Towards MRAM Byte-Addressable Persistent Memory in Edge Database Systems

Luís Meruje Ferreira[1,2], Fábio Coelho[1,2] and José Orlando Pereira[1,2]

[1]INESC TEC, Campus da Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
[2]University of Minho, Campus de Gualtar, Rua da Universidade, 4710-057 Braga, Portugal

Abstract

There is a growing demand for persistent data in IoT, edge and similar resource-constrained devices. However, standard FLASH memory-based solutions present performance, energy, and reliability limitations in these applications. We propose MRAM persistent memory as an alternative to FLASH based storage. Preliminary experimental results show that its performance, power consumption, and reliability in typical database workloads is competitive for resource-constrained devices. This opens up new opportunities, as well as challenges, for small-scale database systems. MRAM is tested for its raw performance and applicability to key-value and relational database systems on resource-constrained devices. Improvements of as much as three orders of magnitude in write performance for key-value systems were observed in comparison to an alternative NAND FLASH based device.

Keywords

MRAM, edge databases, persistent memory, microcontroller

1. Introduction

It is expected that by 2025, 80% of all data will be generated in endpoints with embedded computing capabilities [1], such as those typically present in Internet-of-Things (IoT) devices. At the edge, these use cases are enabled by microcontroller (MCU) or microprocessor (MPU) unit devices due to their compact, low energy consumption, and affordable design. On the downside, these devices have very limited resources when compared to traditional datacenter servers, making it challenging to design data management systems with such requirements.

The application domains of data systems on resource-constrained devices are multiple. Key-value stores are used as lightweight, low resource-consuming data storage solutions for edge applications [2, 3, 4, 5]. Other scenarios require more complex data processing, which allied to the wide use of the SQL language led research towards the relational model, and SQL as its interface. Relational databases can be found either as centralized [6, 7, 8], or as distributed deployments, [9, 10], combining the computation capabilities of multiple endpoints.

At the storage level, FLASH storage has established itself as the standard storage technology used for persisting data in both MCUs and MPUs. FLASH storage is often either embedded, or is provided externally in the form of SD cards [11, 6, 12, 13].

Problem The physical characteristics of FLASH storage limit the flexibility with which read and program (i.e, write) operations can be performed. Consequently, data management systems are required to implement additional mechanisms to cope with these limitations, leading to greater computational overhead. Furthermore, FLASH storage provides limited endurance, i.e., a limited number of erase-write cycles per storage cell.

Looking at the impact of FLASH storage on the application domains in scope for this analysis, key-value stores are hindered by FLASH storage's poor performance for small I/O operations and random accesses, which represent the most common workload for these systems [14]. As for relational database systems, the limited CPU capabilities of the considered devices lead to a greater impact of FLASH storage management overhead on overall performance.

Contributions This work evaluates the viability of Magnetoresistive Random-Access Memory (MRAM), a type of persistent byte-addressable memory, as a possible alternative or complement to FLASH storage for resource-constrained devices. We show that MRAM provides advantages in throughput, power consumption, endurance, and software complexity. More specifically,

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Workshop on Accelerating Analytics and Data Management Systems (ADMS'23), August 28 - September 1, 2023, Vancouver, Canada

luis.m.ferreira@inesctec.pt (L. M. Ferreira); fabio.a.coelho@inesctec.pt (F. Coelho); jop@di.uminho.pt (J. O. Pereira)
https://www.inesctec.pt/en/people/luis-manuel-ferreira (L. M. Ferreira); https://www.inesctec.pt/en/people/fabio-andre-coelho (F. Coelho); https://www4.di.uminho.pt/~jno/sitedi/nm_617.html (J. O. Pereira)
0000-0003-3364-3670 (L. M. Ferreira); 0000-0002-0188-6400 (F. Coelho); 0000-0002-3341-9217 (J. O. Pereira)

CEUR Workshop Proceedings (CEUR-WS.org)

the following **contributions** are provided:

- *Comparison of MRAM and FLASH storage characteristics* - We compare the nominal endurance, energy expenditure, storage capacity, and monetary cost of each type of device as advertised by the corresponding vendors. Moreover, we determine and discuss the advantages and disadvantages of each type of device.
- *Comparison of MRAM and FLASH storage performance* - We experimentally compare the throughput of each type of device under varying I/O operations sizes, both in terms of their raw performance and for their performance under relevant use cases for resource constrained devices, namely key-value and relational database systems.

To evaluate MRAM's capabilities, a new prototype was developed, combining a state-of-the-art MCU with an MRAM memory device. Results show that MRAM is capable of providing full throughput at much smaller I/O operations when compared to FLASH storage, enabling it to provide 3 orders of magnitude better performance in key-value applications. For the case of relational databases, MRAM can forego FLASH specific mechanisms such as wear-leveling, thus freeing resources that can be used instead by the DataBase Management System's (DBMS) query engine.

The rest of the paper is organized as follows: Section 2 provides the necessary background on MCUs and MPUs, FLASH storage, and MRAM persistent memory; Section 3 compares the characteristics of both storage solutions in terms of vendor-provided information; Section 4 details how different data management systems were adapted to work with MRAM memory; Section 5 provides the results of our practical evaluation; and Section 6 discusses the results. Finally, Section 7 draws conclusions and provides possible paths for future work.

2. Background

MPUs vs MCUs Microprocessors (MPUs) are processing units including multiple processing cores (e.g., up to 5 cores in recent offerings [15]), with frequencies over 1GHz, and usually coupled to a few GB of memory (under 10) and hundreds of GB of storage (e.g., SD cards). MPUs usually serve as intermediaries between cloud and end devices, i.e., IoT or edge gateways [13, 16, 11]. RaspberryPi is a popular MPU based device that is recurrently used by researchers in IoT and Edge related publications [17, 18]. MPUs, due to their greater amount of resources and support for the appropriate primitives, often support the use of operating systems such as Linux, so developing systems for these devices is very similar to doing so for a commodity server.

Microcontrollers (MCUs), on the other hand, are limited to up to 1 MB of memory, and a few MB of internal FLASH storage, and single core processing units with frequencies under 0.5GHz. The capabilities of these devices can be extended by adding external memory and storage. They are mainly used as end devices in IoT systems such as sensors or actuators, and are frequently powered by batteries, so power consumption is of major concern. Furthermore, they do not support full-fledged operating systems, so programming MCUs is a lower-level experience.

MCUs constitute the first layer of edge databases, often being the data generators of these systems [16, 9, 10, 13]. Historically, this data was mostly offloaded to MPUs or more capable cloud nodes; however, with the increase in the number of MCU devices, concerns about network overload started to arise, as the increasing number of parallel connections to centralized processing servers imposes a large load on network resources [19]. Furthermore, publications have shown that for an MCU, data transmission can consume more energy than local storage and processing [6]. When coupled with the fact that the capabilities of MCUs continue to increase and that, compared to MPU systems, MCUs are more affordable and consume less energy [20], it makes sense to push as many processing and storage tasks as possible towards MCUs.

FLASH storage FLASH storage is the most common type of storage solution used in both MPU and MCU devices, with the two main types of FLASH used being NOR FLASH and NAND FLASH.

- **NOR FLASH.** NOR FLASH memory has fast read speeds, but has less storage capacity and higher cost per byte than NAND. As such, NOR FLASH memory is often used to store application code in MCUs since code tends to be small, and fast read speeds mean faster execution times (although developers can manipulate unused NOR space however they want). Overall, NOR FLASH memory tends to be used in predominantly read-intensive workloads.
- **NAND FLASH.** NAND FLASH memory, on the other hand, because it is cheaper, provides better write performance and greater storage capacity, is the most widely used technology of the two. It is the underlying technology of most digital storage media, such as SD cards and SSDs.

FLASH storage devices (both NAND and NOR) are organized into *blocks*. Each block contains a series of *pages* (e.g., 128 pages), where each page can store, for example, 8KB of data [21].

Erase operations are the only way in flash memory to convert data bits to 1. Furthermore, the smallest unit that can be erased is a block, affecting multiple pages. Generally, NOR FLASH tends to have much slower erase speeds than NAND FLASH.

Program (i.e. write) operations are done at the page level, and data bits can only be changed from 1 to 0. This means that a data section can only be written once after each erase operation. Similarly, read operations are also performed at the page level.

The fact that the lowest unit of control in FLASH storage is a page hinders its overall performance. For example, if an operation affects only a part of a page, the entire page must still be read or written, and the unwanted data will be ignored. In cases where the data to be read or written fits within a single page, but the data happens to be unaligned such that it is split between two pages, both pages must be read or written. Erase operations being performed at the block level restrict write operations. For example, for a write operation to be performed over a page which is not erased, and the data in the remaining pages to be kept, all pages in the block must be erased and rewritten. This is why erase operations are often delayed until multiple pages have been marked for deletion.

Furthermore, FLASH storage devices support a relatively low number of erase-write cycles per storage block, after which a block can no longer be modified. Therefore, systems must adopt wear-leveling mechanisms, where write operations are carefully spread so that no block is subjected to substantially more write operations than the others. Finally, FLASH storage devices provide asymmetrical performance. Accessing random addresses is slower than accessing sequential positions, and write operations are slower than read operations.

MRAM persistent memory Magnetoresistive Random-Access Memory (MRAM) is a type of persistent memory where data is truly byte-addressable. For the case of the devices showcased here, data is organized into 1 or 2 byte cells, with the possibility for each byte to be read or written independently. Furthermore, data bits can be freely converted between 0 and 1 by write operations, forgoing the need for data to be erased.

Compared to FLASH storage, MRAM provides better read / write performance, more write cycles per cell, as well as symmetric performance for sequential and random accesses. Furthermore, due to being byte-addressable, reads and writes can reach their maximum throughput even with very small operations, whereas FLASH storage only achieves maximum throughput for operations involving multiple kilobytes of data.

MRAM presents similar characteristics to 3D XPoint [22], the byte addressable persistent storage technology on which Intel Optane is based. Contrary to 3D XPoint, however, MRAM chips are available for use with MPUs and MCUs, whereas Intel Optane is only available for more capable computers.

Despite MRAM's technology being available since the 1980s[23], it was only recently that significant advances in performance and chip density have made MRAM attractive for data management applications. It is important to understand how these new MRAM chips compare to current FLASH storage technologies, in order to understand the viability of MRAM as either a replacement, or complement, to the standard FLASH technologies currently in use.

3. MRAM vs FLASH

To understand the viability of MRAM as an alternative to FLASH storage, four MRAM devices with increasing capacity and read/write performance are compared with NAND FLASH and NOR FLASH devices. The MRAM devices chosen were: (M1) AS30040316 [24], (M2) MR4A16 BMA35 [25], (M3) EMxxLx [26] and (M4) EMD4E001GAS 2 [27]. The corresponding M1-M4 notations are used for each of these devices to ease referencing during the rest of this Section. As for FLASH storage, MT29F128G08AJAA AWP-ITZ:A [21] was selected to represent NAND FLASH and MT28EW512ABA1HPC-0SIT [28] was chosen to represent NOR FLASH.

Furthermore, the following characteristics for each device are analyzed:

- Read/Write/Erase throughput - the performance throughput of a device for read, write and erase operations. The metric considered was Megabytes per second (MB/s). Note that as per the discussion in Section 2, erase operations do not apply to MRAM devices.
- Capacity - the amount of data a given device is able to store, in megabits.
- Endurance - the number of writes or erases that a particular data cell can endure before the vendor no longer guarantees correct functioning of the data cell.
- Energy - the amount of energy required to perform a write operation. The metric considered was nanojoules per Byte written.
- Cost - the monetary cost of a given device per amount of storage capacity. The metric considered was euros per megabit of storage capacity.

The values of the characteristics analyzed for each device are presented in Table 1. Values were calculated based on information made available by each device's datasheet. For performance throughput, the values presented correspond to the maximum nominal values. The energy consumption figures are based on either peak consumption or typical consumption values, depending on the information made available by vendors.

Table 1
Storage devices' characteristics comparison.

	Read	Write	Erase	Capacity	Endurance	Energy	Cost
MRAM							
$M1$) AS3004316 [24]	57 MB/s	57 MB/s	N/A	4 Mb	100T	1.58 nJ/B	6.63 €/Mb
$M2$) MR4A16BMA35 [25]	57 MB/s	57 MB/s	N/A	32 Mb	Inf	1.58 nJ/B	1.99 €/Mb
$M3$) EMxxLx [26]	400 MB/s	400 MB/s	N/A	64 Mb	Inf	0.895 nJ/B	0.84 €/Mb
$M4$) EMD4E001GAS2 [27]	2.6 GB/s	2.6 GB/s	N/A	1 Gb	0.01T	0.523 nJ/B	0.098 €/Mb
FLASH							
NAND FLASH [21]	235 MB/s	23.5 MB/s	737 MB/s	128 Gb	60K	7.02 nJ/B	0.0012 €/Mb
NOR FLASH [28]	337 MB/s	2.5 MB/s	0.65 MB/s	512 Mb	100K	66.8 nJ/B	0.023 €/Mb

Performance All considered MRAM devices outperform both FLASH devices in write performance, between $2.42\times$ and $1040\times$. As for read performance, both FLASH devices are outperformed by the M3 and M4 MRAM devices by a factor of between $1.18\times$ and $11\times$. Furthermore, NAND FLASH is $479\times$ faster than NOR FLASH when erasing a block of data. Since MRAM can override data without first deleting it, its operations are not affected by erase performance, which also greatly simplifies the management of data being stored on MRAM, when compared to FLASH storage.

Endurance MRAM supports at least $100000\times$ more operations per cell than FLASH memories, and some devices claim an unlimited number of operations during the lifetime of the chip. As such, there is no need for employing wear-leveling mechanisms, meaning less operational overhead. This also translates into a longer life for the device, making it a better choice for scenarios with high data churn.

Energy In the case of MRAM, the energy required to write a single byte has an inverse correlation with its throughput performance. All MRAM devices show a lower energy consumption when writing data compared to FLASH devices, requiring $4\times$-$13\times$ less energy compared to NAND FLASH, and $42\times$-$127\times$ less energy compared to NOR FLASH.

The two major drawbacks of MRAM are capacity and cost.

Capacity The most capable MRAM device, M4, has a storage capacity of 1000 Megabits, which is $2\times$ the capacity of the NOR FLASH device, but $128\times$ less than the capacity of the NAND FLASH device. Recent advances have achieved multi-Gb capacity in single MRAM chips [29], however, we have not considered these devices for analysis, as they are not yet widely available for commercial use, with vendors marketing those devices only for space-grade applications. Although this is still significantly less than the hundreds of gigabits that a NAND chip can support, it may be enough for current edge and IoT persistent storage requirements.

Cost MRAM has a higher cost per MB than NAND and NOR FLASH. The M4 MRAM device (the less expensive per byte) is $4.26\times$ more expensive than the representative NOR FLASH device and $81\times$ more expensive than the NAND FLASH device. However, there is a logarithmic relationship between the capacity of the MRAM chip and its price per megabit, i.e., as the density increases, the price decreases significantly. If the MRAM chip density continues to increase and this relationship is maintained, we can expect the gap between the cost of MRAM and FLASH memory chips to decrease.

4. Data Systems on MRAM

Three systems were either implemented or adapted to run over MRAM to understand how MRAM memory can impact each of the two use cases previously identified for data storage in resource-constrained devices: key-value stores and relational database systems. Since MRAM works similarly to common volatile Random-Access Memory (RAM), two structures commonly used for in-memory key-value storage were selected: a Linear Probing Hash Table (LPHT) and a Cache-Line Hash Table (CLHT) [30]. Since MRAM is persistent, such data structures can easily be adapted to provide the equivalent of a key-value store. For comparison, RocksDB, a well-established persistent key-value store, was selected as a baseline. Since RocksDB is a more complex system than the selected hash tables, a more capable computation unit was assigned, to offset the increased computational overhead (see Section 5).

For the case of relational databases, we needed a system that could easily be adapted to run on either an MPU or MCU without changing its core functionality, in order to provide a fair comparison. With that objective, SQLite was selected since portability across different operating systems is guaranteed by its separate OS layer,

which allows for custom implementations. Each of these systems interacts with MRAM through a custom driver which supports write and read operations in multiples of 1, 2, 4 or 8 bytes. Below, we detail how each system was adapted to run over MRAM.

Linear Probing Hash Table The LPHT was implemented from scratch, supporting Insert, Read and Update operations. It separates MRAM's space into two sections: one for metadata, which keeps tracks of the occupation state for each key-value slot, and a second for data, which stores the actual key-value pairs. The size of these pairs must be set before the hash table is used, and all key-values share the same size.

Information on occupied slots is stored in an array of bits, where each bit keeps the occupation state of a key-value pair slot. If the bit is set to one, the slot is occupied, otherwise it is free.

- **Insert Operation** - Insert operations are performed through the *put(key,value)* command. When the *put()* command is called, the key is hashed into one of the key-value slots. If the slot is occupied, a try is made for the slot that follows immediately after, and so on, until an empty slot is found. When an empty slot is found, the key-value pair is written into that slot, and then, the bit indicating the slot is occupied is set to 1. If no slot is found, the hash table is full and the insert operation fails.
- **Update Operation** - Update operations are also performed when the *put()* command is called. If during an insert operation, the key is found already stored in the hash table, the corresponding value is replaced with the new one, i.e., update operations replace the old value with a new one.
- **Read operation** - Read operations are performed through a *get(key)* command. Similarly to an insert operation, a read is performed by hashing the key to a slot, and traversing the corresponding and successive slots until either the key is found in an occupied slot, in which case the value is returned; or until an empty slot is found, or all slots are traversed, returning a null value in that case.

Although not implemented, removing a key-value pair is as simple as flipping the occupation bit corresponding to the affected pair to 0.

Each operation in the MRAM memory is split into 16-bit or 8-bit operations, which are performed one at a time over the memory. Assuming that these operations are atomic, insert, remove (if implemented), and read operations are crash-consistent, meaning that in case of failure, the hash table would guarantee a consistent state. Update operations, however, would need further mechanisms to ensure crash consistency. As it is unclear from vendor datasheets whether such elementary operations guarantee atomicity, this issue deserves further study.

Cache-Line Hash Table The CLHT [30] is a dynamic hash table that increases its size as more pairs are added. The table consists of a series of buckets, where each bucket contains a set of key-value pairs, a lock, and a pointer to the next bucket. As such, keys are hashed into positions of the hash table, where each position is composed of a linked list of buckets. CLHT supports insert, read, and remove operations.

CLHT's main advantage is the fact that each bucket is sized to fit into a cache line, thus greatly accelerating consecutive operations in the same bucket, a common occurrence both when inserting and when fetching key-value pairs.

To run a CLHT on MRAM, a series of modifications were applied to the original implementation [31], more specifically to the *Lock-based* version. First, locking was disabled, as the prototype developed only has a single core (see Section 5 for setup details). Although a *Lock-free* version is also provided, that version of CLHT uses snapshotting mechanisms to allow concurrent operation, which incurs computational overhead that is undesirable in an MCU.

Secondly, all read and write operations of the hash table on the underlying storage device are redirected through the MRAM driver. Third, a simple custom heap memory area was implemented on MRAM, since the original implementation relied on *malloc* for space allocation, which caused memory fragmentation when enforcing alignment constraints. By using our own heap implementation, no memory space is wasted. Our heap implementation currently only supports allocating more space. We leave implementing deallocation and defragmentation operations to future work.

Finally, the size of the bucket and key-value pair was adjusted to fit the cache line size of the MCU selected to interface with the MRAM device. Each key or value occupies 4 bytes, and a bucket is set to a size of 32 bytes, holding 3 key-value pairs and additional metadata. The rest of the codebase remained unchanged.

SQLite SQLite is a highly portable embedded relational database. However, it is more commonly used in MPUs, since previous MCUs were not able to run this database system [7]. Even so, with advances in MCU capabilities, and by augmenting an MCU with MRAM, we were able to successfully run SQLite on an STM32 (a popular line of MCUs). To do so, a custom OS portability layer is required [32]. The OS layer establishes how SQLite interacts with the underlying file system and OS calls.

It includes functions for retrieving random values, and current time; and also functions for opening, reading, writing, and closing files.

To build the custom OS layer, three components were required: the OS layer implementation itself; LittleFS [33], a file system for MCUs; and the MRAM driver. The MRAM driver performs low-level read and write operations on the MRAM. LittleFS, in turn, provides a lightweight file system that requires only a handful of functions to be implemented, such as writing and reading data to the storage medium. In this case, this functionality is provided to LittleFS through the MRAM driver. Finally, the custom OS layer makes use of LittleFS to implement file operations, while OS functions such as random number generation are implemented using functions provided by native STM32 libraries.

5. Experiments

For the experimental setup, two devices were used: an STM32 MCU with MRAM memory, and an MPU, more specifically a Raspberry Pi 3B, with an SD card as its storage medium (i.e., NAND FLASH storage). The main characteristics of each are described in Table 2.

The STM32H743ZI microcontroller (MCU) [34] is a single core, 32 bit, 480MHz processing unit that comes with 2MB of NOR FLASH memory and 1MB of RAM memory. The MCU connects to an AS3004316 MRAM memory [24], with 4Mb of storage capacity, and 35ns access time both for read and write operations of either 8 or 16 bits. This MCU has 16 Kilobytes of L1 cache for instructions, and 16 Kilobytes of L1 cache for data. By default, both caches are disabled. For the tests depicted here, the instruction cache is always enabled, however the data cache is set depending on the test being run. Whenever data cache is used, it is set as write-through, so that any write to the cache is immediately persisted to MRAM memory.

The RaspberryPi 3B is driven by a 64 bit BCM2837 microprocessor (MPU), boasting 4 cores at 1.2Ghz. It has 1GB of RAM memory, and uses a SanDisk Extreme SD Card, with 32GB of storage capacity.

Notice that the MRAM uses between $10\times - 100\times$ less energy than the SD Card, and that the Raspberry Pi has considerably more computational power and memory resources than the STM32H743ZI MCU.

For easy reference, the names *STM32* (as well as MRAM), and *RPi* (or one of NAND FLASH or SD Card setup) are used throughout this section to describe the MCU and the MPU based setups, respectively.

It is possible to interface both NAND and NOR FLASH, as well as MRAM, with both MPUs and MCUs. However, this specific setup was selected as it was the option with the greatest potential for success, given that a custom circuit board had to be designed and produced to interface the STM32 with the MRAM device.

Table 2
Hardware specifications.

	STM32	RPi
CPU Model	STM32H743ZI	BCM2837
CPU frequency	480 MHz	1.2 GHz
CPU cores	1	4
RAM	1MB	1GB
Storage class	MRAM	SD Card
Storage device	Avalanche AS3004316 [24]	SanDisk Extreme [35]
Storage size	4Mb	32GB
Peak energy	66 mW	360-1440 mW[a]
Max. write cycles	10^{14}	$10^3 - 10^{4}$[b]
Cost (Euros)	60	50

We perform a series of experiments to assess the viability of MRAM as a suitable alternative, or complement, to current FLASH based storage. First the raw performance of the considered devices is evaluated, and then their performance is compared under key-value and relational database scenarios.

5.1. Raw performance evaluation

The read and write throughput capabilities for the storage mediums in each device are evaluated, both in sequential and random access scenarios. Furthermore, the relation between I/O block size and throughput performance is evaluated.

Testing methodology For MRAM, a random string with length equal to the desired operations size was generated, and written to the device, either to sequential or random addresses. As for reads, blocks of data of the desired size were read, from random or sequential addresses. The addresses were selected before the test was run. In the case of random addresses, duplicates are allowed, so a particular location may be overwritten multiple times. As for sequential addresses, if the maximum address is reached, operations wrap around the initial address. All tests run until 500MB are read or written. The STM32's L1 data cache is disabled for this test. In the case of the SD Card, `fio`, an open-source I/O tester [40], was used. Each test runs for 20 seconds, with a ramp up time of 2 seconds. We chose the following settings for `fio`: the engine chosen was `libaio`; `iodepth` is set to 20; the `direct` option is set to 1; and there is only 1 job running at a time. Results were averaged over 5 independent runs. The direct option only allows operation sizes equal or

[a] Estimation based on: [36, 37]
[b] Estimation based on: [38, 39, 37]

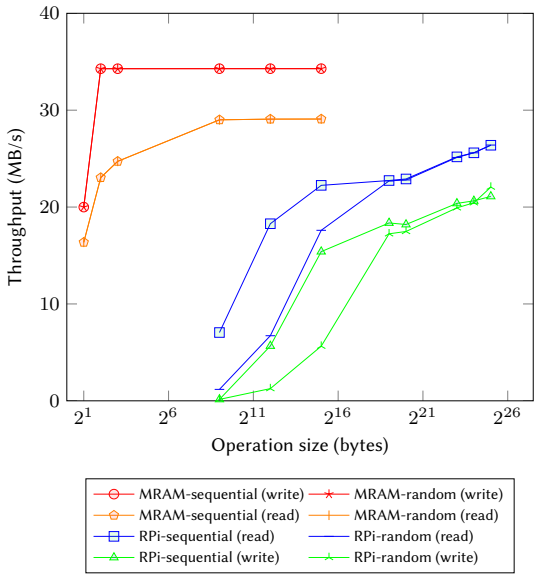

Figure 1: Storage medium read and write throughput.

greater to the page size of the device, so operation sizes for the SD Card start at 512 bytes.

Figure 1 shows the performance of both MRAM and RPi's NAND FLASH under read/write sequential/random workloads with varying request sizes. MRAM is able to achieve its maximum throughput with I/O blocks as small as 4 bytes for writes, and 512 bytes for reads, due to its byte-addressability, being able to maintain that level of throughput as block size increases. Maximum speeds of 34MB/s for writes and 29MB/s for reads were observed. Both random and sequential read/write patterns presented identical performance (notice the overlapping lines in Figure 1). The SD card achieved 22MB/s for reads, and 26MB/s for writes at *block sizes* of multiple kilobytes. Furthermore, random accesses present lower performance than sequential accesses. In conclusion, the MRAM device is able to provide higher throughput than the SD Card storage on the Raspberry Pi for all *block sizes*, especially at I/O operation sizes under 4KB. We confirm that, in the case of MRAM, random or sequential accesses have no impact on performance, with the results for both types of accesses being almost exactly the same. However, we notice that there is a difference in performance between write and read operations, with write operations outperforming the latter. We leave to future work determining the cause for this discrepancy. In the case of the SD card, we note that the read speed is also higher than write speed, which is uncommon for FLASH storage. This is, however, inline with the results found in previous tests for SD card performance with Raspberry Pis [41].

5.2. Impact on key-value systems

Being one of the identified use cases for data management systems in IoT and edge related systems, where resource constrained devices are used to store and process data, the impact of using MRAM for key-value systems is evaluated. In this experiment, I/O operations of varying sizes are executed over different key-value systems. The objective of the experiment is to evaluate how the previously identified advantage in raw performance affects these systems. Both an LPHT, and a Hash Table previously adapted to work with Intel Optane [42], CLHT, are implemented on the STM32 over MRAM. Since data stored in MRAM is persistent, both Hash Tables provide a similar service to a persistent key-value store, although with less functionality. We compare them with RocksDB, a popular persistent key-value, running on the RPi. We run RocksDB both with and without *fsync*, a configuration which when turned on guarantees persistence for each write operation. Single and multi-threaded execution is also considered for the case of RocksDB. We acknowledge that RocksDB is a more complex system than a simple hashtable, but the RPi's MPU gives it a significant computational advantage over the hashtables running on the STM32. We also include results without *fsync*, giving RocksDB the advantage of not having to persist its wal log on every single write operation.

Testing methodology For the key-value scenarios, a series of string arrays were generated separately, in order to ensure that the operations submitted to each of the evaluated systems is identical, and that the data generation process does not affect performance estimation. Datasets composed of arrays of randomly generated 2, 4, 8, 16, 32, 64, 128, and 256 byte strings were built. String deduplication was not performed, making it possible to have multiple put operations for the same key. The size of each dataset is equal to roughly 50% of the storage capacity of the MRAM memory (i.e., 2Mb). For the 2, 4, 8, 16, 32, 64, 128, and 256 byte datasets 65365, 32768, 16384, 8192, 4096, 2048, 1024, and 512 entries were generated, respectively. For each byte size, 5 different arrays were generated.

In the case of RocksDB versus LPHT (Section 5.2.1) the write experiment progresses as follows. One of the 5 datasets with 2 byte strings is selected. For each string *A* in the dataset, an operation of the type put(A,A) is performed, using the same string for both the key and value fields. The performance of each system is then averaged over the 5 different data sets for the same byte size. The same procedure is followed for the remaining byte sizes, and a similar procedure is followed for the read workload, but with *get(A)*, instead of *put(A,A)* operations. For the case of RocksDB, different combinations of *fsync* (on or off) and number of client threads are tested, as they

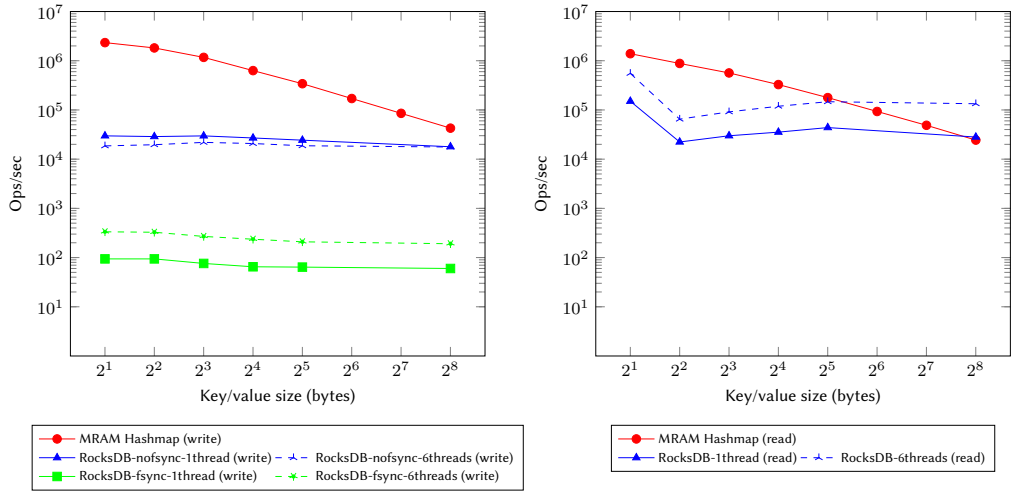

Figure 2: RocksDB (NAND FLASH) vs Linear Probing Hashmap (MRAM) with varying key/value size. Results for write and read operations show on the left side, and right side, respectively.

have a significant impact in system performance. When multiple client threads are used, the elements of each dataset are split as equally as possible amongst them. Since RocksDB with fsync turned on performs significantly slower, tests targeting this setup are limited to 5000 *put* operations per data set.

In the case of RocksDB versus CLHT (Section 5.2.2), the key and value field sizes are fixed to 4 bytes each, so only the 4 byte data sets are used, since the size of buckets must align with the size of a cache line. Furthermore, each run is fixed to 20000 *put* operations, due to the added space occupied by CLHT's additional structures. Similar to LPHT, CLHT is initialized with space to fit 2 times the amount of data that is inserted in each test.

For both LPHT and CLHT, at the end of each run, a consistency check is performed, where each of the stored values is retrieved from the Hash Table, and checked for correctness. We highlight that for the specific case of LPHT, we observed up to 0.002% of pairs missing pairs from the table when checking for consistency in scope of a run. We consider this to be due to a problem with our circuit board design for the MRAM memory chip, since slowing down the speed of the memory solves these errors.

Data L1 cache was enabled for all tests involving key-value systems, with the cache policy set to write-through. Since the results of the experiments resulted in disparities of multiple orders of magnitude, a logarithmic scale is used for the y-axis of all diagrams, which represent operations per second.

5.2.1. LPHT vs RocksDB

Figure 2 compares the number of operations per second that RocksDB and the LPHT are able to perform, when the size of a single key and corresponding value increases. The size represented on the horizontal axis, in bytes, corresponds to the size of a single key, or a single value. This experiment's conclusion is that the MRAM setup outperforms the NAND FLASH alternative on almost all scenarios. For write operations (left side of Figure 2), MRAM outperforms all RocksDB setups. However, as the key/value size increases, the difference between the STM32 setup, and the RocksDB setups where fsync is turned off, shrinks. At a key/value size of 4 bytes, MRAM is able to perform $35\times$ more operations per second than RocksDB with 1 thread and no *fsync*. But, when the size is increased to 256 bytes, the ratio between the two is only $1.4\times$ (still in favor of MRAM).

The LPHT running on MRAM memory guarantees persistence on each write operation, so the RocksDB setups that more closely resemble it are the ones where *fsync* is enforced. When guaranteeing persistence at each operation, the multithreaded RocksDB setup is vastly outperformed by the STM32's Hash Table, with LPHT performing between $134\times$ and $3837\times$ more operations per second.

For the case of read operations (right side of Figure 2), LPHT is able to outperform the multithreaded RocksDB for key/value sizes under 32 bytes, by a factor of between $1.64\times$ and $6.69\times$ more operations per second. For the case of the single threaded RocksDB, the Hash Table is able to outperform RocksDB for key/value sizes under 128 bytes, with up to $20\times$ more read operations per second.

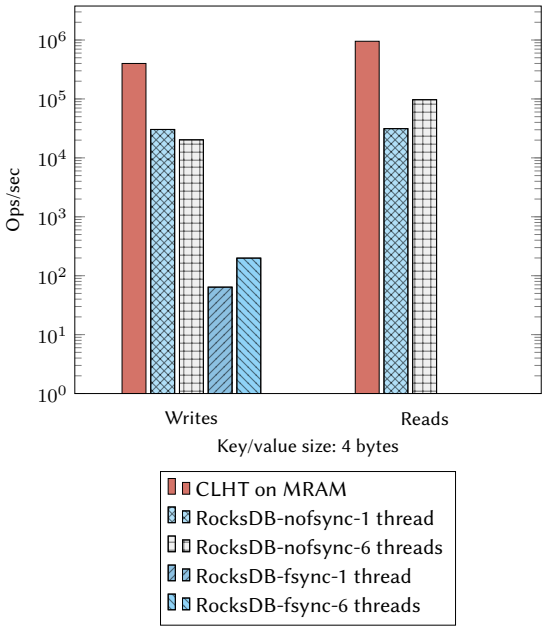

Figure 3: Comparison between CLHT running on MRAM and RocksDB running on NAND FLASH in RPi.

It is interesting to note that RockDB's performance for keys/values with 2 bytes is significantly better than for the remaining sizes. The most likely reason for this is the big number of duplicate values present in the dataset used for this particular experiment, which allows RocksDB to easily keep all values in its block cache. We also note that for key/value sizes above the previously stated, RocksDB is able to perform more read operations per second than LPHT.

In both cases, the performance of LPHT declines at a faster rate than RocksDB as key/value size increases. This can be due to the fact that as keys get bigger, the computational effort to compute its hash value increases. Since the STM32 has less computational power than the RPi, this effect will be more noticeable.

5.2.2. CLHT vs RocksDB

Figure 3 shows how CLHT compares to different configurations of RocksDB while inserting key/value pairs with 4 bytes each. When compared to RocksDB running without the fsync option and a single thread (the best scenario for the no fsync configuration), CLHT is able to perform 11× more write operations per second. Compared to RocksDB's best scenario with fsync being enforced, where RocksDB uses 6 threads, which is also the scenario that most closely resembles the persistence that MRAM provides with each write, CLHT is able to perform 1827× more write operations per second. In terms of reads, MRAM outperforms RocksDB with 6 threads by 9×.

We conclude that the raw performance advantage of MRAM over NAND Flash translates into a significant advantage in key-value systems, especially for smaller key-value sizes. For this use case, trading computational power for storage performance is the correct approach, indicating that the main bottleneck of these systems is indeed the FLASH storage device.

5.3. Impact on relational database system

Finally, on a more complex scenario, SQLite's performance is compared when running on the STM32 over MRAM, and on the RPi. This allows for a comparison of the same exact system across the two platforms. The results of running SQLite on the STM32 with a custom OS layer are compared against SQLite running on the RPi with NAND FLASH (using the default UNIX OS layer).

Testing methodology For SQLite, a schema consisting of a single table representing a sensor is used. The table consists of four columns of the integer type: *timestamp*, *device_id*, *zone*, and *pressure*. Each insert operation inserts a new record which increments the timestamp of the previously inserted record, and generates random values for the remaining columns. Each run reads or writes a total of 5000 rows, but the number of inserted values or selected rows per transaction varies. For example, in the first test 5000 transactions are executed, with a single read or write operation being performed in each transaction. For the last test, however, only 50 transactions are performed, with 100 values being selected or inserted in each transaction. Results depict an average of five independent runs, and all SQLite files are deleted between runs. SQL queries are generated prior to the test, so that throughput estimation is not affected by the time spent generating those queries. STM32's L1 data cache is enabled for all SQLite experiments enforcing a write-through policy.

Figure 4 shows the number of rows either inserted or read per second, in relation to the number of rows affected by a single transaction. Unlike in the case of key-value stores, it is not enough to perform a direct swap of the storage medium from NAND Flash to MRAM for the STM32 to outperform the MPU based (RPi) in a relational database scenario. That is because relational databases impose a greater computational overhead, thus giving the advantage to the more capable RPi. Even so, the greater performance of MRAM for small write operations enables the STM32 to achieve a performance that is close to that of the RPi for insert transactions affecting very few rows. For the experiment where each write transaction performs only two insert operations, the RPi outperforms

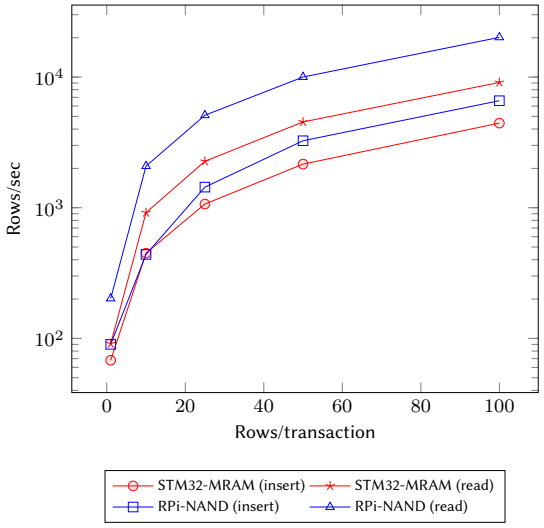

Figure 4: Comparison between SQLite running on: STM32's MRAM and RPI's NAND FLASH.

the STM32 by only $1.02\times$. As the number of insert statements per transaction increases, SQLite performs bigger I/O operations which decreases the performance gap of the two storage media, and allows the RPi to perform up to $1.48\times$ more insert operations per second than the STM32. In the case of select operations, the RPi is able to read around $2.21\times$ more rows per second than the STM32 across all types of transactions. We conclude that for relational databases MRAM can assist an MCU achieve a similar performance to an MPU for basic operations while consuming less energy and having less computational resources. However, a direct substitution of the storage media is not sufficient for the MCU to outperform the MPU. One possible way to further improve performance for SQLite in the MCU would be to shed the additional computational overhead that is imposed by the FLASH oriented mechanisms such as the wear-leveling mechanism in LittleFS, and the Write-Ahead Log in SQLite.

6. Discussion

The first conclusion to draw from this work is that MRAM provides a big advantage in small I/O operations. MRAM adoption can be particularly interesting for key-value applications, such as edge Time-Series Databases (TSDBs) and Key-Value Stores, which often handle small key/value pairs [14]. Furthermore, MRAM can provide strong consistency guarantees, since all write operations are immediately persisted. As depicted in Figure 2, the impact of using *fsync* (i.e., persisting every write) with FLASH memory is significant. Thus, critical applications in sensor networks, i.e., smart health care or industrial IoT, might benefit considering this technology.

MRAM imposes less computational overhead on systems, as it does not require wear-leveling, batching, or sequential ordering mechanisms which are often used by FLASH based systems. This opens the way to lowering systems' complexity when using MRAM.

This can be of special importance for relational database systems in resource constrained devices. In such computationally limited devices, MRAM allows forgoing FLASH focused mechanisms, freeing computational capacity that can instead be used by the DBMSs query engine. This can help support the ongoing effort to enable more features in MCU relational databases, since current options have to severely limit the number of supported features in order to fit available resources [7, 6]. Furthermore, these MCUs provide additional resources such as: Direct Memory Access controllers (DMAs) that enable data to be moved between storage devices without CPU intervention; and dedicated hashing controllers that can calculate hash values also without CPU intervention; which can be explored to further increase database performance while putting less load on the CPU. In the case of key-value systems MRAM could enable more functionality to be shifted from MPU devices to MCU devices, while still improving MCU battery lifetime. For example, in the case of wearable sensors, data has to be uploaded in its entirety to a more capable MPU to calculate statistics on the gathered data (e.g., [11, 43]) due to lack of CPU power, which coincidentally increases the amount of data transmitted, increasing the rate at which the sensors' battery is drained. By consuming a lower amount of computational capacity MRAM can allow the MCU to make those calculations locally, thus only transmitting the already processed data. This data will be smaller, and allow the MCU to conserve more energy by moving the load on the MCU towards computation in place of data transmission. MRAM will be specifically appealing in scenarios where key-values are small (i.e., small write operations), the most common occurrence in key-value systems [14], and where said data needs to be persisted. This may be a requirement for critical systems such as those involving medical scenarios or public services management (e.g., smart grid applications).

Both solutions share the same price bracket, however in our approach CPU is traded for memory performance. We believe this to be the correct choice for the case of edge databases, since storage is the primary bottleneck. However, we must take into account that MRAM has a significantly lower storage capacity per chip (up to 8Gb per chip [44]) and a greater price per space unit. In total, the STM32 used in this work could directly support up to 512Mb of MRAM memory. As such, the main contribution given by MRAM to edge systems, at the moment, is not in storage capacity, which is the case for FLASH,

but rather in performance, energy expenditure, and endurance. As such, a hybrid approach could provide the best of both worlds (i.e., MRAM and FLASH). MRAM could be combined with more conventional FLASH storage, e.g, an SD Card, to achieve both better performance and durability, while still ensuring a large amount of storage space. With the perspective of decreasing prices (see Section 3), MRAM only storage may also be a possibility in the future. MRAM memory may also pave the way to instant recoverability if used as an alternative to non-persistent program memory. Energy-wise, the considered MRAM setup has a power profile 10× smaller when compared with the NAND FLASH which provides a positive impact for edge applications.

We hypothesize two use cases for MRAM use, to better clarify how this technology can benefit edge data management systems.

Relational database use case - Picture a scenario where each sensor runs its own relational database over FLASH (e.g., [6]). At any given moment, a sensor may be queried for its data, however it is limited to only a few operations, such as select, update, delete and insert operations, or simple join operations. More complex operations, such as nested queries are not supported, due to a lack of CPU power which would make the time to complete the query unacceptable. Thus, the client must issue only the innermost select query, and process the received data locally, possibly requiring further queries to complete the original query. This means that more data will be transmitted to the client than the data needed to answer the original query, therefore more energy will be used by the MCU.

Now, replace the storage device for either MRAM only, or a hybrid MRAM and FLASH solution. MRAM having a lower management complexity frees up part of the computational budget, which can now be used by the query engine to support faster processing. Furthermore, faster performance means less I/O waiting time, equating to less unused processor cycles. With the extra computational budget attributed to the query engine we are now able to support nested queries. By executing the entire query in one go only the minimum required amount of data is transmitted to the client, optimizing the amount of energy used.

Key-value use case - Picture a scenario where a patient wears an MCU based and battery powered sensor, that takes heart related measurements. Storing and processing the data locally using FLASH storage would be too computationally expensive for the MCU, so instead those measurements are transmitted in raw form to a more capable MPU, where ECG data is extracted from the raw data. This transmission of data drains the MCUs battery, requiring frequent recharging of the medical sensor device. If instead MRAM storage was used, the MCU could potentially have enough processing power left to extract the ECG data locally, and only relay relevant information to the MPU, extending operational lifetime of the charge cycle.

The conducted experiments used an M1 MRAM device (Table 1), as it allowed to create a prototype in a shorter time frame. Employing faster M3 or M4 devices could potentially increase the observed performance, which we reserve for future work.

Similar to MRAM, there are a series of other persistent memory technologies which can be considered for use with database systems. We consider the comparison of MRAM against other types of persistent memory to be outside the scope of this work, but we encourage interested parties to check on related work which provides that analysis [45]. As for how previous work with the popular Intel Optane persistent memory can be applied to MRAM, we believe there are multiple reasons why such work may not be applicable here. The Intel Optane line is composed of more complex devices which are composed of multiple data storage chips, with non-persistent caching mechanisms and capability for concurrent operations. Related work in Intel Optane enabled key-value stores, for example, focuses on providing consistency guarantees given non-persistent write operations (i.e., involving caching) and maximizing concurrency related performance [14, 42]. Some optimizations are also based on optimizing the use of the libraries provided for Intel Optane access. In contrast, databases for MCUs, as analyzed here, have a single execution thread. Furthermore, the targeted MRAM device does not support concurrent operations and does not provide caching mechanisms. The MRAM memory is accessed in the same way as normal memory: through a pointer to a particular address which is mapped to a location in the MRAM memory. As such, the set of problems for systems targeting Intel Optane is not the same as for MRAM systems.

7. Conclusion

Research for the use of persistent byte-addressable memory for database systems has been focused on data center-scale applications, namely supported by Intel Optane products, [46, 47]. Results show, however, that byte addressable persistent memory should also be explored for use in resource constrained data management systems.

This paper shows that MRAM provides several advantages over NAND FLASH alternatives. At the hardware level, MRAM enables 5 orders of magnitude more write operations per cell, thus making it practically impervious to cell wear-out. Furthermore, random and sequential accesses have identical performance, and maximum

throughput is achieved with writes as small as 4 bytes, and reads of 512 bytes.

MRAM shows a throughput advantage on all I/O block sizes when compared to FLASH, particularly for *block sizes* under 32KB. This was observed in the *Raw Performance* tests, but also for the Hash Table tests, despite being a more complex workload and with the exception that for key/value sizes greater than 32 bytes, RocksDB evaluation with the NAND Flash alternative outperforms MRAM's LPHT. The relational database test with SQLite showed that although MRAM can help MCUs reach a performance close to that of an MPU for a relational database, a direct replacement of NAND FLASH for MRAM is not sufficient for the MCU to outperform the MPU. However, MRAM allows for a lot of the mechanisms that are currently used to accommodate FLASH to be avoided, opening new architectures directed specifically at MRAM to outperform MPUs.

In a nutshell, MRAM presents a big advantage over NAND FLASH in small I/O operations, being able to achieve full throughput at operation sizes of just a few bytes. Furthermore, performance is not affected by random access patterns. The virtually infinite endurance of MRAM memory avoids the need for any wear-leveling mechanisms, and its low power consumption contributes to extend the lifetime of battery powered MCUs. Nominal values also point to MRAM being able to achieve a significantly higher peak throughput than FLASH storage. Thus, MRAM can allow for systems which are simpler to implement, have higher performance, and consume less energy.

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857237. The sole responsibility for the content on this publication lies with the authors. It does not necessarily reflect the opinion of the European Commission (EC). The EC are not responsible for any use that may be made of the information contained therein. It is also funded by National Funds through the FCT — Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) PhD grant (PD/BD/151402/2021).

References

[1] D. Reinsel, J. Gantz, J. Rydning, Data Age 2025: The Evolution of Data to Life-Critical. Don't Focus on Big Data; Focus on Data That's Big., Technical Report, Int. Data Corporation (IDC), 2017.

[2] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, J.-P. Talpin, Femto-containers: Lightweight virtualization and fault isolation for small software functions on low-power iot microcontrollers, in: Proceedings of the 23rd ACM/IFIP International Middleware Conference, 2022, pp. 161–173.

[3] K. Choi, H. Han, H. Jung, S. Kang, Workload-optimized sensor data store for industrial iot gateways, Future Generation Computer Systems 135 (2022) 394–408.

[4] A. Ravindran, A. George, An edge datastore architecture for latency-critical distributed machine vision applications., in: HotEdge, 2018.

[5] H. Gupta, U. Ramachandran, Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access, in: Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, 2018, pp. 148–159.

[6] N. Tsiftes, A. Dunkels, A database in every sensor, in: Proceedings of the 9th ACM Conf. on Embedded Networked Sensor Systems, 2011, pp. 316–332.

[7] G. Douglas, R. Lawrence, Littled: a sql database for sensor nodes and embedded applications, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014, pp. 827–832.

[8] C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, Picodbms: Scaling down database techniques for the smartcard, in: VLDB, 2000, pp. 11–20.

[9] Y. Yao, J. Gehrke, The cougar approach to in-network query processing in sensor networks, ACM Sigmod Record 31 (2002) 9–18.

[10] S. R. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong, Tinydb: An acquisitional query processing system for sensor networks, ACM Transactions on Database Systems (TODS) 30 (2005) 122–173.

[11] A. Rahmani, T. Gia, B. Negash, A. Anzanpour, I. Azimi, M. Jiang, P. Liljeberg, Exploiting smart e-health gateways at the edge of healthcare internet-of-things: A fog computing approach, Future Generation Computer Systems 78 (2018) 641–658.

[12] D. Saxena, V. Raychoudhury, C. Becker, An ndnot based efficient object searching scheme for smart home using rfids, in: Proceedings of the 18th Int. Conf. on Distributed Computing and Networking, ACM, Hyderabad, India, 2017, pp. 1–6.

[13] M. Buevich, A. Wright, R. Sargent, A. Rowe, Respawn: A distributed multi-resolution time-series datastore, in: IEEE 34th Real-Time Systems Symp., IEEE, Vancouver, Canada, 2013, pp. 288–297.

[14] B. Lu, X. Hao, T. Wang, E. Lo, Dash: Scalable hashing on persistent memory, Proc. VLDB Endow. 13 (2020) 1147–1161. URL: https://doi.org/10.14778/3389133.3389134. doi:10.14778/3389133.3389134.

[15] T. Instruments, Am62x sk evm user's guide, 2023. URL: https://www.ti.com/lit/ug/spruj40c/

spruj40c.pdf?ts=1687780680209, rev. C.

[16] M. Li, D. Ganesan, P. Shenoy, Presto: Feedback-driven data management in sensor networks, IEEE/ACM Transactions on Networking 17 (2009) 1256–1269.

[17] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. McGrail, P. Wang, et al., Apache iotdb: Time-series database for internet of things, Proc. VLDB Endow. 13 (2020) 2901–2904.

[18] F. Wu, C. Qiu, T. Wu, M. Yuce, Edge-based hybrid system implementation for long-range safety and healthcare iot applications, IEEE Internet of Things Journal 8 (2021) 9970–9980.

[19] S. Alamouti, F. Arjomandi, M. Burger, Hybrid edge cloud: A pragmatic approach for decentralized cloud computing, IEEE Communications Magazine 60 (2022) 16–29.

[20] aws, What's the difference between microprocessors and microcontrollers?, 2023. URL: https://aws.amazon.com/pt/compare/the-difference-between-microprocessors-microcontrollers/.

[21] M. Technology, Mt29f128g08ajaaawp-itz:a, nand flash memory, rev. h, 2014. URL: https://pt.mouser.com/datasheet/2/671/micron_technology_micts06235-1-1759187.pdf.

[22] Intel, 3d xpoint™: A breakthrough in non-volatile memory technology, 2015. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[23] J. Heidecker, MRAM Technology Status, Technical Report, NASA, 2013.

[24] A. Technology, As3004316, parallel persistent sram memory, rev. t, 2022. URL: https://pt.mouser.com/datasheet/2/1122/1Mb_32Mb_Parallel_x16_MRAM_2-1949428.pdf.

[25] E. Technologies, Mr4a16b, rev. 11.7, 2018. URL: https://pt.mouser.com/datasheet/2/144/MR4A16B_Datasheet-1511254.pdf.

[26] E. Technologies, Emxxlx, expanded serial peripheral interface (xspi) industrial stt-mram persistent memory, rev. 2.9, 2022. URL: https://www.everspin.com/supportdocs/all, rev. 2.9.

[27] E. Technologies, Emd4e001gas2, 1gb non-volatile st-ddr4 spin-transfer torque mram, rev. 1.2, 2020. URL: https://www.mouser.com/datasheet/2/144/EMD4E001GAS2_1_2_08252020-1923803.pdf.

[28] M. Technology, Mt28ew512aba1hpc-0sit tr, parallel nor flash embedded memory, rev. i, 2018. URL: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/nor-flash/parallel/mt28ew_mt28fw/mt28ew_qlkp_512_aba_0sit.pdf.

[29] A. Technology, Avalanche technology - products - space grade, 2023. URL: https://www.avalanche-technology.com/products/discrete-mram/space.

[30] T. David, R. Guerraoui, V. Trigonakis, Asynchronized concurrency: The secret to scaling concurrent search data structures, ACM SIGARCH Computer Architecture News 43 (2015) 631–644.

[31] E. Distributed Computing Laboratory, Clht, 2013. URL: https://github.com/LPD-EPFL/CLHT.

[32] SQLite, The sqlite os interface or "vfs", 2023. URL: https://www.sqlite.org/vfs.html.

[33] littlefs project, littlefs, 2023. URL: https://github.com/littlefs-project/littlefs.

[34] STMicroelectronics, Stm32h743zi, 2023. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32h743zi.html.

[35] SanDisk, Sandisk extreme® microsdxc™ uhs-i card, 2023. URL: https://www.westerndigital.com/products/memory-cards/sandisk-extreme-uhs-i-microsd#SDSQXAF-032G-GN6MA.

[36] G. P. Perrucci, F. H. P. Fitzek, J. Widmer, Survey on energy consumption entities on the smartphone platform, in: 2011 IEEE 73rd Vehicular Technology Conference (VTC Spring), 2011, pp. 1–6. doi:10.1109/VETECS.2011.5956528.

[37] SanDisk, Sandisk® industrial microsd card datasheet, 2016. URL: https://images-na.ssl-images-amazon.com/images/I/91tTtUMDM3L.pdf.

[38] S. Crawford, How secure digital memory cards work, 2011. URL: https://computer.howstuffworks.com/secure-digital-memory-cards.htm.

[39] Samsung, Microsd pro endurance, 2023. URL: https://semiconductor.samsung.com/consumer-storage/memory-card/micro-sd-pro-endurance/.

[40] J. Axboe, Fio-flexible io tester, 2014. URL: https://github.com/axboe/fio.

[41] A. Piltch, Best microsd cards for raspberry pi 2023, 2023. URL: https://www.tomshardware.com/best-picks/raspberry-pi-microsd-cards.

[42] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, V. Chidambaram, Recipe: Converting concurrent dram indexes to persistent-memory indexes, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 462–477.

[43] H. Dubey, J. Yang, N. Constant, A. M. Amiri, Q. Yang, K. Makodiya, Fog data: Enhancing telehealth big data through fog computing, in: Proceedings of the ASE bigdata & socialinformatics 2015, 2015, pp. 1–6.

[44] A. Technology, As308g208, space-grade high performance dual-quad serial persistent sram memory, rev. e, 2023. URL: https://www.avalanche-technology.com/wp-content/uploads/1G-8Gb-Dual-QSPI-Space-Grade-Serial-E-01_10_2023.pdf.

[45] S. Kargar, F. Nawab, Challenges and future di-

rections for energy, latency, and lifetime improvements in nvms, Distributed and Parallel Databases (2022) 1–27.

[46] A. Shanbhag, N. Tatbul, D. Cohen, S. Madden, Large-scale in-memory analytics on intel® optane™ dc persistent memory, in: Proceedings of the 16th International Workshop on Data Management on New Hardware, 2020, pp. 1–8.

[47] Y. Wu, K. Park, R. Sen, B. Kroth, J. Do, Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms, in: Proceedings of the 16th International Workshop on Data Management on New Hardware, 2020, pp. 1–3.