Practical Approaches to Optimizing Virtual Memory Buffer Management
Group 7

CHOU,CHE-WEI
B09502132
機械系大四

TENG, YA-WEN
R12921059
電機所碩一

WANG, WEI-LI
R12922116
資工所碩一

WU, CHI-CHIA
R12921093
電機所碩一

1 ABSTRACT

In the current data-driven era, the rapid evolution of Artificial Intelligence (AI) and big data analytics has escalated the demand for high-performance database systems. Traditional database management systems face significant performance bottlenecks when handling large-scale external storage data, especially for real-time analysis required by AI and machine learning models. This study aims to address these challenges by optimizing the vmcache database caching system's synchronization mechanism and memory allocation strategy. The original implementation of vmcache may lead to CPU resource waste, memory fragmentation, and low storage efficiency under dynamic data loads. We propose replacing the spinlock with mutex and futex mechanisms, implementing an effective idle space management strategy, and incorporating B-tree inner node merge to enhance performance. Furthermore, we have introduced a bitmap-based free space management technique to enhance the flexibility and efficiency of vmcache. Our goal is to optimize data access processes, improving both read and write performance as well as transaction throughput. By introducing these improvements, we aim to support the efficient processing of large datasets required by AI applications and other data-intensive applications, providing a more robust and scalable solution for modern database management systems.

2 INTRODUCTION

In today's data-driven era, the applications of Artificial Intelligence (AI) and big data are rapidly evolving, creating an unprecedented demand for database systems capable of quickly accessing and processing large volumes of data. Traditional database management systems face significant performance bottlenecks when dealing with large-scale external storage data, particularly for the real-time analysis required by AI and machine learning models. Additionally, with the recent rise in DRAM prices, storing all data in memory is prohibitively expensive. Hence, selectively using secondary storage devices like HDDs or SSDs and writing currently needed data to memory is a more cost-effective solution. In this context, enhancing data access speed through software-level optimization without significant hardware investment becomes crucial.

This study explores how to improve data access processes by optimizing existing database cache management strategies. Our goal is to enhance both read and write performance as well as transaction throughput. We focus on optimizing the synchronization mechanism and memory allocation strategy in vmcache. The original implementation may result in CPU resource waste, memory fragmentation, and low storage efficiency, especially under dynamic data loads. To address these issues, we introduce mutex and futex mechanisms to replace the original spinlock and implement an effective idle space management strategy. Additionally, we propose B-tree inner node merge and minor adjustments to spinlock handling to further enhance performance.

3 METHODS

In the subsequent section, we will elaborate on the methodologies employed in this study.

3.1 vmcache

Most database management systems (DBMS) cache pages from storage in a main memory buffer pool. This can be done using a hash table that translates page identifiers into pointers or through pointer swizzling which avoids this translation. vmcache, a novel buffer pool design, uses hardware-supported virtual memory to translate page identifiers to virtual memory addresses. Unlike mmap-based approaches, vmcache [2] retains control over page faulting and eviction, making it portable across modern operating systems and suitable for arbitrary graph data and variable-sized pages.

3.2 exmap

vmcache uses exmap [2], an OS extension that implements scalable page table manipulation on Linux, addressing the scalability problems of existing OS primitives with high-performance NVMe SSDs. Together, vmcache and exmap provide flexible, efficient, and scalable buffer management on multi-core CPUs and fast storage devices

3.3 Mutex and Futex

Mutex and futex [1] are synchronization mechanisms designed to handle lock contention more efficiently. Mutex locks prevent concurrent access and introduce waiting time, while futex reduces the overhead by managing locks in user space, minimizing context switches. Mutexes are often used for their simplicity and robustness, whereas futexes offer performance benefits in specific scenarios by reducing kernel involvement.

3.4 Bitmap

The atomic bitmap is a data structure where each bit indicates the residency status of a page in virtual memory. This technique is favored for its simplicity and efficiency in multi-threaded environments, showing significant improvements over the origianl method, open addressing hash tables, implemented in vmcache [2]. The atomic bitmap reduces memory overhead by representing page residency with individual bits, leading to a more compact memory footprint compared to hash tables. Additionally, the bitmap approach simplifies the management of page residency; unlike hash tables that require complex handling of collisions and tombstones, the bitmap operations (set, reset, and test) are straightforward and efficient.

4 IMPLEMENTATIONS

In the subsequent paragraph, we will detail the specific enhancements incorporated into the code.

Improve Lock Handling Mechanism with Mutex. To reduce performance overhead caused by spin locks, we propose using C++'s wait/notify mechanism for lock contention issues. This allows the system to release CPU resources while waiting for the lock to be released, improving efficiency and responsiveness. By avoiding busy-waiting, where the CPU continuously checks the lock status, we can significantly reduce CPU utilization and power consumption. Following our oral presentation, we dedicated our efforts to continuously refining and optimizing our code's performance. As a result, we have developed an enhanced mutex implementation, namely mutex-opt. This optimized version exhibits significantly improved performance in the random lookup test, primarily due to its employment of finer-grained locking mechanisms such as `unique_lock` and `lock_guard`. These techniques enable more efficient synchronization and resource management, leading to substantial performance gains. The detailed results of our performance evaluation will be presented in the subsequent chapters.

Implement Futex for Enhanced Lock Efficiency. In addition to mutex, we implemented futex (fast userspace mutex) to further minimize performance overhead. Futex reduces kernel involvement by managing locks in user space, which leads to faster lock acquisition and release. This mechanism helps in improving system performance by reducing context switches and lock contention, thus providing a more efficient lock handling approach.

Implement Bitmap Free Space Management. The design and functionality of our implementation of the ResidentPageSet using an atomic bitmap are as follow:

AtomicBitset Class. The atomic bitset consists of multiple 64-bit atomic integers, with each bit representing a page. This design allows for efficient manipulation and checking of page statuses without the need for extensive locking mechanisms. Set operation marks a page as resident by setting the corresponding bit in the atomic bitset. This operation is crucial when a page is loaded into memory. Reset operation clears the corresponding bit, indicating that the page is no longer resident. This is used when a page is evicted. While Test operation checks the status of a page by examining the corresponding bit. This operation is essential for quickly determining if a page is resident.

Batch Processing. The processPagesBatch method iterates over the bitmap, applying a specified function to resident pages. This method is optimized for performance, maintaining a static position to minimize redundant calculations.

Dynamic Resizing. Our method supports dynamic resizing of the bitmap. When the number of pages exceeds the current capacity, the bitmap can expand to accommodate new pages. This ensures that the system remains robust and performs well under varying workloads.

By leveraging the atomic bitmap for page residency management, our implementation achieves superior performance, scalability, and simplicity. This approach is particularly well-suited for environments with high concurrency and dynamic workloads, making it an effective solution for modern virtual memory systems.

B-tree Inner Node Merge. Implementing B-tree inner node merge to optimize data access and reduce fragmentation. This technique involves merging nodes in B-trees when they become underutilized, ensuring that the tree remains balanced and search operations are efficient.

Fine-tune Spinlock Handling. Minor adjustments to spinlock handling to improve performance under high-concurrency conditions. This includes optimizing the spinlock algorithm to minimize contention and using adaptive spinlocks that switch to sleep mode if contention is too high.

5 EXPERIEMENTS

5.1 Experimental Setup

Environment: Google Colab with Ubuntu 22.04

CPU: Intel(R) Xeon(R) CPU @ 2.20GHz, 2 cores, 4 threads.

Loop Device: Simulating block device storage to emulate various data sizes and access patterns. Loop devices allow us to create virtual block devices backed by regular files, providing flexibility in testing different storage scenarios without needing physical hardware changes.

Test Metrics: Random lookup operations and TPC-C benchmark.

Data Size: 5-85, with 85 being the maximum size not exceeding virtual memory limits. In this context, data size specifically refers to the number of warehouses for the TPC-C benchmark and the number of tuples for the random lookup operations.

5.2 Random Lookup Operations

Random lookup operations are critical in evaluating the performance of database caching systems. These operations simulate real-world scenarios where data access patterns are unpredictable and non-sequential, stressing the caching mechanism's efficiency in handling frequent data retrievals and updates. Evaluating random lookup performance provides insights into how well the system can manage and optimize access to data under typical usage conditions.

5.3 TPC-C Benchmark

The TPC-C benchmark [3] is a well-known standard for evaluating the performance of database systems under transaction processing workloads. It models an order-entry environment where multiple concurrent transactions of different types are executed. TPC-C is chosen for its comprehensive nature, covering various aspects of database performance, including transaction rates (tx), read

memory bandwidth (rmb), and write memory bandwidth (wmb). This benchmark helps in understanding how different optimization strategies affect the overall throughput and efficiency of the database system. Using TPC-C allows us to measure the impact of our proposed methods on realistic and complex workloads.

5.4 Testing Procedure

Random Lookup (tx): Testing different methods (Mutex,Mutex-Opt, Merge, Futex, Yield-Opt, Bitmap) against the original implementation, focusing on transaction rates under random access conditions. The goal is to measure how each method handles the unpredictable nature of random data accesses.

TPC-C Benchmark: Evaluating the performance of different methods under TPC-C benchmark scenarios, focusing on transaction rates (tx), read memory bandwidth (rmb), and write memory bandwidth (wmb). These metrics provide insights into how well each method performs under a mixed workload of reads and writes, simulating real-world database usage.

6 EVALUATION

In the evaluation section, our overall performance data is compared based on the cumulative sum of all data sizes. This means that we sum the tx (transaction rate), rmb (read memory bandwidth), and wmb (write memory bandwidth) for each method across different data sizes and then compare them to the original implementation. This approach provides a more comprehensive evaluation of the overall performance improvements for each method. Although this method is somewhat simplistic, it allows for a convenient and quick comparison of overall performance. We also provide comparative overview charts and subcharts for various methods versus the original implementation, as well as overall performance comparison charts and tables. These visualizations help to more intuitively display the performance of different methods

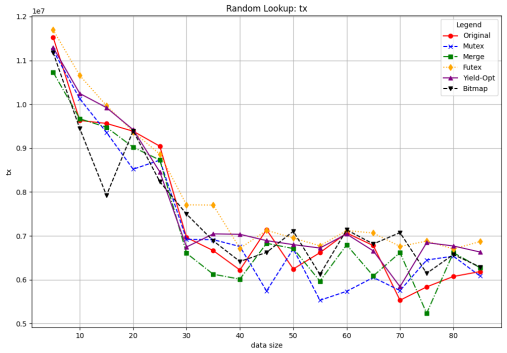

Figure 1: Random Lookup (tx) Summary Chart

6.1 Random Lookup (tx)

The summary table is shown shown in Figure 1, and the overall improvement is in Figure 2 and Table 1.

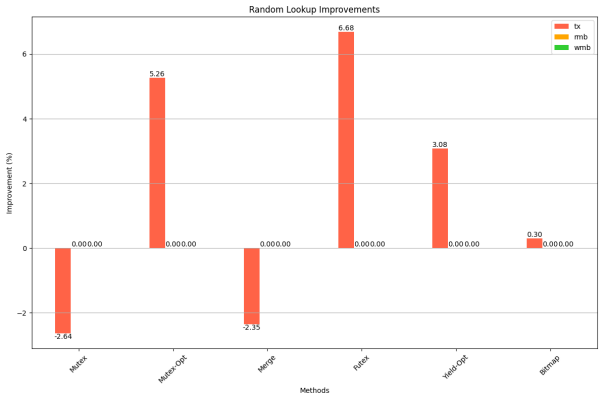

Figure 2: Random Lookup Overall Improvement Charts

Table 1: Random Lookup Overall Improvement Table

Method	tx (%)
Mutex	-2.64
Mutex-Opt	5.26
Merge	-2.35
Futex	6.68
Yield-Opt	3.08
Bitmap	0.30

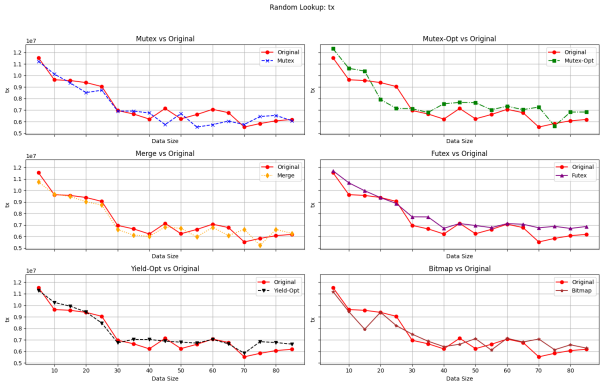

Figure 3: Random Lookup (tx) Individual Subgraphs

Mutex: In top-left subgraph of Figure 3. Performance slightly worsened with a reduction of 2.64% in transaction rates. This is attributed to the additional overhead from lock mechanisms. Mutex locks introduce waiting periods that can slow down transaction rates, especially under high concurrency.

Mutex-Opt: In the top-right subgraph of Figure 3, the performance of Mutex-OPT shows an improvement with an increase of 5.26% in transaction rates. This improvement is due to refined lock granularity, the use of atomic operations, and batch processing for page operations, which together reduce lock contention and overhead. Additionally, the use of condition variables minimizes CPU usage by avoiding busy-waiting, thereby enhancing overall system

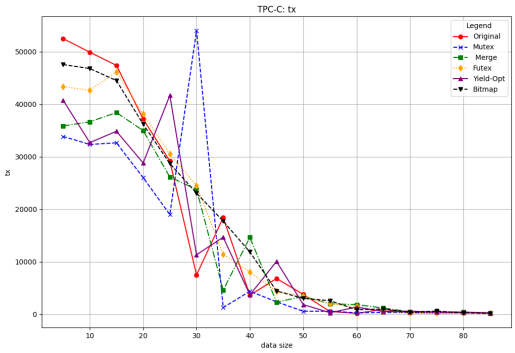

Figure 4: TPC-C (tx) Summary Chart

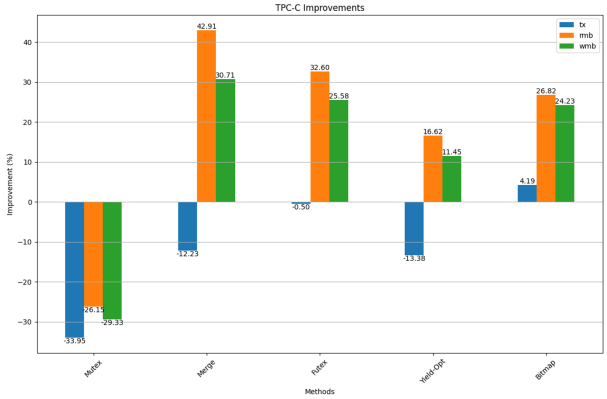

Figure 5: TPC-C Overall Improvement Charts

Table 2: TPC-C Overall Performance Comparison Table

Method	tx (%)	rmb(%)	wmb(%)
Mutex	-33.95	-26.15	-29.33
Merge	-12.23	42.91	30.71
Futex	-0.50	32.60	25.58
Yield-Opt	-13.38	16.62	11.45
Bitmap	4.19	26.82	24.23

efficiency under high concurrency conditions.

Merge: In middle-left subgraph of Figure 3.Performance remained close to the original with a slight reduction of 2.35%. This method introduces some additional overhead during merge operations, such as inner node merges, which may cause a slight decrease in performance. However, the performance decrease is not significant due to these additional merge operation overheads, thus remaining close to the original implementation.

Futex: In middle-right subgraph of Figure 3. Showed improved performance with a 6.68% increase in transaction rates. This improvement is more pronounced with data sizes larger than 40, due to reduced overhead. Futexes minimize kernel involvement, leading to faster lock acquisition and release.

Yield-Opt: In bottom-left subgraph of Figure 3. Demonstrated a slight improvement with a 3.08% increase in transaction rates. This method optimizes resource allocation through a dynamic wait strategy, adjusting the waiting strategy based on contention, thereby improving overall efficiency.

Bitmap: In bottom-right subgraph of Figure 3. Showed a marginal improvement of 0.30% in transaction rates. The efficient free space management provided by bitmaps helps in managing free space more effectively, reducing fragmentation and improving access times.

6.2 TPC-C Benchmark (tx, rmb, wmb)

The summary table is shown shown in Figure 4, Figure 6 and Figure 7. The overall improvement is in Figure 5 and Table 2. Mutex-Opt has a bug when running the TPC-C benchmark, so we did not include it in the TPC-C benchmark test.

Mutex: Performance significantly dropped across all metrics with a decrease of 33.95% in transaction rates (top-left subgraph of Figure 8), 26.15% in read memory bandwidth (top-left subgraph of Figure 9), and 29.33% in write memory bandwidth (top-left subgraph of Figure 10). The reduction in performance is primarily due to lock contention and context switching overheads.

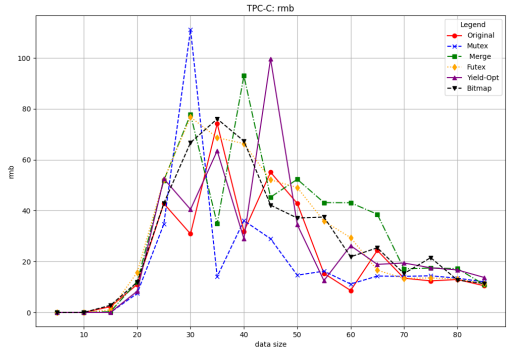

Figure 6: TPC-C (rmb) Summary Chart

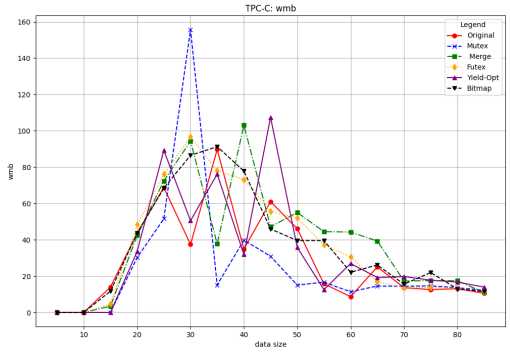

Figure 7: TPC-C (wmb) Summary Chart

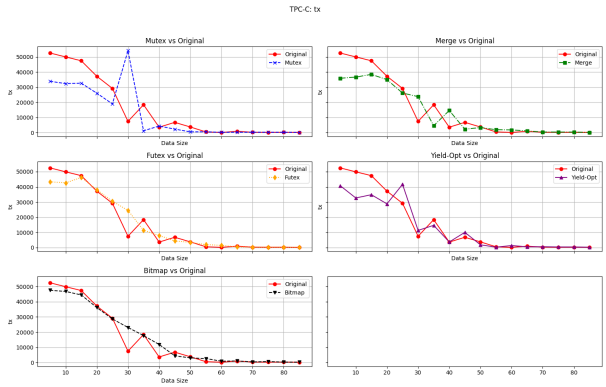

Figure 8: TPC-C (tx) Individual Subgraphs

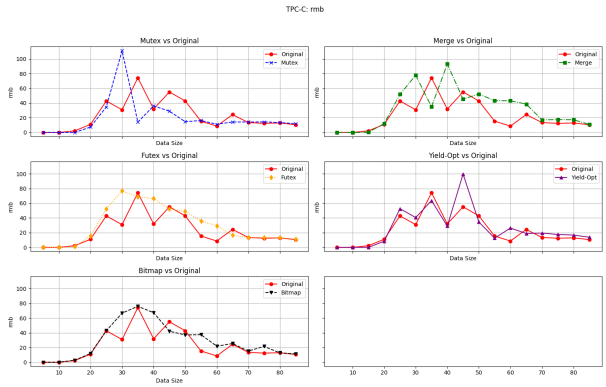

Figure 9: TPC-C (rmb) Individual Subgraphs

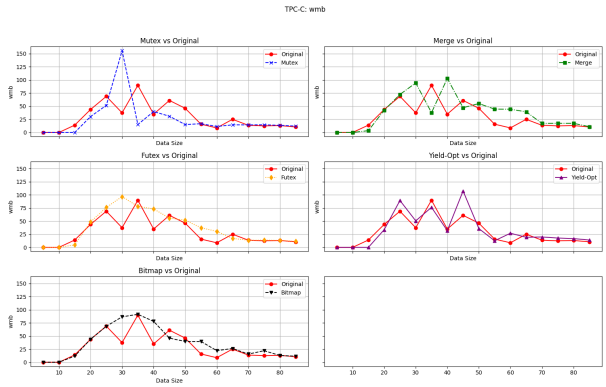

Figure 10: TPC-C (wmb) Individual Subgraphs

Merge: Performance improved in read and write memory bandwidth by 42.91% and 30.71% respectively (top-right subgraph of Figure 9 and 10), but transaction rates decreased by 12.23% (top-right subgraph of Figure 8). This method benefits medium data sizes, but the overhead increases with larger datasets, leading to a drop in transaction rates.

Futex: Showed a balanced performance improvement, with a slight decrease in transaction rates by 0.50% (middle-left subgraph of Figure 8), but notable improvements in read and write memory bandwidth by 32.60% and 25.58% (middle-left subgraph of Figure 9 and 10). The reduced context switching and lock contention contribute to these gains.

Yield-Opt: Performance was mixed, with transaction rates decreasing by 13.38% (middle-right subgraph of Figure 8), while read and write memory bandwidth increased by 16.62% and 11.45% (middle-right subgraph of Figure 9 and 10). The adaptive wait strategy helped improve memory bandwidth but was not as effective in maintaining transaction rates.

Bitmap: Performance improved across all metrics, with transaction rates increasing by 4.19%, read memory bandwidth by 26.82%, and write memory bandwidth by 24.23% (bottom-right subgraph of Figure 8, 9 and 10). The efficient free space management provided by bitmaps helps in managing free space more effectively, reducing fragmentation and improving access times.

6.3 Discussion

In the Random Lookup test (Figure 3), Futex demonstrated the best improvement in transaction rates (tx) with a 6.68% increase, primarily due to reduced overhead and minimal kernel involvement. The Mutex-Opt method also showed significant improvement (5.26%), attributed to its refined lock granularity and efficient handling of lock contention. The Yield-Opt method also showed some improvement (3.08%), attributed to its dynamic wait strategy, while Bitmap showed a slight improvement (0.30%) thanks to efficient free space management. However, both Mutex and Merge methods resulted in slightly worse performance compared to the original implementation, indicating that their additional overheads outweighed any potential benefits in this context.

For the TPC-C benchmark (Figure 8, 9, 10), the Merge method showed the most significant improvements in read memory bandwidth (rmb) and write memory bandwidth (wmb), with increases of 42.91% and 30.71%, respectively, at the cost of a slight decrease in transaction rates. The Futex method provided balanced improvements across all metrics, with slight improvements in transaction rates and substantial gains in rmb and wmb. Bitmap also showed consistent improvements across all metrics but was less pronounced than Futex. On the other hand, the Mutex method significantly reduced performance across all metrics due to lock contention and context switching overheads. Yield-Opt had mixed results, with improvements in rmb and wmb but a noticeable decline in transaction rates.

7 FUTURE WORK

Due to the environmental limitations of our current setup, we were only able to test with a maximum of 4 threads. However, the original research compared performance across various thread counts such as 4, 8, 16, 32, 64, and 128.

Future work should include testing on devices that support a higher number of threads to fully understand the scalability and performance implications of the proposed methods. In high-concurrency environments, methods such as mutex and futex may

exhibit significantly different behaviors, potentially leading to improved performance outcomes. Also,it is essential to fix the bug in Mutex-Opt to ensure its reliability. Additionally, exploring more advanced synchronization techniques and their impact on performance would be beneficial. For instance, investigating hybrid approaches that combine the strengths of multiple methods could yield more robust solutions. Conducting a detailed analysis of workload characteristics and fine-tuning each method based on specific use cases can help better understand and mitigate the trade-offs involved. Expanding the evaluation to include other performance metrics, such as latency and throughput under varying data loads, will provide a more comprehensive view of the system ' s capabilities. Incorporating real-world workloads and applications into the testing process will ensure that the optimizations are practical and effective in diverse scenarios.

By addressing these future work directions, we aim to enhance the robustness and applicability of the vmcache system, providing a solid foundation for high-performance database management in AI and big data applications.

8 CONCLUSION

This study evaluated various methods to improve the performance of the vmcache system under random access and TPC-C benchmark conditions. The Futex method demonstrated the best overall improvement, particularly in transaction rates for random lookups and memory bandwidth for TPC-C benchmarks, due to reduced overhead and minimal kernel involvement. Mutex-Opt also showed significant improvement in transaction rates at random lookup tests, attributed to its refined lock granularity and efficient handling of lock contention. Yield-Opt and Bitmap showed notable improvements, albeit to a lesser extent. In contrast, Mutex and Merge methods did not perform as well, often resulting in decreased performance due to additional overheads and lock contention. Future work should focus on scalability testing, exploring hybrid synchronization techniques, and incorporating real-world applications to further enhance the robustness and applicability of the vmcache system for high-performance database management in AI and other applications.

REFERENCES

[1] Michael Kerrisk. The linux programming interface, 2010. URL: https://man7.org/linux/man-pages/man2/futex.2.html.

[2] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proc. ACM Manag. Data*, 1(1), may 2023. doi:10.1145/3588687.

[3] TPC. Tpc-c, 1992. URL: https://www.tpc.org/tpcc/.