Inducing Huge Tail Latency on a MongoDB deployment

Remo Andreoli
Sant'Anna School of Advanced Studies
Pisa, Italy
remo.andreoli@santannapisa.it

Tommaso Cucinotta
Sant'Anna School of Advanced Studies
Pisa, Italy
tommaso.cucinotta@santannapisa.it

***Abstract*—The NoSQL paradigm has emerged as the leading design choice for cloud providers offering highly scalable storage services. Contrary to traditional relational databases, NoSQL architectures are capable of ingesting the ever-growing volume of nowadays' data-driven applications characterized by low-latency and high-throughput requirements. However, it is difficult to build an ultra-scalable, high-performance storage engine that can sustain an arbitrary number of concurrent clients. A common technique to increase throughput is minimizing the OS overhead, quantified as the number of context switches, through busy waiting (or "spinning"). While this simple synchronization mechanism proves to be beneficial in the high-performance computing community, it requires special care to avoid wasting resources and counter-intuitive behaviors. In this paper, we address an instance of "unsafe" busy waiting in WiredTiger, the underlying storage engine of MongoDB, which leads to a consistent, excessive increase of tail latency in high contention scenarios.**

***Index Terms*—Performance Engineering, Cloud Storage, WiredTiger, MongoDB, Concurrency, Scheduling**

I. Introduction and Related Work

Cloud Computing has deeply changed how the IT industry deals with the technical challenges posed by today's data-driven applications [20]. Depending on the customer's needs, there are many different cloud-based storage services, each with its own focus area: from distributed data management systems for scaling operations in web services, to highly reliable data processing platforms available around-the-clock for Internet-of-Things (IoT) devices. In this context, the emergence of cloud-native applications and fully-managed services [12] shifted the entire management cycle of applications and virtual resources from the customer itself to the operations teams of the cloud provider.

A relevant metric to assess the performance of a cloud service is the *tail latency* [11], [17]. It encompasses the high percentiles of the response-time distribution, and usually, it focuses on the requests experiencing response times longer than 99% (or more) of all user requests. A poor tail latency will impact most users in the case of large-scale applications with modern architectures, such as microservices, especially when a single user interaction can translate into many service calls. Improving the tail latency is a well-known performance engineering challenge that spans multiple layers of the cloud software stack, and it has been addressed by academics in different ways [3], [7], [16], [18], [23]. The majority of existing research efforts focus on high-level abstractions, neglecting the lower end of the cloud stack, such as the Operating System (OS) layer [10]. In parallel and distributed cloud components, this leads to a general lack of integration with low-level mechanisms, such as tuned scheduling policies [9], [13], [22] or efficient synchronization constructs beyond traditional lock-based primitives, or contention control techniques [25]. Nonetheless, industrial-grade database software often employs low-level optimizations to fully exploit the capabilities of modern many-core machines. For instance, WiredTiger, the underlying storage engine of MongoDB, implements a variety of lock-free algorithms and busy waiting to increase the throughput on modern many-core machines [2].

Busy waiting [14], or spinning, is a lightweight synchronization technique, mainly used by High-Performance Computing (HPC) communities to minimize the OS overhead. Busy waiting is a quick and simple mechanism in which a process, or a thread, continuously checks in loop for a condition to be satisfied while consuming processing resources (hence why "busy" waiting). It is used when a process/thread (referred to as *task*, from now on) is likely to wait for short periods of time to access the critical section; shorter than the delay associated with a context switch. Contrary to conventional blocking synchronization primitives (i.e., mutexes, semaphores, condition variables), which require interactions with the OS kernel to block tasks, busy waiting can be done fully in user-space, avoiding the need for context switches.

Busy waiting may lead to excessive waste of CPU time, if misused, due to the costly looping procedure. The results are unexpected scaling issues and performance degradation in the form of high tail latency, task starvation, or even deadlock. Given a task A waiting for a condition that can be made true by a running task B, an instance of busy waiting is "safe" when the task being waited for (task B) is not pre-empted. For example, busy waiting in a spinlock within the Linux kernel is safe, because task pre-emption can be disabled. However, this is not possible in user-space, therefore one of the most common ways to ensure that tasks are unlikely pre-empted is to bind them with dedicated cores, enforcing a 1:1 task-to-core pinning. This constraint is usually irrelevant in an HPC context, as the computations are usually deployed on custom-built hardware with hundreds of physical cores. However, core pinning is not so commonly enforced in conventional cloud infrastructures, i.e., it is used only by performance-sensitive applications. In summary, busy waiting is "unsafe" in user-space if performed when the number of active tasks exceeds the parallelism capabilities of the underlying hardware.

A. Contributions

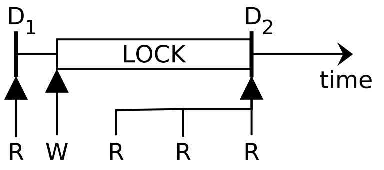
In this work, we describe the internals of WiredTiger and explore a synchronization bug in MongoDB caused by an "unsafe" implementation of busy waiting within WiredTiger (Section II). A series of possible quick fixes are presented; the best of which is an "adaptive" implementation of busy waiting that ensures safety by spinning for a short while only. Finally, Section III demonstrates how to consistently reproduce the bug using two common system tools within Linux, `taskset` and `nice`, as well as showing the superiority of adaptive busy waiting over the original implementation. In practice, it is difficult to test for concurrency misbehaviors: a bug in the synchronization logic may sit latent for many months (or years) and inevitably manifests itself in production [19], as a side effect of heavy loads, in the form of unexpected performance degradation, starvation or even deadlock. Moreover, most of the time it is not possible to consistently reproduce it in a test environment. Therefore, one of the goals of this research paper is to prove that more focus on system research can significantly reduce the risk of unexpected behaviors when dealing with low-level optimizations.

II. WiredTiger internals

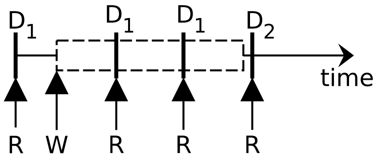
WiredTiger[1] is a high-performance, scalable, and transactional storage engine for NoSQL data stores. It is primarily known for being the underlying database management system of MongoDB[2] since version 3.2. WiredTiger offers both high-throughput and low-latency, as well as predictable behaviors under heavy access and large volumes of data. WiredTiger is designed to efficiently scale on modern many-core machines with lots of RAM. This is thanks to several compression algorithms to save both in memory and disk consumption, as well as the use of high-performance techniques and lock-free algorithms to minimize the resource contention between concurrent transactions. The main focus of this section is exploring how WiredTiger optimizes the synchronization of concurrent and parallel transactional operations for the sake of high-performance. A WiredTiger-compliant user application establishes a connection with an instance of WiredTiger and starts sending requests using a session. A user session is always executed as a sequential activity by one thread, although it can be shared between threads. WiredTiger offers a standard ACID-style transactional model [15]. A transaction represents an atomic unit of work requested by a user session. It may consist of multiple data manipulation operations. Each transaction is given a global, unique, and monotonically increasing identifier (ID) before performing the first write operation. Consistency is enforced through an "optimistic" version of the classic Multi-Version Concurrency Control (MVCC) mechanism [6], which completely avoids the bottleneck of a centralized lock

[1]See: https://source.wiredtiger.com/

[2]https://www.mongodb.com/



(a) Read operations are blocked until the write transaction on data item D is successfully committed.



(b) Read operations get an older version of data item D, until the write transaction is successfully committed.

Fig. 1: Traditional lock-based (a) versus Multi-versioned (b) concurrency control in the presence of concurrent write (W) and read (R) operations. The plots show how data item D is handled over time. The subscripts indicate the subsequent changes to data item D.

manager. In traditional MVCC, an update operation does not overwrite the original data item but instead creates a newer *version* of such data item. This way reads do not block writes, and vice versa. Figure 1 demonstrates the difference between traditional lock-based and multi-version concurrency control. The term "optimistic" refers to the fact that WiredTiger assumes that multiple transactions can frequently complete without interfering with each other, regardless of the operation type. Therefore, this implementation of MVCC is essentially lock-free, since transactional operations in one session do not block operations in other sessions, even if they are writes. Notice that if multiple concurrent transactions update the same data item, only one is committed and the others must be repeated. WiredTiger is the reason for the improved write performance in MongoDB [1] since its initial introduction in version 3.0. Compared to the previous storage engine, MMAPv1, WiredTiger allows for document-level concurrency control in MongoDB, meaning that multiple write operations to different documents, but on the same collection, can occur at the same time. WiredTiger presents to each transactional operation a point-in-time consistent view of the in-memory data, called a *snapshot*. The version of data that each transaction sees depends on the isolation level. The strongest guarantee is *snapshot isolation*: all reads within a transaction will see a consistent snapshot of the database; no updates within a transaction will be committed if they conflict with any concurrent updates made since that snapshot. In practice, this is implemented with *timestamp*s, a monotonically increasing sequence of numbers associated with each operation: a transaction can only see updates with timestamps smaller or equal to its read timestamp. Snapshots are implemented by capturing the global state of transactions at the time of snapshot creation.

Transactions that are concurrently active are not visible to the snapshot, as they do not comply with the snapshot isolation guarantee (i.e., the transaction has not been committed at the time of snapshot creation). Snapshots are periodically flushed to disk to act as recovery points (*checkpoint*s), thus ensuring data durability in case of failure.

A. "Unsafe" Busy waiting

WiredTiger cannot publish a new snapshot until all concurrently active transactions are assigned a valid ID, or else it will not be able to infer if they are visible to the snapshot. To this end, it employs busy waiting to increase throughput, since ID allocation is assumed to be a fast operation; faster than yielding the CPU or "sleeping" as in conventional blocking synchronization primitives, which involve a series of context switches. However, the way WiredTiger is used within MongoDB makes busy waiting unsafe, as the rules recalled in Section I are not enforced. Indeed, each remote connection to a MongoDB server is reserved a unique dedicated thread to handle the server-side activities [4], [5], which are ultimately performed by WiredTiger. Therefore, MongoDB should have controlled/restricted the number of clients concurrently issuing requests to avoid the potential waste of CPU time, as they result in threads concurrently using WiredTiger. However, MongoDB designers deemed such restriction too binding and counterproductive to the need to serve potentially arbitrary workloads in a cloud usage context, as mentioned in Section I.

In the context of a DBaaS cloud offering, such as MongoDB Atlas, it is not clear to the authors of this paper whether the number of concurrent connections to a single MongoDB instance is controlled, or it can also exceed the number of underlying physical cores, leading to potentially high tail-latency problems.

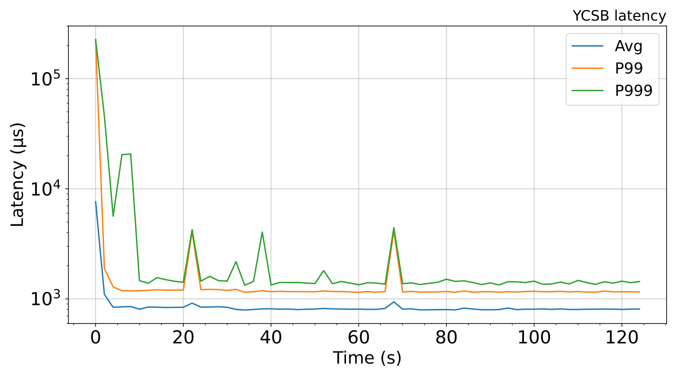
Additionally, in the presence of threads with different priorities, the synchronization problem described above results in a nasty priority inversion problem. Let T_a be a WiredTiger thread trying to create a snapshot to apply the operations required by its user session. Let T_b be a different thread trying to initialize a new transaction. If both threads happen to be ready-to-schedule simultaneously, T_a has to busy wait on thread T_b, if it did not allocate its transaction ID yet. Recall that without such a mechanism, a consistent view of the data cannot be guaranteed. If there are not enough free physical cores to run both threads in parallel, and T_a is given precedence over T_b in the scheduling queue, T_a will spin uselessly until it exhausts its time-slice, effectively starving thread T_b. When T_b is subsequently scheduled for execution (i.e., exiting starvation), and completes the ID allocation procedure, the spinning condition for T_a is finally satisfied, and therefore it can proceed. Therefore, the spinning duration directly depends on the scheduling decisions. Section III demonstrates how to consistently induce such thread synchronization bug exploiting 2 common system tools available on Linux: `taskset` and `nice`. The `taskset` command allows restraining the scheduling of a task to a subset of the available cores. The `nice` command manipulates the scheduling priority of a task in the Completely Fair Scheduler (CFS) [24], the default scheduler within Linux. The "niceness" of a thread corresponds to its willingness to give precedence to other threads, which ultimately affects the CPU time of each task scheduled for execution. There are a total of 40 nice levels in the range $[-20, 19]$, where the negative values correspond to less willingness to give up CPU time.

B. Restoring safety in WiredTiger

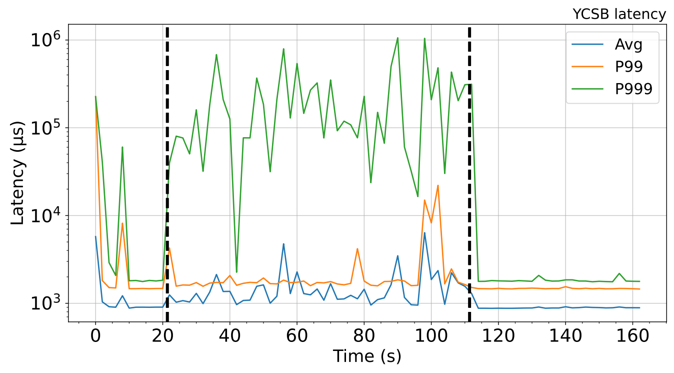
There are multiple, straightforward fixes to the unsafe instance of busy waiting in WiredTiger. The "safer" solution is to replace busy waiting with conventional blocking synchronization, at the cost of more OS overhead due to context switches. Not only that, but it also requires a more involved modification due to the wait-signal synchronization model for inter-thread communication. A quicker and easier solution is to yield the CPU so that the spinning thread relinquishes the CPU to another thread. However, the former may be immediately rescheduled for execution if CFS decides that the spinning thread was not given a "fair" amount of CPU time, or if no other threads are in the ready queue. A better solution is to put the spinning thread to sleep for a fixed amount of time. Contrary to locks, sleeping is more lightweight (i.e., less OS overhead), being a simpler mechanism, but it requires proper tuning of the sleep duration. A good compromise, in line with the principles of WiredTiger, is an "adaptive" version of busy waiting which encapsulates a backoff procedure: spin for a fixed amount of time, then yield for a while, and then back off to sleep. WiredTiger already implements such a mechanism in `__wt_spin_backoff`; although it is used in a different context. Section III demonstrates how adaptive busy waiting is a valid solution to the synchronization problem during snapshot creation and transaction ID allocation in WiredTiger.

III. Experimental Evaluation

This section shows how to consistently starve of CPU time the WiredTiger threads dedicated to the client sessions of a MongoDB deployment, using the `nice` and `taskset` tools in Linux. More specifically, the experiments have been performed on MongoDB version 6.0, which employs WiredTiger version 10.0.2. We also demonstrate the validity of the adaptive busy waiting mechanism described in Subsection II-B by repeating the experiments on a modified version of WiredTiger. Notice that concepts like data fragmentation (for scalability) and data replication (for availability) are MongoDB-level constructs that do not affect the behaviors of the underlying storage engine. Instead, they add complexity to the testing environment; hence why, for the sake of simplicity, replication and sharding have not been used. The data store is subjected to heavy write load using the well-known YCSB [8] benchmarking tool. The YCSB client threads are hosted on a dedicated, 96-core physical system (Arm 64 server with 2 ThunderX 88XX CPUs and 64 GB of RAM) connected to the MongoDB deployment via a 1 gbE physical link. The latter is hosted on a 112-core physical system (x86-64 server with 2 Xeon Gold CPUs and 125 GB of RAM) to ensure no

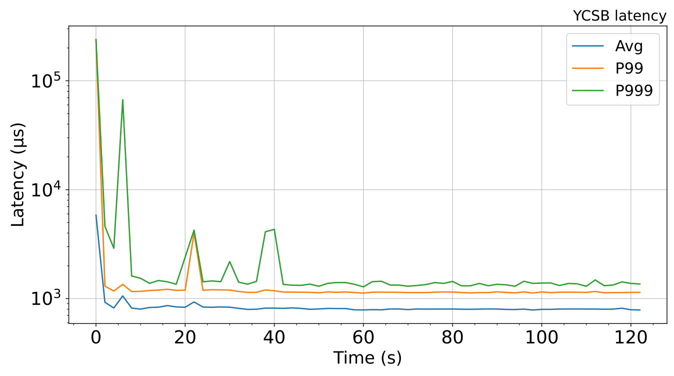


(a) Unrestricted (i.e., no `taskset`) MongoDB deployment.

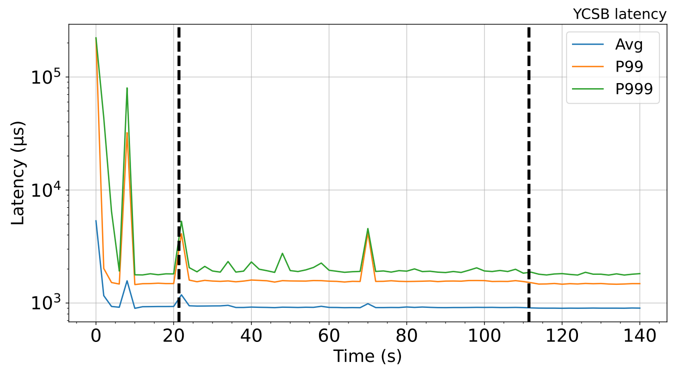


(b) MongoDB deployment with session threads restricted to 10 cores (the thread-core ratio is 4:1). The thread priority of 4 user sessions is temporarily changed during the time window highlighted by the two dashed lines.

Fig. 2: Per-second statistics of the end-to-end latency experienced by 40 YCSB clients issuing 6 million update operations to a MongoDB deployment using an unmodified version of WiredTiger (i.e., original busy waiting).



(a) Unrestricted (i.e., no `taskset`) MongoDB deployment.



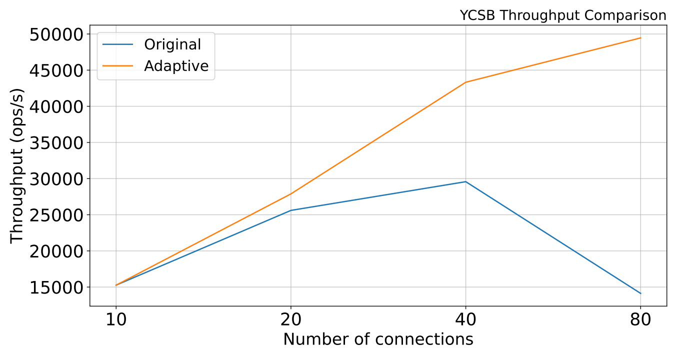
(b) MongoDB deployment with session threads restricted to 10 cores (the thread-core ratio is 4:1). The thread priority of 4 user sessions is temporarily changed during the time window highlighted by the two dashed lines

Fig. 3: Per-second statistics of the end-to-end latency experienced by 40 YCSB clients issuing 6 million update operations to a MongoDB deployment using a modified version of WiredTiger (i.e., busy waiting with backoff).

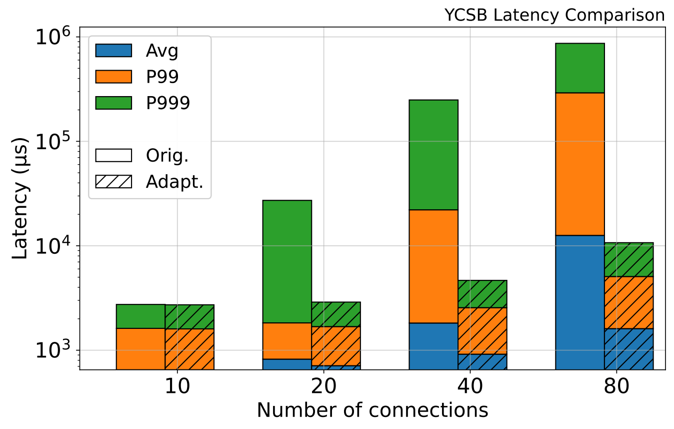
interferences between the server and the clients. On the 112-core machine, CPU frequency is blocked at 2.20 GHz, and hyper-threading and turbo-boosting are disabled to minimize the experimental error. In order to violate the condition of "safe" busy waiting, the MongoDB deployment has to be subjected to a high CPU contention (i.e., the number of user connections must exceed the number of physical cores dedicated to the database). The easiest way to achieve that is through the `taskset` Linux command so that the WiredTiger threads dedicated to the user sessions are restricted to a small range of physical cores. Next, the priority inversion is induced by tweaking the scheduling parameters of such threads using the *nice* command.

The first series of experiments aim to visually show how latencies are affected by the use of thread priorities under a high contention scenario. Figure 2 depicts the latency over time for two experiments using an unmodified version of WiredTiger: Subfigure 2a shows a MongoDB deployment with no core restriction to session threads (i.e. no `taskset`) and no tweaking of thread priorities; Subfigure 2b shows a MongoDB deployment with session threads restricted to 10 physical cores and the niceness of 4 session threads temporarily set to -20. Each test has been performed 3 times to assess the variability of the experimental results. Figure 3 depicts analogous experiments using a modified version of WiredTiger, implementing the adaptive busy waiting mechanism described in Subsection II-B. In our test environment, the latencies experienced by users interacting with an unrestricted MongoDB deployment, using the original version of WiredTiger, is 890 microseconds on average (Subfigure 2a). The modified version of WiredTiger achieves similar results, showing no noticeable side effects in replacing the original implementation with the adaptive one. However, the fallacy of the original instance of busy waiting is demonstrated in Subfigure 2b: the use of `taskset` and `nice` affects the 99.9^{th} percentile ($P999$) the most, showing a $\sim$2168% increase, with peaks of 1 second during the thread priority time window (highlighted by dashed lines). On the other hand, Subfigure 3b, which depicts the latency over time using the adaptive busy waiting solution, shows a much lower $\sim$7% increase of the $P999$ latency instead (with respect to the unrestricted case in Subfigure 3a).

The second series of experiments (Figure 4) compare the



(a) Throughput comparison.



(b) Latency comparison.

Fig. 4: Performance comparison between the two busy waiting implementations on different CPU contention scenarios. The MongoDB deployment is restricted to 10 cores. Thread priority is tweaked for 4 user sessions throughout the entire test period.

throughput and latency of the two implementations on several many-client scenarios. In these experiments, the core restriction and nice manipulation setup are the same as the timeline plots; however, the thread priorities are tweaked at the start and kept as such throughout the entire test period. In the unmodified version of WiredTiger, the overall throughput dramatically decreases as the number of session threads (one per connection) increases. In the highest contention scenario, which shows 80 YCSB connections restricted to 10 physical cores, the $P999$ latency experienced in the adaptive version is lower than the average latency of the original implementation. The scaling issue is evident: as of now, it is possible to effectively starve a WiredTiger-based data store by simply tweaking the thread priorities if the number of concurrently active session threads is not controlled under high CPU contention scenario. Fortunately, the experiments show that busy waiting with backoff is a valid fix to avoid starvation.

IV. Conclusion

In this paper, an unsafe instance of busy waiting in WiredTiger, the underlying storage engine of MongoDB, has been thoroughly described. Busy waiting is a synchronization technique commonly employed by modern OSs in kernel mode and HPC applications. It requires special care when used in user-space to avoid resource waste, or worse, starvation. However, MongoDB does not employ any safety control, causing unexpected behaviors under heavy CPU contention scenarios. For this reason, we replaced the original busy waiting implementation within WiredTiger with an adaptive one, which falls back to blocking after a short spinning period (this primitive was already used in other parts of the code, as mentioned above). This way, the scheduler is allowed to reschedule, greatly reducing the resource waste caused by busy waiting. The assumption regarding the unsafety of the original implementation is backed up with a series of experiments on a test environment carefully crafted to consistently reproduce the synchronization bug. The two implementations are then compared in high contention scenarios, manipulating the thread priorities within the Linux scheduler to consistently starve WiredTiger. The original implementation incurs in huge tail latency, making current WiredTiger-based data store that does not control concurrency a "risky" choice for large-scale applications with low-latency and predictable performance requirements (such as cloud-based, time-critical applications). The adaptive version of busy waiting overcomes the scaling issues and shows no anomalies in tail latency. An interesting future work is the use of mutable locks [21], a modern synchronization technique with a self-tuned optimized trade-off between responsiveness and CPU time usage.

References

[1] MongoDB v3.0 performance improvements. https://www.mongodb.com/blog/post/performance-testing-mongodb-30-part-1-throughput-improvements-measured-ycsb. Accessed: 2023-04-03.

[2] Breaking the WiredTiger Logjam: The Write-Ahead Log. https://www.mongodb.com/blog/post/breaking-wired-tiger-logjam-write-ahead-log, Sep 2022.

[3] Muhammad Abdullah Adnan et al. A Prediction Based Replica Selection Strategy for Reducing Tail Latency in Geo-Distributed Systems. *IEEE Transactions on Cloud Computing*, 2023.

[4] Remo Andreoli, Tommaso Cucinotta, and Daniel Bristot De Oliveira. Priority-Driven Differentiated Performance for NoSQL Database-As-a-Service. *IEEE Transactions on Cloud Computing*, pages 1–14, 2023.

[5] Remo Andreoli., Tommaso Cucinotta., and Dino Pedreschi. RT-MongoDB: A NoSQL Database with Differentiated Performance. In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 77–86. INSTICC, SciTePress, 2021.

[6] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[7] Felipe Castro-Medina, Lisbeth Rodríguez-Mazahua, María Antonieta Abud-Figueroa, Celia Romero-Torres, Luis Ángel Reyes-Hernández, and Giner Alor-Hernández. Application of data fragmentation and replication methods in the cloud: a review. In *2019 International Conference on Electronics, Communications and Computers (CONIELECOMP)*, pages 47–54, 2019.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[9] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. Strong temporal isolation among containers in OpenStack for NFV services. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021.

[10] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, and Carlo Vitucci. The Importance of Being OS-aware - In Performance Aspects of Cloud Computing Research. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 626–633. INSTICC, SciTePress, 2018.

[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[12] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[13] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods of Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '91, page 120–132, New York, NY, USA, 1991. Association for Computing Machinery.

[14] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods of Performance of Parallel Applications. *SIGMETRICS Perform. Eval. Rev.*, 19(1):120–132, apr 1991.

[15] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, dec 1983.

[16] Ru Jia, Yun Yang, John Grundy, Jacky Keung, and Li Hao. A systematic review of scheduling approaches on multi-tenancy cloud platforms. *Information and Software Technology*, 132:106478, 2021.

[17] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[18] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Transactions on Computer Systems (TOCS)*, 34(2):1–33, 2016.

[19] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions. *ACM Comput. Surv.*, 50(6), dec 2017.

[21] Romolo Marotta, Davide Tiriticco, Pierangelo Di Sanzo, Alessandro Pellegrini, Bruno Ciciani, and Francesco Quaglia. Mutable locks: Combining the best of spin and sleep locks. *Concurrency and Computation: Practice and Experience*, 32(22):e5858, 2020.

[22] Mohammad Samadi Gharajeh, Sara Royuela, Luis Miguel Pinho, Tiago Carvalho, and Eduardo Quiñones. Heuristic-based Task-to-Thread Mapping in Multi-Core Processors. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, Sep. 2022.

[23] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 513–527, 2015.

[24] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards achieving fairness in the Linux scheduler. *ACM SIGOPS Operating Systems Review*, 42(5):34–43, 2008.

[25] Chenle Yu, Sara Royuela, and Eduardo Quiñones. Taskgraph: A Low Contention OpenMP Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems*, 34(8):2325–2336, Aug 2023.