase operations), is executed in order to profile and trace SQL queries. Flashmon (described farther) gives a precise idea on the number of generated flash memory I/O accesses for a given workload. The novelty of our approach resides in using both quantitative and qualitative information in order to understand the system performance. By quantitative measures, we mean SQL request response times, while by qualitative measures; we mean the type and number of flash memory operations generated for our tests (number of reads, writes, and erasures). The use of both type of information gives more hints for understanding the interactions between the SQLite engine and the FFS (and more generally the embedded operating system, and the flash memory).

The results of our experimentations can allow to: (1) provide the developer with guidelines on how to better use the SQL database engine, which file system to use and how to tune it in order to get better performance. (2) Secondly to model performance of SQL requests with the objective to extract cost models in order to predict performance for a given workload. (3) Finally, to design optimization techniques based on the understanding of the performance behavior and the modeling step (cost models) [16]. The optimization can be applied on different levels: applicative layer (database engine itself) or the operating system layer (the flash management layer, FFS).

B. Metrics and Tools

In order to understand the performance of embedded databases, we mainly relied on two metrics: the response times (and throughput) of SQL requests, describing the system performance, and the number of generated flash memory operations: reads, writes, and erasures. This allows a better understanding of the performance in addition to a precise idea on the flash memory wear out through the metrics of number of erasures and number of writes.

1) Embedded Database Benchmarking Tool: The tests consist in issueing SQL requests on a given table, and measuring the unitary response time of each request. The

tested SQL requests are: the insertion, selection, join, and update of records. We first begin by generating a simple table that has two fields: a short integer that is the primary key and a character string. This string contained random data to avoid disturbance caused by the data compression performed at the FFS layer. Two parameters were varied for each SQL request: the number of records and their size (the table size). Each test was repeated many times to be sure of getting stable results. Response time was obtained with the gettimeofday system call (microsecond granularity). The database is located on a dedicated test partition of 50MB initially configured in JFFS2 format and then in UBIFS format to compare results. The partition is fully erased then formatted before each test to ensure a homogeneous initial state. Each set of experimentations for a given SQL request was preceeded by a BEGIN TRANSACTION instruction and ended by an END TRANSACTION instruction. It is recommended [17] to use those two instructions especially in embedded systems, for instance, to prevent data corruption caused by power failures in the middle of a data-base transaction. This is implemented throughout the creation of journal log file that is specific to a given transaction and that is removed once the transactions succeeds.

Algorithm 1.Microbenchmarking algorithm

1: **Input**:
2: Number of requests: Nb_{Req}
3: Record size: Rec_{size}
4: Query type: Q_{type} = insert | select | join | update
5: File system: Fs = jffs2 | ubifs
6: **Output:**
7: Response time of all requests: RT[Nb_{Req}]
8: Flashmon output (I/O trace) for the experiment : IOtrace
9: **Init()**
10: Format & mount test partition with Fs
11: Reset I/O tracer Flashmon
12: Create database schema
13: **if**(Q_{type} = select | join | update)
14: Fill database with random data
15: **end if**
16: Empty all system caches
17: Initialize I/O tracer Flashmon
18: **MainFunction():**
19: Init()
20: RT[0] = responseTime(BEGIN TRANSACTION)
21: **for (i = 1; i <=Nb_{Req}; i ++)**
22: RT[i] = responseTime(request i of Q_{type} ,with Rec_{size})
23: **end for**
24: RT[Nb_{Req}+1] = responseTime(END TRANSACTION)
25: Return IOtrace
26: Return RT

The database benchmark described in the preceeding algorithm was executed on UBIFS and JFFS2, for the insert, select, join (nested loop join), and update SQL queries with record sizes of 150, 300, 450, and 600 bytes, and a number of requests going from 100 to 5000 with an increment of 100.

2) The Flash Memory Monitor Flashmon: Flashmon [15] is a Linux kernel module allowing to trace NAND flash I/O low-level operations on raw flash chips. Flashmon stands for flash monitor and traces the page read, page write and block erase operation performed on physical pages / blocks of the flash memory. The module is implemented at the MTD subsystem level (see Figure 1), it is then independent from the FFS and the NAND chip layer. Therefore, it can be used with all the previously evocated FFS, on all the NAND flash chips supported by the Linux MTD subsystem. Flashmon traces the operation type, time of arrival, physical address targeted, and also the name of the current task executed when the operation is traced. As MTD is completely synchronous, the traced task is responsible of the traced operation.

Flash management mechanisms such as dedicated FFS are rather complex due to the specific constraints exhibited by the memory. One simple I/O request from the applicative layer can end up in several read / write / erase flash operations according to the FFS state and algorithms as well as the flash memory state. So, in the context of this paper, performance evaluation of traditional metrics such as I/O response times or throughput are not sufficient and should be completed by some precise knowledge on the flash operations occurring during the benchmark. Flashmon helps embedded systems developers / researcher to extract and collect such information.

C. Hardware & Software test platform

The benchmarks were launched on the Armadeus APF27 development board [18]. It embeds an ARM9 based Freescale i.MX27 microprocessor clocked at 400 MHz, and 128 MB of RAM. The board contains 256 MB of Micron SLC NAND flash. Blocks in this flash chip are composed of 64 pages of 2 KB each. The chip datasheet [19] indicates that read, write, and erase operations' latencies are respectively 25 µs, 300 µs and 500 µs. From a software point of view, we used the 2.6.29.6 version of the Linux kernel, and SQLite 3.7.10.

V. RESULTS AND DISCUSSIONS

In this section, we try to analyze and discuss the results of the performance measures on the tested SQLite operations: "select", "insert", "join", and "update" operations.

A. Flash memory throughput calibration

In order to assess the raw throughputs offered by the flash chip and its management (driver and FFS), we ran a series of simple tests at various levels in the storage software stack (previously presented on Figure 1). We measured throughputs for (A) moving data from RAM to flash and (B) from flash to RAM in order to evaluate respectively flash write and read performance. These measurements were achieved at three levels: (1) the driver (MTD) level, under the FFS layer and close to the hardware layer; (2) the MTD applicative level: MTD programs allowing to access flash from user-space bypassing the FFS; and (3) the applicative level: a simple C program accessing files through the FFS. Results are depicted in Table 1. As one can see, the complexity induced by adding multiple software layers between the application performing flash I/O operations and the flash chip itself leads to an important performance.

TABLE I. MEASURED FLASH READ & WRITE THROUGHPUTS AT VARIOUS LEVELS IN THE FLASH STORAGE MANAGEMENT STACK

Level	Write (MB/s)	Read (MB/s)
MTD kernel level (driver)	3.36	5.94
MTD userpace level (bypassing FFS)	2.33	3.80
Applicative level (using FFS)	1.26	3.65

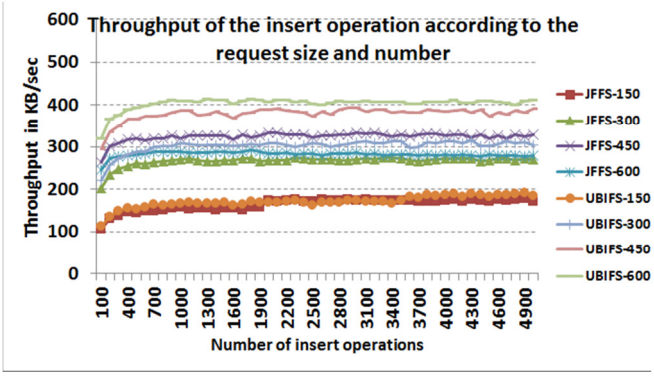

Figure 3: I/O throughput (KB/s) of the insert operations according to record size and number of requests

B. *The "insert" operation*

For the insert operation, we created a database that was filled by inserting records in a loop via the SQLite API. We varied the number of inserted records and their size. The SQLite configuration used is the default one with the size of a SQLite page equal to 1KB.

From the throughput results shown on Fig. 3, one can observe that: (1) the throughput for both JFFS2 and UBIFS is low as compared to the throughput the flash memory can sustain for simple write operations (see previous section). This is due to meta-data management (file-system and SQLite) that is investigated farther in this section. (2) UBIFS performance is better than JFFS2 for most cases (except small 150 bytes records). The reasons behind such a performance difference are due to many file-system related factors that are highlighted in the following sections. We also can notice a large performance gap between small and large size records. This is related SQLite overhead of page management.

From the per-request response time analysis (see Table 2), one can draw three main observations (note that in addition to each request response time, we measured both "BEGIN TRANSCTION" and "END TRANSACTION" response times): (1) the insert operation response times are almost constant for a given request size, for both JFFS2 or UBIFS. (2) The "BEGIN TRANSACTION" command at the beginning of each test takes almost a constant time to execute (around 32ms), whatever the request size, number or mounted file-system. (3) The major performance difference between JFFS2 and UBIFS is noticed when the transaction is committed through the "END TRANSACTION" command. We noticed that the response time of this command is related to the size the inserted data (number and size of insert requests) and the file system. In fact, as we will see farther, this response time is related to the flush operation of the SQLite buffer to the flash memory.

In order to understand the difference between the behaviors of both tested file-systems, we performed some qualitative measures on the number of flash read and write accesses for the achieved tests. Fig. 4 shows the number of read I/O accesses per KB of data according to the number of insert requests for different record sizes (150, 300, 450, and 600).The curves show that for both reads and writes JFFS2 do more accesses to the flash memory. UBIFS performs a small fixed number of read operations that do not depend on the number of inserted data while for JFFS2 the number of performed reads strictly depends on the volume of inserted objects. Concerning the write operation, one can observe that JFFS2 performs more writes than UBIFS, this is due to two main characteristics: UBIFS maintains a proper buffer allowing to absorb more write operations, while JFFS2 does not (it is completely synchronous). The other reason behind such a behavior is that for JFFS2, we observed that the system performs interleaved reads and writes during the insert operation while UBIFS generates only writes. This may concern the JFFS2 metadata reading/checking.

TABLE II. SNAPSHOT OF RESPONSE TIMES (IN MS) OF INSERT OPERATIONS FOR DIFFERENT REQUEST SIZES ON JFFS2

	Record size (Bytes)/ resp. times in ms		
Query nb	150	300	600
0 (BEGIN TR.)	32,75	35,02	34,29
1	7,27	6,47	6,83
11	0,66	0,63	0,91
12	0,78	0,61	0,77
13	0,76	0,67	0,78
14	0,81	0,60	0,90
15	0,66	0,60	0,79
16	0,65	0,67	0,78
17	0,65	0,83	0,92
18	0,65	0,61	0,90
19	0,74	0,67	0,78
101 (END TR.)	20,86	39,07	116,84

Finally, we also noticed an extremely interesting behavior related to SQLite. In fact, it maintains a very large buffer which prevents data from being flushed to the FFS underlying layer until the END TRANSACTION commit instruction is issued. Once the commit launched, all inserted objects are flushed to the flash memory, provided that the size of the inserted data is under a given threshold. In fact, above a given size of inserted data, flush operations from the SQLite buffer to the flash begin. This threshold was measured to be approximately 1.75MB. After this threshold flush operations to the flash memory are pipelined with the insert operations. We infer from this behavior that in Table 2, measured response times (except the first and last one) for the first requests (size > 1.75MB) are related to SQLite management of data in memory.

C. *The "select" operation*

In this section, we give some elements on the performance of the select operation. Before performing the tests, the database was filled through a set of insert requests. For each

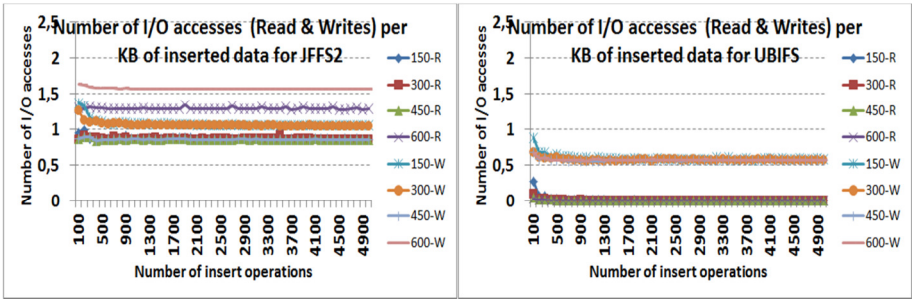

Figure 4: Number of I/O accesses to the flash memory for insert operations according to record size and number of requests

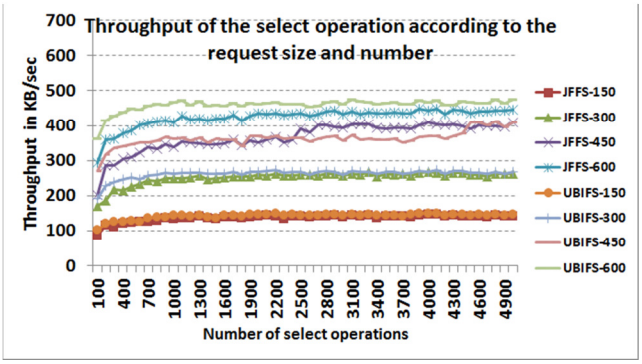

Figure 5: I/O throughput (KB/s) of the select request according to record size and number of requests

record size, we defined a fixed size data base according to the maximum requests it receives (5000 in our case).

A first observation one can do when looking at Fig. 5 representing the throughput of the select SQL operation for different record sizes and request number, is that UBIFS slightly outperforms JFFS2 for the select operation for some measured record sizes. The throughput follows a logarithmic shape for both file-systems due to the constant overhead related to the log journal manipulation and other SQLite specific processing and memory usage.

Response time analysis (see Table 3) shows more stability (less value variations) for UBIFS response times as compared to JFFS2. In fact, for UBIFS, we can observe periodic small peaks (~2ms) which periodicity decreases according to the request size. The peaks have values that are approximately the double of the other stable values. For JFFS2, the amplitude of the peaks is much more significant as the value is around 42ms. The period of those peaks is also much larger and depend on the request size. In fact, this phenomenon (in JFFS2) is related to the Linux page cache read-ahead algorithm that prefetches chunks of 128KB (around 40ms, giving ~3MB/s throughput) of data, allowing future SQLite read accesses to be served from RAM rather than flash. This behavior is only observed for JFFS2 as in UBIFS the read-ahead mechanism is disabled by default [20]. Read-ahead is mainly designed for hard disk drives, in which case system management allows asynchronous IO: while the system prefetches data, the CPU can execute other tasks taking benefit from I/O timeouts. As stated earlier the NAND driver (MTD) is fully synchronous, so in the best case read-ahead does not impact flash memory I/O performance. In the general case, read-ahead lead to a performance drop on flash storage, because all the prefetched data is not necessarily accessed in the future. This is the reason why it is disabled in UBIFS, and one of the explanations of the performance difference between the two tested FFS.

Even though UBIFS disables the read ahead mechanism of the Virtual File System of Linux, one can observe that the period of the peak response times is 4, knowing that each record (600 B for this example) is stored in a separate SQLite page (of 1KB), we can infer that the prefetch size is 4KB (2 flash pages). In fact, this is related to the I/O system page granularity on Linux which is 4KB.

TABLE III. SNAPSHOT OF RESPONSE TIMES (IN MS) OF SELECT OPERATIONS FOR 600 BYTES OBJECTS ON JFFS2 AND UBIFS

	JFFS2 (req. Of 600 B)	UBIFS (req, Of 600 B)
Query nb	Time (ms)	Time (ms)
0	30,52	33,25
43	0,96	1,93
44	1,06	0,93
45	0,96	1,14
46	42,65	1,17
47	0,99	1,98
48	1,07	1,18
49	0,98	1,06
50	0,96	1,04
51	1,06	2,00
52	0,96	1,19
109	1,05	1,06
110	42,14	1,17
111	1,00	1,83
112	0,98	1,24

When observing the number of accesses (here I/O reads) in Fig. 6, one can clearly see the impact of the read-ahead prefetching algorithm. In fact, the peak values in Fig. 6 (for JFFS2 especially for 150 B objects) represent the case where a last prefetching (of 128KB) of data has been made but was not profitable as prefetched data were not used, thus generating more flash reads as compared to the needed data. In addition to the read-ahead related behavior, under JFFS2, the system performs more read operations especially for request number less than 1300 select operations. This is coherent with the I/O throughput observed in Fig. 5 but one could await a larger difference in the throughput when looking at the number of I/O reads generated for both file-systems. In fact, the flash memory I/O accesses times count for a given percentage of the total execution times but the SQLite processing and memory activity should not be ignored. They represent a large part of the execution time especially for small number of requests. For the example shown in Table 3 in case of JFFS2, measures show ~42ms response time each 64 requests. So if we consider a period of 64 requests, around 42ms are related to flash memory management while ~1*63ms are due to SQLite processing (memory and CPU). This proves the important overhead related to non-I/O operations.

D. The "join" operation

For the join operation requests, we created two tables from which we selected all elements of the second table that have an identifier (key) which value is equal to the one of the first table. This is done on a given number of objects corresponding to the number of requests. This join operation simply selects

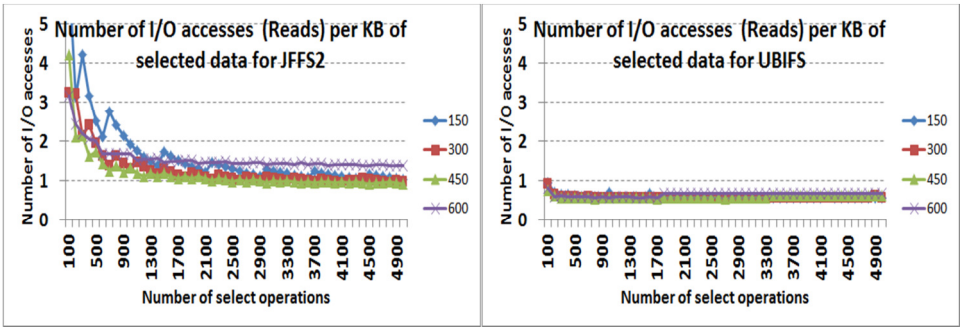

Figure 6: Number of I/O accesses to the flash memory for select operations according to record size and number of requests

all the elements of the second table which identifier is smaller than the given number of requests. So the executed request looks like:

"*SELECT table2.val FROM table1, table2 WHERE table1.id = table2.id AND table2.id <= nb_request;*"

Fig. 7 shows the results for the measured throughput of the join operation relative to the set of selected objects. One can observe that for a high number of selected objects, UBIFS and JFFS2 give approximately similar results. Another observation one can do, is that we obtain better performances for join operations on 450 B objects than on 600 B objects for both file-systems. This is due to the internal fragmentation generated when the record size is 600 B, in this case, one SQLite file page (1 KB) contains only 1 record and the rest of the storage space is unused, while for record sizes of 450 B, two objects are put in 1 SQLite file page (filling 900 B of 1KB page as compare to 600 B).

The join operation performs only read I/O accesses to the flash memory layer. Fig. 8 shows the number of flash read operations performed per KB of requested data for both file-systems and different request size and number. The figure shows that for large request numbers, 600 B record request sizes give the worst performance for both file-systems due to internal fragmentation. For JFFS, we can observe the same phenomena as for the select requests, the read-ahead algorithm of the VFS page cache provokes some peaks that are visible especially for 150 B objects. The read-ahead mechanism is active whatever the record size and request number, but is graphically more visible when small portion of data are accessed. This is due to the fact that the overhead of the prefetch is proportionally higher.

One can also observe that for a small number of request sizes, UBIFS outperforms JFFS2, while for large numbers, performance are approximately equivalent as the number of flash memory reads per KB of accessed data is around 2 for both systems.

If we compare the join operation to the previous select query tests, one could question about the fact that the join gives far better performance. This is related to the manner with which the tests were performed. For the select operation, we issued one request per selected record while for the join, one query was issued to read the whole table objects. The difference between both cases is very important as the SQLite related processing overhead is much higher when issuing as many requests as there are records in the table. A single select on the same objects retrieved with the join operation gave better results which is normal. Indeed, for the join, the IDs of both tables are compared and the values of objects of the second table are retrieved while for the select, only the values of one table are read (no comparison).

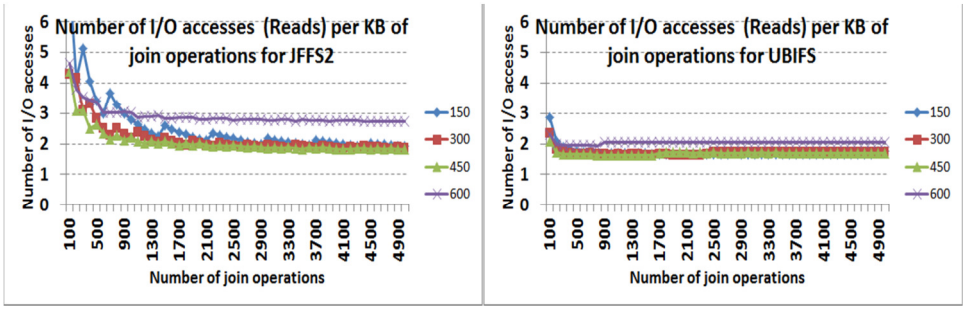

Figure 8: Number of I/O accesses to the flash memory for join operations according to record size and number of requests

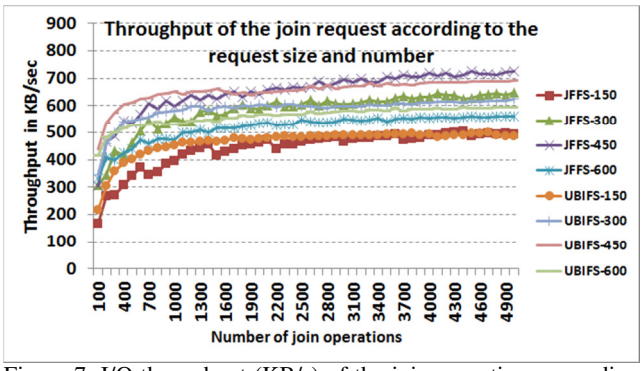

Figure 7: I/O throughput (KB/s) of the join operations according to record size and number of requests

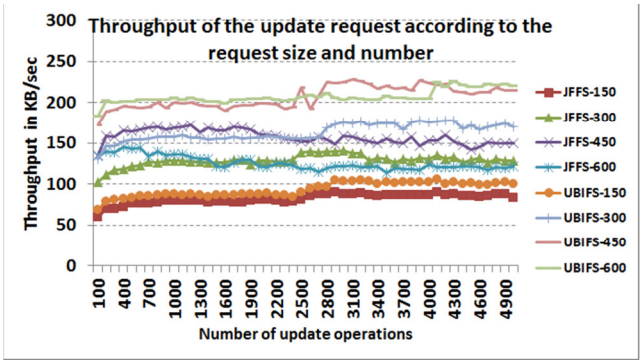

Figure 9: I/O throughput (KB/s) of the update operations according to record size and number of requests

E. The "update" operation

A first extremely important observation one can draw is the cost of the update operation as compared to the insert for both UBIFS and JFFS2. Updates are 2 times less performing than inserts for both JFFS2 and UBIFS. In order to update one object, the system needs to read the flash page containing the SQLite page which encapsulates the record to update. So to update the object, the system needs to rewrite all the data that are in the SQLite page.

We can also observe that UBIFS performs better than JFFS2, especially for large update sizes. This is mainly due to the additional buffering on UBIFS which prevents synchronizing the SQLite file page as frequently as JFFS2. Indeed, it collects the updates on the buffer before flushing to the flash memory. This behavior is clearly underlined by the number of write accesses described in Fig. 10. In addition to the additional read operations due to meta-data management of JFFS2, it performs an average of more than two times more write operations.

We can also see that for JFFS2, a larger number of erase operations are performed that can reach values of 0.07 block erasures per updated KB of data. Knowing that a block consists of 64*2KB pages, we can say that for updating 1KB of data for the 600B objects, we erase an equivalent of 4.48 flash pages (0.07*64), which is very significant.

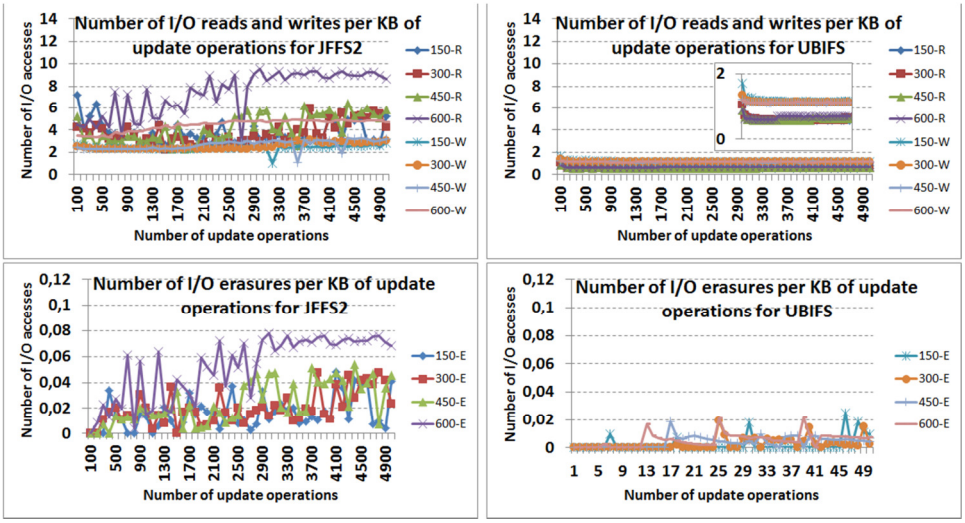

Figure 10: Number of I/O accesses to the flash memory for update operations according to record size and number of requests

The values of the throughput and read/write I/O accesses are unstable, especially for JFFS2 due to the asynchronous garbage collector that is launched while the update test is running, generating additional flash read/write / erase operations and thus disturbing response times of the SQL queries.

VI. Conclusion and Future Works

This paper presents a set of results on measuring the performance of SQL requests targeting on-flash SQLite running on an embedded Linux operating system. As observed from the presented results, the performance behavior highly depends on the used FFS and can have a significant impact on the lifetime of the flash memory, for instance under JFFS2, the system can perform up to 0.07 to 0.08 erase operations for updating the equivalent of 1KB of data.

The performed tests revealed a very high disparity according to the varied parameters for the tested queries and the used FFS. The obtained results are closely related to the used FFS as it has a strong interaction with the SQL engine output due to the constraints of flash memories.

We can conclude that the performance of SQLite on embedded flash storage system depends on (1) the I/O load generated by the applicative layer (SQL requests) and its own buffering mechanisms; (B) the flash management algorithms (FFS) and (C) the state of the system: system / FFS caches state, but also flash memory state in terms of amount and location of valid / invalid / free pages. This was especially observed for update operations in which the garbage collector was launched after a given number of write operations were performed.

For future works, we consider investigating more SQLite complex requests as joins. We are also about to study real SQLite traces from typical embedded applications. Finally, we did not focus on the initial state of the flash memory in this paper. In fact we considered clean partitions at the beginning of each test. It would interesting to inject different initial states in terms of invalid/valid/clean pages and see how FFS deal with different configurations.

References

[1] Market Research, “Advanced Solid State Non-Volatile Memory Market to Grow 69% Annually through 2015”, from http://www.marketresearch.com/corporate/aboutus/press.asp?view=3&article=2223, MarketResearch press release, 2012, (accessed on 06/2013).

[2] Storage Newsletter, “NAND Chip Market Growth Propelled by Smartphones and Media Tablets”, from http://www.storagenewsletter.com/news/marketreport/ihs-isuppli-nor-nand-chip, 2012, (accessed on 06/2013).

[3] J. Boukhobza, "Flashing in the Cloud: Shedding Some Light on NAND Flash Memory Storage Systems." Data Intensive Storage Services for Cloud Environments. IGI Global, 2013. 241-266.

[4] D. Woodhouse, “JFFS: The journalling flash file system,” in Ottawa Linux Symposium, 2001, vol. 2001.

[5] Wookey, “YAFFS2: a NAND Flash File System,” 2004.

[6] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif, “Abstract Specification of the UBIFS File System for Flash Memory,” in FM 2009: Formal Methods, vol. 5850, A. Cavalcanti and D. R. Dams, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 190–206.

[7] A. B. Bityutskiy, “JFFS3 design issues”, from http://www.linux-mtd.infradead.org/doc/JFFS3design.pdf, 2005 (accessed 12/2013).

[8] V. Kataria, S. Amit, P.Singh, S. Pravin, "Commercial Embedded Database Systems: Analysis and Selection", 2004.

[9] M. A. Olson, K. Bostic, M. Seltzer, "Berkeley DB", in Proceedings of the USENIX Annual Technical Conference, 1999.

[10] Firebird developers, Firebird website, from http://www.firebirdsql.org/, 2013 (accessed 12/2013).

[11] SQLite developers, SQLite homepage, from http://www.sqlite.org/, 2013 (accessed 12/2013).

[12] J. Li, Y. Xu, "Remote Monitoring Systems Based on Embedded Database", Proceedings of the 3rd International Conference on Genetic and Evolutionnary Computing (WGEC'09), 2009.

[13] K. Yue, L. Jiang, L. Yang, H. Pang, "Research of Embedded Database SQLite Application in Intelligent Remote Monitoring System", in Proceedings of the International Forum on Information Technology and Applications (IFITA'10), 2010.

[14] G. Qinlong, C. Xingmei, T. Weiwei, Y. Minghai, "Study and Application of SQLite Embedded Database System Based on Windows CE", in Proceedings of the 2nd International Conference on Information Science and Engineering (ICISE'10), 2010.

[15] P. Olivier, J. Boukhobza, E. Senn, “Flashmon v2 : Monitoring Raw Flash Memory Accesses for Embedded Linux,” in Proceedings of the Embed With Linux Workshop (EWiLi'13), 2013.

[16] L. Bellatreche, S. Cheikh, S. Breß, A. Kerkad, A. Boukhorca, et J. Boukhobza, « How to Exploit the Device Diversity and Database Interaction to Propose a Generic Cost Model? », in Proceedings of the 17th International Database Engineering & Applications Symposium, New York, NY, USA, 2013, p. 142–147.

[17] SQLite developers, Journal mode pragma statement, from http://www.sqlite.org/pragma.html#pragma_journal_mode, 2013 (accessed 12/2013).

[18] Armadeus Systems, “APF27 Board Datasheet”, from http://www.armadeus.com/_downloads/apf27Dev/documentation/dataSheet_APF27Dev.pdf, 2012 (accessed 12/2013).

[19] Micron Technology, Inc., « MT29F2G16ABDHC-ET:D NAND Flash Memory Datasheet ». 2007.

[20] UBIFS developers, UBIFS and Linux read-ahead, from http://www.linux-mtd.infradead.org/doc/ubifs.html#L_readahead, 2008 (accessed 12/2013).