Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware

Demian Vöhringer
Friedrich-Alexander-Universität Erlangen-Nürnberg
demian.voehringer@fau.de

Viktor Leis
Technische Universität München
leis@in.tum.de

ABSTRACT

In this paper, we revisit the classical data management problem of page replacement. We propose Write-Aware Timestamp Tracking (WATT), a novel replacement algorithm that is optimized for modern hardware. By explicitly tracking the access history of each cached page, WATT achieves state-of-the-art replacement effectiveness. WATT is also carefully co-designed with modern multi-core CPUs and can be implemented with very low overhead. Finally, WATT allows trading of read versus write I/O operations, which is useful for prolonging flash SSD lifetime.

PVLDB Reference Format:
Demian Vöhringer and Viktor Leis. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. PVLDB, 16(11): 3323 - 3334, 2023.
doi:10.14778/3611479.3611529

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/leanstore/leanstore/tree/WATT.

1 INTRODUCTION

Replacement Algorithms. Every system that caches data eventually must decide what to evict from cache. Page replacement is therefore a classical data management problem, and several algorithms for this task exist. Most database systems still rely on classical algorithms such as LRU and its approximations such as CLOCK. Only few systems use advanced algorithms, including LRU-K [15] and ARC [13], which are also well known and have better accuracy.

Hardware Changes. We argue that it is time to revisit the page replacement problem due to major changes in the hardware landscape. Solid-State Drives (SSDs) have largely replaced magnetic disks as a storage medium. In comparison with disks, high-end SSDs have four orders of magnitude higher random read throughput. Flash SSDs also have read/write asymmetry, meaning that writes are slower than reads, and that device lifetime is determined by write volume. Additionally, modern servers changed from having only one core to dozens or even hundreds of cores. From these changes, we can derive four goals that any modern replacement algorithm should achieve.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611529

Goal 1: Replacement Effectiveness. The primary goal of any replacement algorithm is to minimize I/O. For many workloads, the number of I/O operations directly determines overall system throughput. Given that DRAM capacities have stagnated in the past decade [4] and persistent memory has been canceled by Intel, I/O-bound workloads are expected to become more common and consequentially replacement effectiveness to (again) become a crucial optimization goal, directly affecting DBMS performance.

Goal 2: Write Awareness. Most replacement algorithms (including LRU, LRU-K, and ARC) explicitly or implicitly try to maximize the cache hit rate. Writes of dirty pages are assumed to happen in the background and have no cost. We argue that optimizing for the hit rate alone, which does not take writes into account, is too simplistic. For example, consider two algorithms $\mathcal{A}$ and $\mathcal{B}$ with hit rates of 90% and 89%, respectively. Algorithm $\mathcal{A}$ appears superior – but it could easily be the case that $\mathcal{A}$ causes twice as many writes as $\mathcal{B}$. Therefore, $\mathcal{A}$ could in fact be substantially slower than $\mathcal{B}$ both in terms of throughput and (tail) latency – while also halving SSD lifetime. Any algorithm designed for flash should therefore take writes into account.

Goal 3: CPU Efficiency. A single modern PCIe 4.0 SSD can achieve more than 1M IOPS [20] and commodity servers can directly connect to multiple devices. Consequently, a database system trying to exploit such hardware must be capable of replacing millions of pages per second. This means that finding replacement candidates must be highly efficient to avoid becoming CPU bound. A second important aspect of CPU efficiency is that accessing cached pages needs to have low overhead as well to avoid slowing down workloads with high page hit rates.

Goal 4: Multi-Core Scalability. Our final goal is multi-core scalability. Both page access tracking and replacement must scale well, as they will be performed by multiple threads concurrently. This, for example, prohibits the use of global lists and queues which would immediately become points of contention on modern highly-concurrent machines. Therefore, only decentralized algorithms and data structures are viable on modern hardware.

Write-Aware Timestamp Tracking. In this work, we propose *Write Aware Time Tracking (WATT)*, a new page replacement algorithm that achieves all four goals simultaneously. Our extensive simulation results show that WATT achieves state-of-the-art replacement effectiveness. This is achieved by tracking individual page accesses, which contain substantially more information than other tracking mechanisms, together with a page value scoring function that considers both frequency and recency. WATT tracks reads and writes separately and has a write-weight parameter, which allows explicit write penalization.

Implementation. WATT is engineered to be efficient on modern CPUs – for example, by exploiting SIMD instructions and prefetching. The implementation also avoids centralized data structures and exploits epoch-based tracking, which make WATT scalable on multi-core CPUs. We integrate WATT into the open-source high-performance storage engine LeanStore and demonstrate that it substantially improves end-to-end performance for out-of-memory workloads without negatively impacting in-memory performance. WATT can easily be integrated into existing caching-based systems, and we believe that many systems would benefit from adopting it. We additionally provide an open-source page replacement simulation framework that includes workload traces and ten replacement algorithms, which we believe will prove useful to others.

Outline. The rest of the paper is organized as follows. Section 2 surveys the replacement algorithms employed by widely-used database systems and discusses prior work of page replacement. Section 3 presents the main algorithmic ideas behind WATT. We then compare WATT against three classical and six state-of-art competitors through simulation in Section 4. Section 5 describes how to implement WATT efficiently, and Section 6 evaluates this implementation within LeanStore. Finally, we summarize our paper and discuss future work in Section 7.

2 RELATED WORK

A Neglected Problem. Although page replacement is a classical problem in the data management field, in the past two decades research on the topic has become sparse. This may have been due to the shifting focus from disk-based to main-memory database systems. However, given that DRAM prices have stagnated during the past two decades [4], and flash has become affordable and efficient, we argue that it is time to revisit the problem. In the following, we first present a short survey of the replacement algorithms used by existing systems, before discussing more recent proposals.

2.1 What Database Systems Use

Commercial systems. Most existing systems rely on *Least Recently Used* (LRU), or its variants and approximations. By default, *DB2* [5] uses LRU but also offers users the option to switch to *First In First Out* (FIFO), which has less internal latch contention. *Oracle* [16] uses standard LRU as well but partitions the LRU lists to reduce latch contention. Additionally, a separate strategy is used to avoid thrashing for large table scans. *SQL Server* [3] is one of the few widely-used systems that uses a slightly more advanced algorithm, namely LRU2, which is an instantiation of LRU-K [15]. LRU2 uses the next-to-last access to find replacement candidates.

Open-source systems. *InnoDB*, the storage engine of *MariaDB* [11] and *MySQL* [14], modifies LRU by inserting new pages in the middle (instead of the front) of the LRU list. This ensures that large table scans do not thrash the entire buffer pool (but only a part of it). *PostgreSQL* [12] uses Clock-Sweep, a variation of CLOCK. *WiredTiger* [23], which is the default storage engine of *MongoDB*, uses an approximation of LRU. The implementation scans the whole RAM for pages older than a certain threshold and replaces them.

LeanStore. This paper is part of the *LeanStore* [9] project. LeanStore is a high-performance storage engine for flash and is based on pointer swizzling for fast access to cached pages. Its original replacement algorithm, called *LeanEvict*, was co-designed with the pointer swizzling scheme to have no overhead for accessing hot pages. This is achieved by a combination of randomly selecting replacement candidates and then putting them into a FIFO list ("cooling stage"). Pages are then replaced if they arrive at the end of the list without having been accessed in the meantime. The length of the FIFO list can be configured (e.g., 10% of the buffer pool) and determines how much access history is considered. In Section 6, we show that for out-of-memory workloads, WATT enables replacement effectiveness without sacrificing the efficiency and scalability introduced with LeanEvict.

2.2 Advanced Replacement Algorithms

Most surveyed systems rely on variants or approximations of LRU. Modifications such as partitioning the LRU list may improve multi-core scalability and CPU efficiency, but come at the cost of even lower replacement effectiveness, i.e., more I/O. In the following, we first discuss more advanced algorithms that try to mitigate these issues, and then discuss optimizations for modern hardware.

Replacement Effectiveness. An inherent problem of LRU is that a page with low access frequency ("cold") that happens to be accessed will reside in the cache for a long time, i.e., until it arrives at the end of the list. It would be beneficial to replace such cold pages earlier to make space for "warmer" pages with higher access frequency. One algorithm that tries to achieve this is LRU-K [15], which replaces pages based on their K^{th}-to-last accesses timestamp. In practice, K is set to 2, which effectively means that the most recent access is ignored. Another approach is having two lists instead of one such that pages with very different frequencies can be separated. With the 2Q [7] algorithm, for example, pages are first entered into a FIFO list, and only if they are accessed a second time, they are moved to an LRU list. One downside of 2Q is it has several difficult-to-set parameters, in particular the lengths of the two lists. The *Adaptive Replacement Cache* (ARC) [13] algorithm also relies on two lists, but can adaptively determine their sizes based on the workload. All these algorithms try to improve replacement effectiveness, but they all run into severe scalability problems on modern CPUs due to contention hot-spots at the head of the LRU list.

Multi-Core scalability. Scalability can be achieved by making replacement a distributed and sampling-based process, as has been proposed by the *Hyperbolic Caching* [2] approach. The algorithm relies on a page value function, determining which of the sampled pages to replace. WATT relies on similar ideas for scalability, but differs in its tracking and page value function approach.

Write awareness. With *Clean-First LRU* (CFLRU) [18] and *LRU with Write Sequence Reordering* (LRU_WSR) [8], two heuristics that try adding write awareness to LRU have been proposed. LRU_WSR gives modified (dirty) pages a second chance through the LRU list before replacing them. Rather than always replacing from the end of the LRU list, CFLRU replaces pages from an interval with given size at the end, where it prefers replacing clean to dirty pages. Both are simple heuristics that do not allow trading off reads vs. writes in a principled way, e.g., keeping dirty pages in RAM might even be prioritized *too much*.

Table 1: Comparison of replacement algorithms

	Replacement Effectiveness	Write Awareness	CPU Efficiency	Multi-Core Scalability
random	−−	no	++	++
CLOCK	−	no	+	+
LeanEvict [9]	−	no	+	+
LRU2 [15]	+	no	++	+
LRU	−	no	−	−
CFLRU [18]	−	yes	−	−
LRU_WSR [8]	−	yes	−	−
Hyperbolic [2]	~	no	++	++
ARC [13]	+	no	−	−
WATT (our)	++	yes	++	++

2.3 Discussion

Table 1 surveys ten classical as well as modern replacement algorithms in terms of the four goals introduced in Section 1. Only WATT, the technique proposed in this work, satisfies all four goals. Given that LRU has been known for decades and more effective algorithms exist [13, 15], it is surprising that most systems use a variant of LRU. LRU scores badly on all four metrics and, as we show in Section 4, switching to a more effective algorithm can reduce the volume of I/O by 10% or more on many workloads. Interestingly, the PostgreSQL community discussed adopting ARC, but these efforts were eventually abandoned due to patent concerns [19].

3 WRITE-AWARE TIMESTAMP TRACKING

Access History. The task of an online replacement algorithm is to replace pages based on their past access history, and algorithms differ in how much and what they track. In general, it is intuitively clear that storing more information should lead to more accurate page replacement. For example, whereas the CLOCK algorithm maintains only a single bit per cached page indicating whether the page has recently been accessed, LRU maintains a recency rank by ordering cached pages in a list. However, even the more effective LRU (1) does not directly track access frequency, (2) stores no history for pages, and (3) does not distinguish between reads and writes. WATT maintains more information and is therefore capable of substantially reducing I/O.

3.1 Basic Algorithm

Timestamp Tracking. WATT tracks the access history of each page by maintaining timestamps in an access log, adding new timestamps to the front at position i=1. In the following example, a page has initially been accessed at timestamps 0, 8, and 15, and is then accessed a fourth time at timestamp 42:

i	1	2	3
t_i	15	8	0

$\Rightarrow$

i	1	2	3	4
t_i	42	15	8	0

The Value of a Page. A full access history contains strictly more information than just a recency ordering (LRU) or an access frequency count (LFU). However, it is not obvious how to use this information to determine which pages to replace. To do this, WATT uses a value function that takes the page access log as its input. A page with a lower value is more likely to be replaced than one with a higher value.

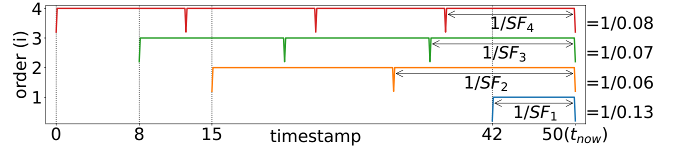
Subfrequencies. To compute the page value at timestamp t_{now}, we first compute the subfrequency (SF) of each access in the log:

$$SF_i(t_{now}) := \frac{i}{t_{now} - t_i} \tag{1}$$

As the name of the function implies, it computes the current access subfrequency for each order i. For the first order subfrequency the formula considers only the most recent access, for the second order subfrequency the two most recent accesses, and so on. For each order, the formula divides the number of accesses occurred (i) by the time difference between the current time and the oldest timestamp in that order ($t_{now} - t_i$). In our example, we obtain the following results for $t_{now} = 50$:

i	1	2	3	4
t_i	42	15	8	0
SF_i	$1/8 \approx 0.13$	$2/35 \approx 0.06$	$3/42 \approx 0.07$	$4/50 \approx 0.08$

The example can also be illustrated geometrically:



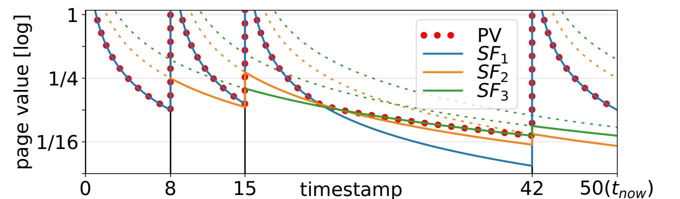
We see that within each order, we split the overall time interval into equi-width ranges. Each range equals the inverse of the subfrequency.

Computing the Page Value. After computing the subfrequencies, we have to combine them to obtain the overall page value (PV). We do this by computing the maximum of all subfrequencies:

$$PV^*(t_{now}) := \max_i SF_i(t_{now}) \tag{2}$$

It would also be plausible to use the mean or median, but the maximum is cheaper to compute and works just as well.

Recency vs. Frequency. The two classical page replacement strategies LRU and LFU try to model two completely different scenarios. LFU implicitly assumes statistically-independent and constant page access frequencies – therefore only long-run frequency matters. LRU, in contrast, assumes bursts of potentially changing working sets that fit into the cache – here only recency matters. In our framework, LRU (LFU) would be equivalent to using SF_1 ($SF_{\max}$) as the value function. By combining the subfrequencies, we obtain a compromise between both extremes. During a burst, the recent frequencies will be high and give the working set a high probability of staying in cache. After the burst, the value of a page with just a few recent accesses will shrink quickly until it is dominated by a higher-order subfrequency, which corresponds to the long-run frequency. This can also be illustrated graphically by showing how the page value and the subfrequencies evolve in our running example:



Accesses are indicated as solid vertical lines, and subfrequencies of accesses are shown with dashed lines. They turn into a solid line once enough accesses happen and the subfrequency becomes valid. The subfrequency responsible for the pages value is marked with red dots. For example, we see a green dashed line for subfrequencies of third order between $t = 0$ and $t = 15$, resulting from the access at $t = 0$, turning solid and valid at the access at $t = 15$. From $t \approx 20$ to $t = 42$ this third order subfrequency is responsible for the pages value and marked with red dots. We can also see that whenever a new access arrives, *PV* briefly spikes due to a high SF_1 value, but also quickly decreases over time – converging towards the long-run frequency.

3.2 Refinements

Finding Replacement Candidates. Due to its dependence on t_{now}, the value of a particular page changes over time even when the page is not accessed. It is therefore not feasible to maintain all pages in a queue ordered by their value. To find replacement candidates, we instead rely on *random sampling* and *lazy evaluation*, as proposed by Blankstein et al. [2]. We pick $n \in \mathbb{N}$ random pages, evaluate their values, and select the lowest value page for removal from the buffer. This method on average selects the lowest (n+1)-quantile of all pages for replacement. For example, if we sample nine pages, in expectation we replace pages at the tenth percentile.
Write Awareness. To make our algorithm write-aware, we add a second list to track write timestamps separately from access timestamps. A write access is tracked twice, once as an access, once as a write. We calculate the value for accesses and writes separately before combining them to a single page value:

$$PV(t_{now}) := PV^*_{access}(t_{now}) + write_weight \cdot PV^*_{write}(t_{now}) \quad (3)$$

The *write_weight* parameter determines how expensive a write access is in comparison with a read access. Increasing the parameter will cause WATT to prioritize the replacement of clean pages – thereby reducing write-backs by keeping more dirty pages in the cache.
Bounding History Size With Epochs. It is of course not practical to store the full access history of each page. We therefore limit the history size of each page to a constant number of accesses (e.g., eight accesses plus four writes). The downside of a limited history is that a short burst of accesses quickly exhausts the history limit and all older access history gets lost. To prevent this problem, we group accesses into epochs by incrementing the current time (t_{now}) infrequently. Storing 12 timestamps and 2 positions with 4 Byte timestamps and 1 Byte positions adds 50 Bytes to the pages header, fitting into a single cache line. A comparison of memory overhead per cached page follows:

Memory Overhead	random	CLOCK	LeanEvict	LRU2	LRU	CFLRU	LRU_WSR	Hyperbolic	ARC	WATT
Bytes[1]	0	$\frac{1}{8}$	8	8	24	24	$24+\frac{1}{8}$	8	48	50
Overhead[2] [≈%]	0	0	0.2	0.2	0.6	0.6	0.6	0.2	1.2	1.2

[1]Using: timestamp = 4 Byte, pointer or PageID = 8 Byte, doubly-linked list = 2 pointers + 1 PageID = 24 Byte
[2]4KB pagesize

Table 2: Workload statistics. *Hot pages* with >2% of accesses; *Top 10*: fraction of accesses to 10 hottest pages

	TPC-C	TPC-E	ZipfRO	dynZipfRO	dynZipfRW
Writes	15.6%	5.7%	0%	0%	16.7%
Accesses	1 M	1.5 M	1 M	2 M	1.2 M
Pages	16,128	65,656	10,000	20,000	20,000
Hot pages	8	24	7	1	0
Top 10	36.0%	30.4%	20.5%	4.5%	2.5%

How Many Epochs? An approach we found to work well is to couple the growth of t_{now} to the number of pages replaced. Suppose, for example, we have a cache of 1 million pages. To be able to distinguish which of these pages to replace next, we set a goal of having ten epochs for every 1 million replacements. In other words, we increment t_{now} once for every $cacheSize/10 = 0.1$ million replacements. Epochs not only allow bounding the history size, they also prevent contention hot-spots on multi-core CPUs, which can otherwise occur on frequently-accessed pages as we discuss in Section 5.
Dampening First Order Subfrequencies. Many replacement strategies, including LRU-K [15] and ARC [13], are based on the idea of not giving the most recent access too much weight. Intuitively, this helps because the most recent access often proves to be transitory rather than exemplary. We can incorporate this idea through dampening the first order subfrequency. We therefore divide SF_1 by a constant that we will determine experimentally.

4 SIMULATION-BASED EVALUATION

Methodology. The evaluation of WATT is split into two parts. In this section, we compare its effectiveness against existing strategies through simulation and describe how we tune parameters. The efficiency and scalability of a high-performance implementation of WATT in LeanStore will later be evaluated in Section 6. We implemented the ten algorithms shown in Table 1 in our open-source simulation-based framework and tested them on five workload traces. The simulator takes a cache size, and a page access trace as input and computes the number of read and write I/O operations. This allows us to quantitatively compare the effectiveness of WATT with its competitors.

4.1 Workload Traces

Workload Overview. For our evaluation we use the five workload traces listed in Table 2. TPC-C and TPC-E are two well-known OLTP benchmarks containing a non-negligible fraction of writes [22]. The workloads were created by instrumenting Shore-MT [6] with Shore-Kits. The other three workloads are based on the Zipf distribution [21]. *ZipfRO* is generated by a standard Zipf distribution and is read only. By slightly shuffling the Zipf order every few accesses, and by adding a scan, we created the *dynZipfRO* trace, whose access pattern can be visualized as follows:

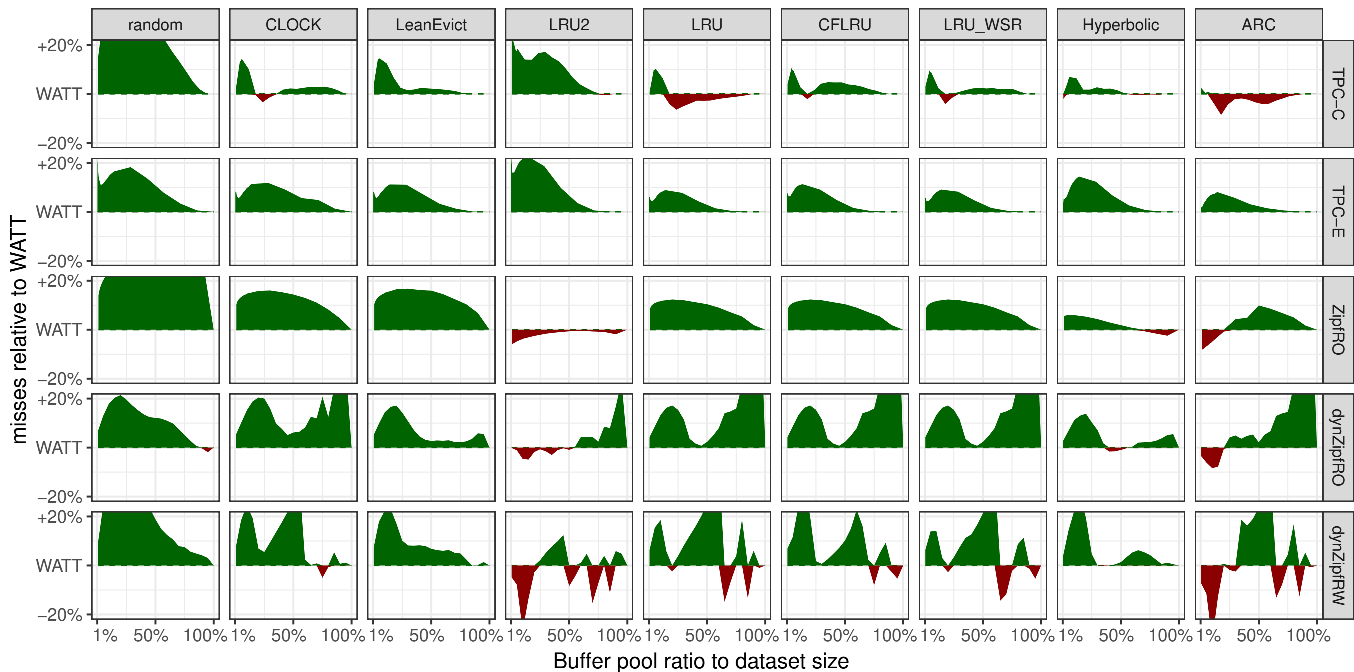
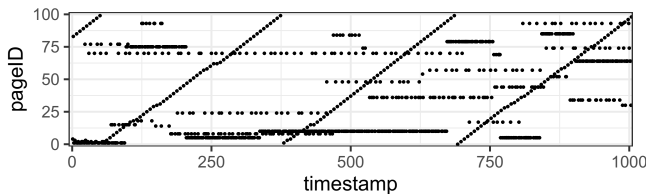


Figure 1: Replacement algorithm comparison



This workload is challenging not only because a sequential scan may cause hot pages to be replaced unnecessarily but also because the set of hot pages changes over time. Finally, we weave Zipf-distributed writes into the mix, obtaining *dynZipfRW*.

Workload Statistics. Statistics of the workloads are shown in Table 2. The fraction of writes for TPC-E, TPC-C, and dynZipfRW ranges from 5.7% to 16.7%. To give an indication of the amount of skew in the workloads, the table shows the *hot pages* and *top 10* statistics. The former is the number of pages with more than 2% of all accesses, and the latter is the fraction of all accesses on the 10 hottest pages in the trace.

Strategy Parameters. To show a generalized and objective view of the performance of each strategy, we tried to get a set of different workloads, and run each strategy on multiple RAM sizes. For each strategy that has configuration parameters, we performed a grid search for an overall good performing parameter set. For *CFLRU* we picked a *Clean-First-Window* of 30%, *LeanEvict* was run with a *Cooling-Stage-Size* of 30%, and for *Hyperbolic Caching* we used a *Sample-Size* of 20.

4.2 Replacement Effectiveness

Page Hit Rate. In Figure 1, we show the page misses normalized by WATTs' page misses for every RAM size and every workload. Situations where WATT outperforms its competitor are colored in green; red areas indicate cases where WATT is less performant than the competitor. Overall, across all workloads and data sizes, WATT has the best effectiveness on average. The results also provide insights into the relative strengths and weaknesses of the algorithms. Clearly, the random strategy has the worst effectiveness. CLOCK and LeanEvict are slightly worse than LRU, which is not surprising as they are effectively approximations of LRU. The same is the case for the write-aware LRU variants CFLRU and LRU_WSR. Interestingly, LRU2 achieves better performance than LRU on Zipf-based workloads, but on TPC-C and TPC-E it performs worse than LRU. Hyperbolic and ARC generally perform better than LRU, but on average still worse than WATT.

Page Hit Rate Vs. Writes. As discussed in Section 1, one of our main goals is extending SSD lifetime by reducing writes. In Figure 2, we show how the different algorithms balance page misses and writes on TPC-C and TPC-E with a fixed RAM size. Algorithms without tuning parameters are presented as points while WATT and CFLRU are presented as lines. In TPC-E, all configurations of CFLRU give the same result as all dirty pages fit in the *clean first* area. Out of all write-oblivious strategies, WATT with *write_weight* set to 0 performs best by having the least number of misses and in TPC-C the least amount of writes, too. In TPC-E, Hyperbolic generates slightly fewer writes but significantly more misses. By increasing the write_weight WATT is able to shift its focus from page misses to writes and reduce writes by over 10% in TPC-C (20% on TPC-E) while increasing page misses by less than 15% (7%). Only CFLRU is able to achieve similar levels of write reduction but has much more misses. Overall, we see that WATT not only offers a lower page miss rate than its competitors, it also enables trading off these misses to a substantially reduction in writes.

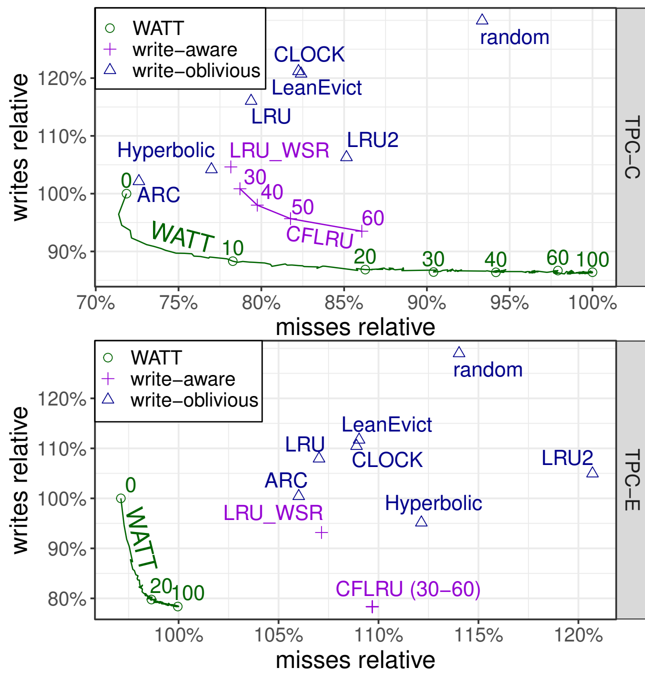


Figure 2: Read/write tradeoff

4.3 WATT Parameters

Parameters. In Section 3, we mentioned that WATT has several parameters, but did not specify how we configured them. In the following, we experimentally show each parameters impact on effectiveness and derive appropriate default parameters. We start with an untuned configuration, and then incrementally tune each parameter in a separate experiment:

Parameter	Untuned	WATT
random sampling size	8	8
access log length	1	8
epochs per full RAM replacement	max	4
dampening factor of first access	1.0	0.1
aggregation function	max	max
write access log length	0	4
write weight	0	4

Sampling Size. Our replacement algorithm samples a certain number of pages and uses the lowest page value among them as the eviction threshold. The impact of this sample size on the miss rate is shown in Figure 3. Interestingly, a larger sample does not always reduce misses. Given that a larger sample size also leads to slower replacement, we chose eight samples as the default. This means that, in expectation, pages with a value at the eleventh percentile are evicted. As in all following figures, the default setting is indicated by a dotted green line.

Access Log Size. In order to keep used space in bounds, we need to limit the number of timestamps we keep for each page. As Figure 4 shows, it is a trade off between memory and CPU consumption and the quality of a page value. As default we chose eight.

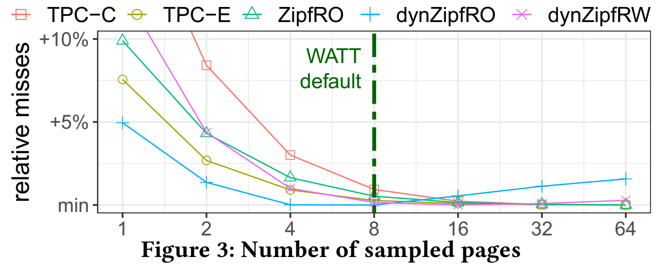


Figure 3: Number of sampled pages

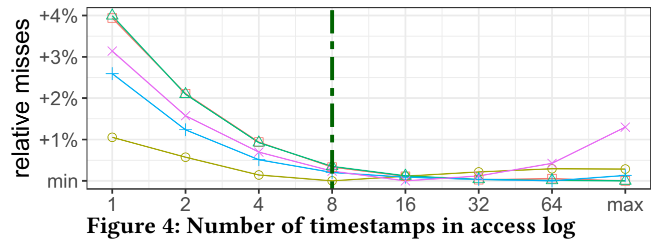


Figure 4: Number of timestamps in access log

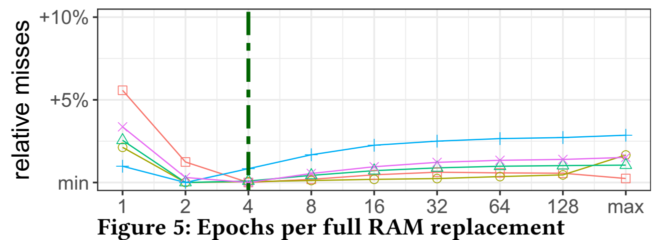


Figure 5: Epochs per full RAM replacement

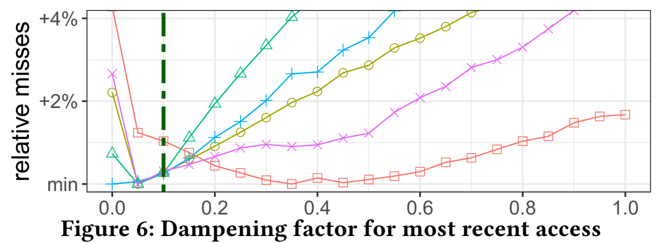


Figure 6: Dampening factor for most recent access

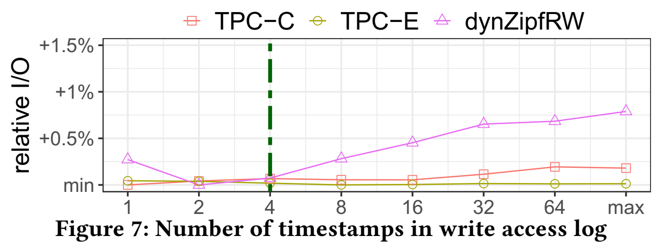


Figure 7: Number of timestamps in write access log

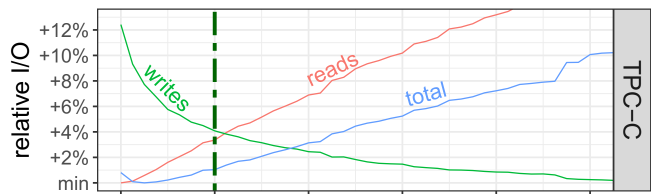


Figure 8: Impact of write_weight (cf. Equation 3) on I/O

Replacements per Epochs. An other parameter presented is the update frequency of the global time. One extreme would be an increment after as many evictions as we have pages in the buffer pool (1 epoch per full cache replacement). The other extreme would be an increment after every eviction (max). Figure 5 shows the impact of different settings. For our default configuration we chose 4 epochs per full cache replacement.

Dampening of Most Recent Access. Figure 6 shows that dampening the most recent access (by multiplying it with a value less than 1) substantially improves replacement effectiveness. Considering all workloads, a value of 0.1 is a good choice.

Aggregation Function. WATT aggregates the subfrequencies by computing their maximum. Since other functions appear plausible as well, we compare their impact in misses relative to taking the *maximum* below:

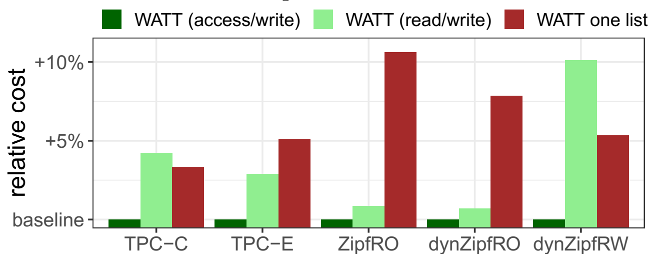
Function	TPC-C	TPC-E	ZipfRO	dynZipfRO	dynZipfRW
minimum	+4.5%	+5.8%	+11.1%	+7.7%	+9.9%
average	-1.0%	+0.5%	+1.1%	+0.7%	+1.1%
median	+0.5%	+0.4%	+1.0%	+0.6%	+0.4%

These results show that the minimum would be a bad choice, and that median and average are slightly worse than our default maximum function.

Write Access Log Size. By adding a write access log, WATT becomes write-aware. Figure 7 shows that tracking 4 writes results in the best performance over all workloads. For this experiment, we used a write weight of 1.

Write Weight. Because write costs depend on the used hardware and workload, the page value function of WATT includes a write_weight parameter. Its effect on *read*, *write*, and *total* I/O can be seen in Figure 8. We chose a default write_weight of 4.

Write-Awareness Variants. We simulated two other logging variants of WATT. One has a strict separation of reads and writes in different lists. The second variant uses one list and a boolean per timestamp to distinguish reads from writes. The approach with one access and one write list outperforms the alternatives:



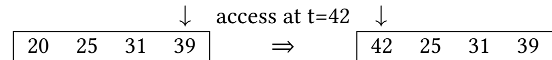
Parameter Correlations. So far, we optimized each parameter individually. To find out whether there are meaningful correlations between the them, we also looked at the pairwise interactions between all parameters. For example, we tried the 7 settings for *random sampling size* and the 8 settings for *access log length*, resulting in 56 simulation runs per workload. Across all parameters, the observed differences were small and we did not find any meaningful correlations.

5 HIGH-PERFORMANCE IMPLEMENTATION

LeanStore Integration. We integrated WATT into LeanStore, an open source, high-performance OLTP storage engine optimized for many-core CPUs and NVMe SSDs. LeanStore was designed to achieve in-memory performance comparable to main-memory database systems [9]. As mentioned in Section 2, its replacement algorithm basically incurs no overhead for accessing cached pages. Unless implemented very carefully, a more sophisticated algorithm like WATT may run into the risk of slowing down in-memory workloads though additional tracking and page value calculation overhead. In this section, we present a high-performance implementation that is carefully co-designed with respect to the properties of modern multi-core CPUs.

5.1 Efficient and Scalable Tracking

Cyclic Timestamp Insertion. Listing 1 shows the code for page tracking. A naive implementation of timestamp tracking would store the latest timestamp at the front of the timestamp log and, on access, would have to shift all elements to the right (cf. Section 3.1). To avoid shifting, we insert new timestamps in a cyclic manner:



For this we introduce an *accessHead* (↓) to indicate the current head's position.

Fast Access. Retrieving the timestamps and the accessHead must be fast, because it is done on every page access. We therefore store all tracking information, which fits into one cache line, next to other frequently-accessed page metadata such as the (optimistic) latch [10] of the page. Together, the metadata and tracking information fit into two neighboring cache lines. In practice, the adjacent cache line hardware prefetching feature of modern CPUs thus hides the additional cache line accesses caused by timestamp tracking.

Epochs Prevent Cache Line Ping Pong. As mentioned in Section 3.2, we group accesses into epochs by incrementing the current global time (`globalTrackerTime`) infrequently. This design choice not only has conceptual benefits (it prevents an access burst from invalidating the entire history of a page), it is also crucial for scalable concurrent page tracking. In the implementation of tracking, we only insert a new timestamp if the current time changed since the most recent page access timestamp (line 20). Therefore, the tracking mechanism avoids writing to hot pages too frequently. For example, consider a concurrent in-memory B-tree workload where every operation accesses the root. Without epochs, the root page of the tree would become a contention hot-spot because every access would modify the tracking information – leading to *cache line ping pong* and degrading scalability. With epochs and slower changing global time, for most accesses the tracking information of the root remains unchanged – avoiding scalability issues at the root.

Lock-Free Insertion With Release Memory Order. In cases where the current time is greater than the most recent timestamp of a page, we must insert that timestamp. We avoid explicit locking and perform the insertion in three steps. After computing the new head (line 22), we store both the timestamp (line 23) and the new head (line 24). The timestamps and the heads have to be atomic variables because they may be accessed concurrently. However, instead of the default sequentially-consistent memory order, we use the *release* memory order. On x86, a *release* store will simply be translated into a single MOV instruction – without any fences that cause write buffer flushes.

```
atomic<uint32_t> globalTrackerTime // time (epoch)

class PageTracker // sizeof(PageTracker) < cache_line
   // 8 and 4 from Figure 4 and Figure 7
   atomic<uint32_t> accessLog[8], writeLog[4]
   atomic<uint8_t> accessHead, writeHead
   // Track read and write accesses
   void track()
   void trackWrite() // Similar to track()
   // Calculate access frequency using SIMD
   float PVaccess() // See Listing 3
   float PVwrite() // Similar to PVaccess
   // Calculate total page value
   float getValue()
      return PVaccess() + (4 * PVwrite())
      // 4 from Figure 8

void PageTracker::track()
   // Compare last tracked and current epoch
   uint8_t oldPos = accessHead.load()
   uint32_t now = globalTrackerTime.load()
   if (now != accessLog[oldPos])
      // Store current epoch if they differ
      uint8_t pos = (oldPos+1) % 8
      accessLog[pos].store(now, memory_order_release)
      accessHead.store(pos, memory_order_release)
```

Listing 1: Tracking a single read access

Low Overhead for High-Frequency Pages. The page tracking code (PageTracker::track in Listing 1) is optimized to be efficient on modern processors. With GCC 12.2, checking whether the most recent page timestamp differs from the current global time translates to 6 x86 instructions. For frequently-accessed pages, this is all that has to be done. If the check fails, 6 simple instructions implement storing the timestamp. The code uses no fences or expensive atomic instructions. We exploit the fact that x86 offers strong memory ordering by default (e.g., no reordering of writes), and that the remaining potential races are harmless in our use case.

5.2 Replacement

Requirements. A single modern SSD can execute one million random read I/O operations per second or more. Out-of-memory workloads will therefore have to replace pages at very high rates. The implementation of the replacement algorithm therefore must be fast (i.e., few CPU instructions) and scalable (i.e., multiple threads should be able to perform eviction in parallel).

Basic Algorithm. As discussed in Section 3.2, replacement uses sampling and is completely distributed, i.e., it can be executed in separate threads. Listing 2 shows the three main steps of the algorithm. In step 1, a small number of pages is sampled. The minimum page value among this small sample becomes the threshold for eviction. Step 2 samples a larger number of pages as eviction candidates and compares their page value with the threshold. To avoid having to physically latch each page, our implementation in LeanStore relies on optimistic, version-based locks and validation [1] (not shown in the code). Dirty pages are handled separately by moving them to an asynchronous write buffer[3] and evicting them after the write finished (not shown in the code). Clean pages can be evicted directly. Step 3 increments the epoch whenever a sufficiently large number of pages have been evicted. This ensures that the global time advances in lockstep with the buffer pool evictions without having too many epochs.

```
// Step 1: Determine value threshold through sampling
vector<Page*> sample = samplePages(8) // 8 from Figure 3
double threshold = +Infinity
for(Page* page : sample)
   threshold = min(threshold, page.tracker.getValue())

// Step 2: Evict pages below threshold
for(Page* page : samplePages(64))
   if(!page.tryLock())
      continue // skip page
   if(page.tracker.getValue() > threshold)
      page.unlock()
      continue // skip page
   if(page.isDirty())
      // will be evicted once write is done
      page.writeAsych()
      continue
   // Evict page
   page.evict()
   page.unlock()
   pagesEvicted++

// Step 3: Occasionally move to next epoch
uint32_t epochSize = (cacheSize/64) // 64 from Figure 5
if(pagesEvicted >= epochSize)
   pagesEvicted = 0 // reset counter
   globalTrackerTime++
```

Listing 2: Simplified page replacement algorithm

Prefetching and SIMD. Our algorithm samples many more pages than it evicts, and every sampled page results in a cache miss and page value calculation. We optimize the former through explicit prefetching instructions, and the latter through SIMD. Because the algorithm shown in Listing 2 always works on batches of pages, prefetching can easily be implemented, e.g., within the samplePages function. This largely hides the cache miss latency that would otherwise dominate overall replacement runtime. Once latency is hidden, page value calculation becomes a bottleneck. As a reminder, the page value (with dampening) is computed as follows:

$$\max\left(\max_{i\in\{2,..,8\}}\left(\frac{i}{t_{now}-t_i}\right), \frac{0.1}{t_{now}-t_1}\right)$$

Listing 3 shows a SIMD implementation of page value calculation using the AVX2 instruction set. Conveniently, one AVX2 register exactly fits eight 32-bit timestamps, which is the default number

[3] This achieves the same effect as the ACE [17] technique.

```
// Table of precalculated quotients i Equation (1)
// with dampening of 0.1 from Figure 6
float iQuotient[] = {{0.1, 8, 7, 6, 5, 4, 3, 2},
                     {2, 0.1, 8, 7, 6, 5, 4, 3},···}

float PageTracker::PVaccess()
   uint8_t  head = accessHead.load()
   uint32_t     now = globalTrackerTime.load()
   __m256i      ts8 = _mm256_loadu_si256(accessLog)
   __m256i     now8 = _mm256_set1_epi32(now)
   __m256i ageInt8 = _mm256_sub_epi32(now8, ts8)
   __m256    age8  = _mm256_cvtepi32_ps(ageInt8)
   __m256      i8 = _mm256_loadu_ps(iQuotient[head])
   __m256 subfreq8 = _mm256_div_ps(i8, age8)
   return           _mm256_reduce_max_ps(subfreq8)
```

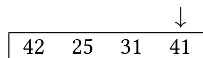
Listing 3: Evaluate a frequency

of read timestamps for each page. This makes computing subfrequencies straightforward, with one exception: due to the cyclic implementation of insertion described earlier in this section, i values depend on the current access head position. We solve this through a constant lookup table, which pre-computes i values and includes the first-access dampening factor.

Background Eviction. Prefetching and SIMD-based page value calculation make sampling many candidate pages efficient. Also note that the algorithm shown in Listing 2 does not have to be executed by worker threads. Indeed, in LeanStores current implementation eviction is done by dedicated background threads.

5.3 Access on Inconsistent Data

Using no global lock on the PageTracker can lead to access on inconsistent data where the *accessHead* (↓) points to a wrong position one behind its real head:



Concurrent Tracking Executions. If two concurrent tracking executions (Listing 1) occur this represents no problem. Because of our code access, we ensured that in a tracking execution the accessHeads read (line 18) and write (line 24) shield the other reads and writes, forcing a tracking execution to write only on a specific position determined by its first action. Therefore concurrent executions reading the same accessHead value can only interfere with each other by reading different values from globalTrackerTime and storing it to the same new position. Either value stored to this position will lead to a consistent timestamp log. Concurrent executions reading different accessHead values from a concurrency with a third execution can lead to a race on writing a different new value to the accessHead. If the execution reading the newer accessHead value succeeds it leads to a consistent timestamp log. If the other execution succeeds, the execution with the newer accessHead value wrote a timestamp to a position one ahead of the current head as seen in our example above. This situation is extremely rare, as it can only occur if three tracking executions are executed concurrently and the first finishing and the last one starting are in a different epoch. As already three concurrent executions happened, it is extremely likely that a fourth execution will happen soon. As the false value is never read by the fourth execution, it has no effect on it and the false value will be overwritten soon resulting in consistent data.

Concurrent Track and PVaccess Executions. Because PVacces Listing 3 only reads the data, access to inconsistent data can only happen as a result of a) inconsistent data by three concurrent tracking executions as presented above, or b) a PVaccess calculation is executed concurrent to a track execution and reads the new timestamp at the new accessHead position before accessHead is incremented. Therefore both inconsistency are only temporary and lead to PVaccess reading data as shown in the example. In this situation, the least recent access has a higher timestamp than expected and its computed subfrequency will be extremely high, preventing replacement of this page. This is a good outcome because the concurrent time tracking access to the page would have led to a high value in the head position, also preventing replacement.

6 SYSTEM EVALUATION

Through simulation, Section 4 demonstrated that, in comparison with state-of-the-art competitors, WATT is more effective at minimizing both read and write I/O. This corresponds to the first two goals stated in the introduction (Replacement Effectiveness and Write Awareness). To evaluate WATT, we integrated it into LeanStore. This allows us to show in Section 6.2 that WATT achieves I/O effectiveness not just in a simulation, but in a real system. Section 6.3 and Section 6.4 then focus on CPU efficiency and scalability, respectively. Finally, we show end-to-end performance results in Section 6.5.

6.1 Experimental Setup and Workloads

Competitors. To evaluate WATT, we integrated it in LeanStore (Section 5) together with random eviction (Random) and Hyperbolic Caching (Hyperbolic). Random was implemented like WATT, only without the page value comparison. For Hyperbolic, we only modified the access tracking and page value calculation in comparison with WATT. Therefore, both offer the same optimizations presented for WATT, like prefetching and asynchronous writing. In addition, we used *LeanEvict*, the reference replacement strategy of LeanStore, for comparison. LeanEvict was tuned by running a grid search over its parameters.

Hardware. The experiments were performed on a Linux 5.15 system with an AMD EPYC 7713 (2.0 GHz (base), 3.675 GHz (max), 64 cores, 128 hardware threads), 512 GB of main memory and a PCIe 4.0 attached Samsung PM1733 U.2 SSD of 3.8TB as storage.

Page Eviction in LeanStore. In LeanStore, page eviction is outsourced to a special page evictor thread. It sleeps if there are enough free pages available for the worker threads. If not enough empty pages are available, it enters the eviction loop (See Listing 2).

YCSB Workload. To measure the different aspects of WATT, LeanEvict, Hyperbolic, and Random, we used the *YCSB* and *TPC-C* benchmarks in different configurations. The YCSB experiment performs Zipfian distributed skewed accesses to 25 and 400 GB datasets. For its skew we selected $\alpha = 0.9$. We performed it with 100 % reads (100R) and 90% reads mixed with 10% writes (90R 10W).

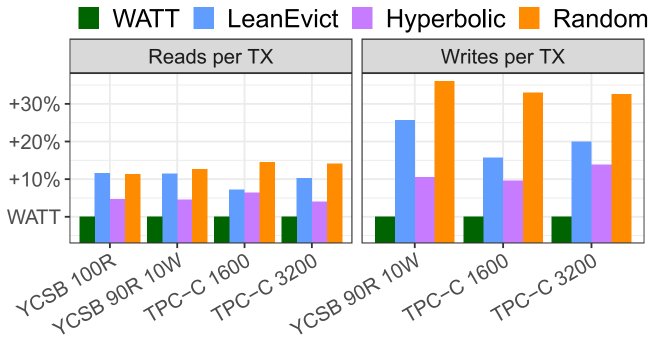


Figure 9: Reads and writes per transaction

TPC-C Workload. TPC-C offers a more complicated online transaction processing benchmark with multiple transaction types and a complex database. The dataset size depends on the number of warehouses and grows in size during the benchmarks execution. We use datasets with 50 warehouses resulting in 9 GB, 1600 warehouses with 264 GB, and 3200 warehouses leading to 588 GB of data.

6.2 Replacement Effectiveness (Goal 1) and Write Awareness (Goal 2)

As shown in the simulation, WATT outperforms all other strategies in terms of effectiveness (Figure 1) and write awareness (Figure 2). We now evaluate, whether these properties are also achieved in a real system.

Experiments. To measure effectiveness and write awareness we ran LeanStore with multiple datasets on small RAM sizes. We ran the YCSB experiments with a 400 GB dataset and TPC-C experiments with 1600 (264 GB) and 3200 warehouses (588 GB). Each experiment was performed with a freshly generated dataset from an actively air-cooled SSD on an empty 8 GB buffer pool (RAM). We used 120 worker threads supported by up to eight page evictor threads and ran it twice for 20 minutes. Transactions and I/O measurements were taken once every second leading to at least 2400 measurements per experiment. Aggregation of measurements was performed by taking the median value of each measurement type separately. The results averaged over all page evictors are shown in Figure 9.

YCSB 100R. First, we focus on YCSB 100R and read only effectiveness in workloads based on a simple Zipf distribution. This workload is quite similar to *ZipfRO* in Section 4. Here, a good eviction strategy should be able to determine the static frequency of a page and keep high frequent pages in RAM and have less *reads per transaction*. In this experiment, Random and LeanEvict were unable to determine static frequencies of pages. Hyperbolic can determine some pages frequencies, but WATT, with its processing of subfrequencies, is able to determine the pages frequencies more accurately, needing the fewest reads per transaction. This leads to 5% more reads per transaction for Hyperbolic, 11% for Random, and 12% more reads per transaction for LeanEvict compared to WATT.

YCSB 90R 10W. To see the effect of writes on the eviction strategies, we added 10% writes to the experiment, while keeping everything else the same. These additional writes slow down the eviction capability as the page evictor thread now additionally has to handle writes. Similar as with *YCSB 100R*, WATT is able to improve *reads per TX*. Additionally, the write awareness helps WATT to reduce *writes per TX* significantly more than *reads per TX*. In comparison to WATT, LeanEvict uses 26%, Hyperbolic 11%, and Random 36% more writes per transaction.

TPC-C – 1600 Warehouses – 264 GB. TPC-C has a more complex access patterns than YCSB. In this experiment LeanEvict needed 7% more reads and 15% more writes per transaction than WATT. Hyperbolic Caching had an overhead of 6% in reads and 10% in writes per transaction compared to WATT, and Random used 14% more reads and 33% more writes per transaction than WATT.

TPC-C – 3200 Warehouses – 588 GB. Working on a bigger dataset increased the difficulty and amount of page writes and evictions per transaction, because only a smaller fraction of the data was able to fit into RAM. In comparison to WATT, LeanEvict needed 10% more reads and 19% more writes per transaction. Hyperbolic had an overhead of 4% in reads and 13% in writes compared to WATT, and Random used 14% more reads and 32% more writes per transaction compared to WATT.

Conclusion. As expected, Random offers the worst effectiveness and write awareness. LeanEvict performs better but is beaten by Hyperbolic. WATT can outperform all other evaluated strategies. On the YCSB benchmarks, LeanEvict and Random perform quite similar in not being able to detect page access frequencies. Hyperbolic was able to notice frequency differences in pages. However WATT had a clear advantage in effectiveness of at least 5% in reads and 10% in writes per transaction. For the TPC-C benchmarks, the differences in effectiveness varied slightly with dataset size. LeanEvict and Hyperbolic were able to reduce reads and writes compared to Random, but performed worse than WATT. The results are similar to the simulated results in Figure 1. We can conclude that the implementation of WATT is as effective and write aware as shown in the simulation.

6.3 CPU Efficiency (Goal 3)

To evaluate CPU efficiency, we measured the performance of a single page evictor thread as well as the impact of updating the tracking structure with one worker thread.

Experiments. To measure the performance of one page evictor, we reused the experiments for effectiveness and write awareness, using only one page evictor. Here the important measurement is the number of evictions performed per second (Figure 10). To measure the overhead of updating the tracking structure the worker thread, we ran it isolated in some in-memory experiments. We evaluated an YCSB and TPC-C experiment on an in-ram dataset six times for five minutes on 60 GB of RAM with one worker thread. YCSB 90R 10W used a 25 GB dataset while TPC-C was performed with 50 warehouses, growing from 9 to 22 GB in size. Figure 11 shows the CPU statistics obtained.

Page Evictor – YCSB. As expected, Random, as a no-overhead strategy, is able to perform the most evictions per second with one thread, because no data structure or function has to be evaluated before evicting a page. Due to the easier to calculate value function, Hyperbolic can outperform WATT by a few percentages. However, this experiment additionally shows how well WATT can close the

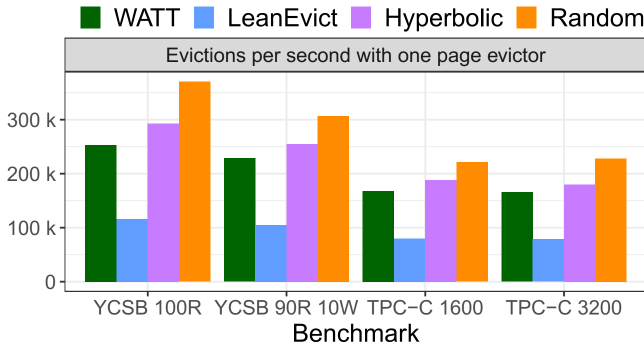


Figure 10: CPU efficiency of one Page Evictor thread

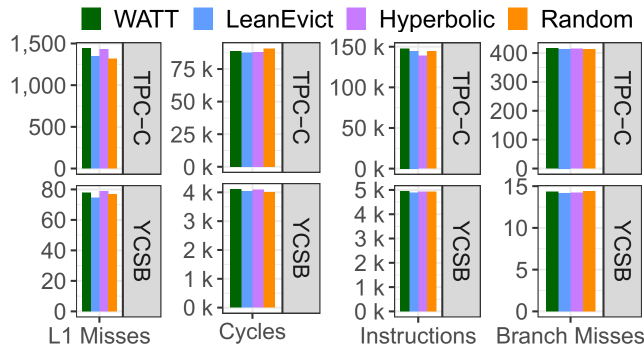


Figure 11: CPU efficiency per TX of one worker thread

performance gap between Random and LeanEvict. While LeanEvict processes 69% less evictions per second than Random, WATT handles only 32% less, more than halving the gap. Hyperbolic caching can close the gap even further to only 21%. With processing dirty evicts in *YCSB 90R 10W* WATT was able to close the gap between LeanEvcit and Random even further from 66% to 25%. Hyperbolic only leaves a gap of 17% to Random.

Page Evictor – TPC-C. In our smallest TPC-C experiment with 1600 warehouses WATT can reduce the performance gap between LeanEvict and Random from 63% to 24%. Hyperbolic can reduce it even further to 15%. With 3200 warehouses the gap can be reduced from 65% in LeanEvict to 27% with WATT. Hyperbolic can once again reduce the gap even further to 21%.

Page Evictor – Conclusion. The comparison of Random, Hyperbolic, and WATT show the effect no page value calculation, a quite simple, and our carefully designed page value calculation (See Section 5.2) have on the eviction performance of a single thread. WATT achieves at least two times as many evictions per second compared to LeanEvict, halving the gap to Random. Therefore, WATT is highly efficient.

Effect on Worker Threads. Figure 11 shows different measurements of our in-memory experiments like instructions, cycles, L1 misses, and branch misses per transaction. By comparing each measurement, we see that there are just small differences between those four algorithms, all just at noise level.

Worker Thread – Conclusion. As LeanEvict and Random are by design zero-overhead strategies (for the workers in an in-memory

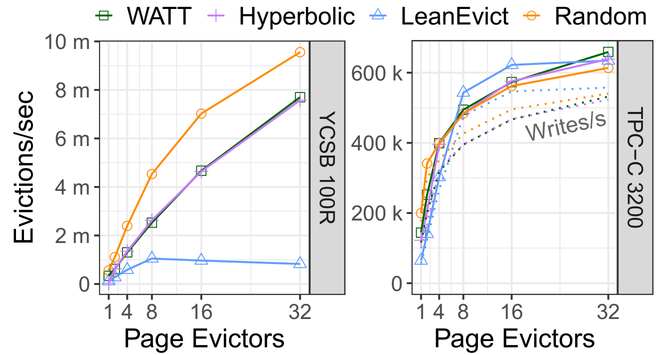


Figure 12: Multi-core scalability

situation), WATT can also be considered a low-overhead strategy. Its low overhead is due to the optimizations described in Section 5.1: accessing a page takes only 6 instructions to check if the epoch is already tracked, and 6 additional instructions to eventually update the epoch. These few instructions are overshadowed by the huge number of instructions required for accessing and working with data from a page.

6.4 Multi-Core Scalability (Goal 4)

Because it is difficult to evaluate the scalability of a single software component without influences from the others, we performed experiments measuring the multi-core scalability of page eviction in isolation.

Experiments. Instead of running page evictor threads and worker threads simultaneously, we ran them consecutively in different phases. During the *work phase* we let the used RAM grow to 190 GB by 120 worker threads before reducing it in an *evict phase* to 180 GB by running up to 32 page evictor threads. For the YCSB experiments we used a 400 GB dataset, and TPC-C was performed with 3200 warehouses. The experiments were run twice for 30 minutes. Figure 12 shows the *evictions per second* during the *evict phase.*

YCSB. In the read-only YCSB 100R, each strategy only has to evict pages without writing changes. Therefore, they scale very well. WATT performs quite similar to Hyperbolic, leaving a small gap to the no overhead eviction strategy Random, showing its high scalability. Only LeanEvict has scalability issues and is not able to evict more than 1 million pages per second.

TPC-C. Because this benchmark generates dirty pages, all strategies use a huge amount of writes per second, marked with a dotted line. As dirty pages have to be persisted before eviction, this impacts the performance of all strategies. Apart from that, all strategies scale well and quite similar.

6.5 Overall Performance

The performance of an eviction strategy depends on many factors, like efficiency, effectiveness, write awareness, and scalability. We previously showed how WATT performs on each of these factors independently. To combine them, we measured *transactions per second* in our experiments used for effectiveness and write awareness

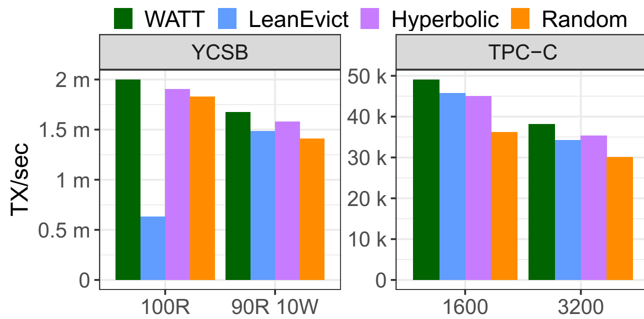


Figure 13: Overall Performance

with four or more page evictors, and present the averaged results in Figure 13.

YCSB. Here LeanEvict shows its YCSB 100R scalability issues. Random shows quite good results based on its multi-core scalability. Hyperbolic is able to outperform Random by using its better effectiveness, but WATT can outperform all others by combining its good scalability with its superior effectiveness. Changing to YCSB 90R 10W resolves the scaling problem of LeanEvict and makes it comparable to the other strategies. Without this issues LeanEvict is able to outperform Random. Hyperbolic performs more transactions per second than LeanEvict and Random, but is still outperformed by WATT with 6% more transactions per second.

TPC-C. On TPC-C experiments Random performed worse than the other strategies with at least 22% less transactions compared to WATT. With 1600 warehouses LeanEvict was able to perform slightly more transactions than Hyperbolic, but with 3200 warehouses this was the other way around. WATT can outperform both of them with 7% more transactions per second.

7 SUMMARY

In this paper, we present WATT, a low-overhead, effective, write aware, efficient, and scalable page replacement strategy designed for modern hardware. Its replacement effectiveness is based on timestamp tracking, subfrequencies and their aggregation, while its efficiency and scalability depend on random sampling with lazy evaluation and a careful implementation design. We evaluated WATT in terms of *effectiveness*, *write awareness*, *CPU efficiency*, and *multi-core scalability*. Effectiveness and write awareness were evaluated against three classical and six state-of-the-art replacement strategies in a simulation and three high-efficient strategies in LeanStore. WATT has the best effectiveness and offers a substantial write reduction compared to all other strategies. Comparing it against LeanEvict, a low-overhead and efficient LRU approximation, Hyperbolic Caching, a highly scalable memory caching strategy, and the no-overhead strategy Random, WATT showed its high efficiency and multi-core scalability. We can therefore conclude that WATT is a highly effective, highly write aware, highly CPU efficient, and highly scalable replacement strategy.

REFERENCES

[1] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW (LNI)*, Vol. P-331. 259–281.
[2] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *USENIX ATC*. 499–511.
[3] K. Delaney, B. Beauchemin, C. Cunningham, J. Kehayias, B. Nevarez, and P.S. Randal. 2013. *Microsoft SQL Server 2012 Internals*. Microsoft Press.
[4] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.
[5] IBM. 2023. DB2 – Choosing a page-stealing algorithm. Retrieved July 10, 2023 from https://www.ibm.com/docs/en/db2-for-zos/12?topic=pools-choosing-page-stealing-algorithm
[6] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold (Eds.), Vol. 360. 24–35.
[7] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). 439–450.
[8] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. 2008. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *IEEE Transactions on Consumer Electronics* 54, 3 (2008), 1215–1223.
[9] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.
[10] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.
[11] MariaDB. 2023. InnoDB Buffer Pool. Retrieved July 10, 2023 from https://mariadb.com/kb/en/innodb-buffer-pool/
[12] Diego Mazzeo. 2023. Page Replacement Algorithm: Clock Sweep. Retrieved July 10, 2023 from https://www.interdb.jp/pg/pgsql08.html#_8.4.4.
[13] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*.
[14] MySQL. 2023. Buffer Pool. Retrieved July 10, 2023 from https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html
[15] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *SIGMOD*. 297–306.
[16] Oracle. 2023. Buffer Replacement Algorithms. Retrieved July 10, 2023 from https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/memory-architecture.html#GUID-D1429BAA-6543-4B34-93DB-C8F33D497B53
[17] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. (2023). Retrieved July 10, 2023 from https://cs-people.bu.edu/mathan/publications/icde23-papon.pdf
[18] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. 2006. CFLRU: a replacement algorithm for flash memory. In *International conference on Compilers, architecture and synthesis for embedded systems*. 234–241.
[19] PostgreSQL. 2023. Release Notes PostgreSQL 8.0.2. Retrieved July 10, 2023 from https://www.postgresql.org/docs/release/8.0.2/
[20] Samsung. 2023. Samsung PCIe Gen 4-enabled PM1733 SSD. Retrieved July 10, 2023 from https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735/mzwlj3t8hbls-00007/
[21] SciPy. 2023. SciPy Documentation for Zipfian. Retrieved July 10, 2023 from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.zipfian.html
[22] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *EDBT*. 17–28.
[23] WiredTiger. 2023. Eviction in WiredTiger. Retrieved July 10, 2023 from https://source.wiredtiger.com/11.0.0/eviction.html