The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware

Tiemo Bang
TU Darmstadt & SAP SE
tiemo.bang@cs.tu-darmstadt.de

Norman May
SAP SE
norman.may@sap.com

Ilia Petrov
Reutlingen University
ilia.petrov@reutlingen-university.de

Carsten Binnig
TU Darmstadt
carsten.binnig@cs.tu-darmstadt.de

Abstract

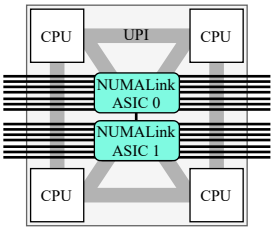
In this paper, we set out the goal to revisit the results of "Starring into the Abyss [...] of Concurrency Control with [1000] Cores" [27] and analyse in-memory DBMSs on today's large hardware. Despite the original assumption of the authors, today we do not see single-socket CPUs with 1000 cores. Instead multi-socket hardware made its way into production data centres. Hence, we follow up on this prior work with an evaluation of the characteristics of concurrency control schemes on real production multi-socket hardware with 1568 cores. To our surprise, we made several interesting findings which we report on in this paper.

ACM Reference Format:
Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. 2020. The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware. In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3399666.3399910

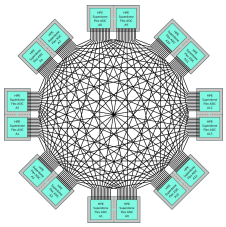
1 Introduction

We are now six years after "Starring into the Abyss [...] of Concurrency Control with [1000] Cores" [27], which presented an evaluation of concurrency schemes for in-memory databases on simulated hardware. The speculation of the authors at that time was that today we would see hardware providing single-chip CPUs with 1000 cores. However, so far reality is different [11, 20]. Instead of single-chip CPUs with 1000s of cores, multi-socket machines are prevalent and made their way into production data centres, indeed offering 1000s of cores. Accordingly, in-memory DBMS are facing not only challenges of massive thread-level parallelism, such as coordination of hundreds of concurrent transactions as predicted by [27], but large multi-socket systems also expose in-memory DBMS to further challenges, such as deep NUMA topologies connecting all CPUs and their memory as in Figure 1.

In this paper, we set out the goal to bring in-memory DBMS to a 1000 cores on today's multi-socket hardware, revisiting the results of the simulation of [27] based on the original code, which the authors generously provide as open source. That is, we follow up on [27] with an evaluation of the characteristics of concurrency control schemes on real production multi-socket hardware with 1568 cores. To our surprise, we made several interesting findings:



(a) Chassis topology



(b) NUMALink topology

Figure 1: System topology of the HPE SuperdomeFlex [10, 12].

(1) First, the results of running the open-source prototype of [27] on today's production hardware revealed a completely different picture regarding the analysed concurrency schemes compared to the original results on simulated hardware. (2) Afterwards, in a second "deeper look" we analysed the factors leading to the surprising behaviour of the concurrency control schemes observed in our initial analysis, where we then find further surprises such as unexpected bottlenecks for workloads with a low conflict rate. (3) Based on these findings, we finally revisited the open-source prototype of [27] and reran the evaluation with our optimised version of DBx1000, which we think helps to establish a clear view on the characteristics of concurrency control schemes on real large multi-socket hardware.

In the following, we first report the concrete setup used in this paper (Section 2) and then discuss our findings (Section 3-5).

2 Setup for our Experimental Study

In the following, we provide a brief overview of the concurrency control (CC) schemes, the hardware as well as the benchmarking environment used in our evaluation.

Bouquet of Concurrency Control: Table 1 summarises the evaluated CC schemes. They range from lock-based CC, with diverse mechanisms against deadlocks, to timestamp-ordering-based CC, including multi-versioning, 2-versioning, coarse locking, and advanced ordering. For details on these CC schemes, we to their original publications [2, 3, 16, 19, 25, 28] and to [27]. The first seven CC schemes in Table 1 correspond to the prior evaluation in [27]. We further include the more recent schemes *SILO* [25] and *TICTOC* [28], not included in the original study. Unfortunately, *TIMESTAMP* from [27] has a fatal bug in the latest version of the prototype, so we excluded this scheme from our experiments.

DAMON'20, June 15, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8024-9/20/06…$15.00
https://doi.org/10.1145/3399666.3399910

DL_DETECT	2PL with deadlock detection [2]
NO_WAIT	2PL with non-waiting deadlock prevention [2]
WAIT_DIE	2PL with wait-and-die deadlock prevention [2]
MVCC	Multi-version T/O [3]
OCC	Optimistic concurrency control [19]
H-STORE	T/O with partition-level locking [16]
TIMESTAMP	Basic T/O algorithm [2]
SILO	Epoch-based T/O [25]
TICTOC	Data-driven T/O [28]

Table 1: Bouquet of concurrency control schemes.

NUMA level	Latency	Bandwidth
local	102 ns	95.1 GB/s
1 hop UPI	150 ns	17.2 GB/s
2 hop UPI	208 ns	16.6 GB/s
NUMALink	380 ns	11.2 GB/s

Table 2: Memory access latency and bandwidth by NUMA level as measured via Intel MLC [26].

Real Hardware with a 1000 Cores: The prevalent hardware in production today offering 1000 cores are large multi-socket machines [11, 20]. As shown in Figure 1, such hardware connects many CPUs to a single system rather than hosting many cores on a single CPU. Our *HPE SuperdomeFlex* system [11] contains 28 *Intel Xeon 8180* CPUs. It groups four CPUs into hardware partitions (chassis), which are then joined together, forming a single cache coherent system with a total of 1568 logical cores and 20 TB of DRAM. As shown in Figure 1a, within the chassis each CPU connects to two neighbouring CPUs and to a *NUMALink* controller via UPI links. Then the *NUMALink* controllers couple all chassis in a fully connected topology (Figure 1b), yielding four levels of NUMA with performance properties summarised in Table 2.

Comparing this hardware to potential many-core hardware as simulated in [27] reveals that this multi-socket setup for 1000 cores differs in many aspects. Importantly, one similarity of today's hardware to the simulated architecture of [27] is that both communicate and share cache in a non-uniform manner via a 2D-mesh on the chip [8] (and UPI beyond), such that the cores use the aggregated capacity to cache data but need to coordinate for coherence. This non-uniform communication is an important hardware characteristic, as it can amplify the impact of contention points in the CC schemes on any large hardware (multi-socket and many-core). Otherwise, the simulation differs from today's hardware, since it assumed low-power and in-order processing cores clocked at 1GHz, cache hierarchies with only two levels, and cache capacities larger than today's caches. Notably, it simulates the DBMS in isolation without an OS, disregarding overheads and potential side effects of OS memory management, scheduling etc., omitting essential aspects of real systems [7, 23] like ours.

Benchmarking Environment: As [27], we evaluate the CC schemes mentioned before on our multi-socket hardware with the TPC-C benchmark [24] as implemented in the latest version of DBx1000[1]. This version of DBx1000 includes the extended set of CC schemes as mentioned before and bug fixes, beyond the version used in the original paper [27]. For running the benchmarks, we use the given default configuration of DBx1000. This configuration defines the TPC-C workload as equal mix of *New-Order* and *Payment* transactions covering 88% of TPC-C with standard remote warehouse probabilities (1%[2] and 15%). This configuration partitions the TPC-C database by warehouse (WH) ID for all CC schemes. Based on this configuration, we specify four warehouses for the high conflict TPC-C workload and for the low conflict workload we specify 1024 or 1568 warehouses, maintaining the ratio of at most one core per warehouse as [27].

An interesting first observation was, that the TPC-C implementation of DBx1000 does not include insert operations, presumably due to the mentioned limitations of the simulator, e.g., memory capacity and no OS. In this paper, we first start with the very same setup, but later we also enable insert operations in the evaluation after taking a first look at the CC schemes. As minor extension, we added a locality-aware thread placement strategy to DBx1000 for all experiments in this paper, which exclusively pins DBMS threads to a specific core. For scaling the DBMS threads in our experiments, we use the minimal number of sockets to accommodate the desired resources, e.g., 2 sockets for 112 threads. Otherwise OS and NUMA effects would dominate the overall results. Note that as consequence of this thread placement strategy, *cores* and *threads* equally refer to a single execution stream (i.e., a worker) of the DBMS. In our initial experiments (Sections 3-4), we use up to 1024 cores like the simulation in [27]. Only after our optimisations, we leverage the full 1568 cores of our hardware, showing the scalability of our optimised DBMS in Section 5.

3 A First Look: Simulation vs. Reality

We now report the results of running the DBx1000 prototype directly on the multi-socket hardware as opposed to a simulation.

3.1 The Plain Results

Figures 2 and 3 display the throughput of TPC-C transactions for *4 warehouses* and *1024 warehouses*, i.e., high and low conflict OLTP workloads. *On the left* of each figure are the original simulation results [27] and *on the right* are our results on a real multi-socket hardware. We first report the plain results. Then we break down where time is spent in the DBMS to better understand our observations.

We first look at the results for 4 warehouses as shown in Figure 2. Overall, it is obvious that the absolute throughput differs due to the characteristics of the CPUs in the simulation and our hardware, e.g., low-power 1 GHz cores versus high-power 2.5 GHz cores, which can be expected and therefore only the relative performance of the CC schemes matters. In the following, we now discuss some similarities but also significant differences.

First, comparing the simulation and the real hardware in Figure 2, we see that the CC schemes *HSTORE*, *MVCC*, and *NO WAIT* show similar trends. That is, these CC schemes have a similar thrashing point in the simulation and the real hardware, i.e., *HSTORE* at 4 to 8 cores and *MVCC* as well as *NO WAIT* at 56 to 64 cores. After the respective thrashing point, these CC schemes degrade steeper on the multi-socket hardware, which can be linked to the additional

[1] https://github.com/yxymit/DBx1000/tree/b40c09a27d9ab7a4c2222e0ed0736a0cb67b7040

[2] Based on a typo, the original paper [27] states 10% instead of 1%

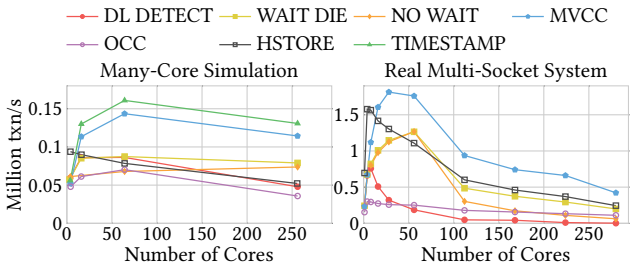


Figure 2: Throughput of TPC-C high conflict workload (4 WH) in original simulation [27] and on real multi-socket hardware.

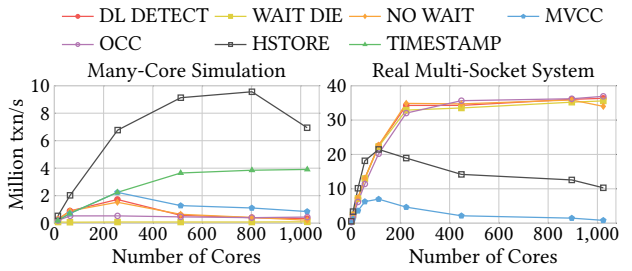


Figure 3: Throughput of TPC-C low conflict workload (1024 WH) in original simulation [27] and on real multi-socket hardware.

NUMA effect of the multi-socket hardware appearing beyond 56 cores. For the other CC schemes the results for the simulation and real hardware differ more widely, especially the diverging behaviour of the pessimistic CC schemes sticks out. Considering these pessimistic CC schemes, *DL DETECT* behaves broadly different already degrading at 8 cores rather than 64 cores and *WAIT DIE* performs surprisingly close to *NO WAIT*. In the Section 3.2, we analyse the time breakdown of this experiment to explain these results. It reveals characteristic behaviour of the individual CC schemes, despite the diverging throughput in the simulation and the multi-socket hardware.

Next, we look at the low contention TPC-C workload (1024 warehouses) in Figure 3. The results here present fewer similarities of the many-core simulation and the multi-socket hardware, i.e., only the slope of the *MVCC* scheme is similar. Additionally, *DL DETECT* and *NO WAIT* stagnate at high core counts (>224) in the simulation and on the multi-socket hardware. In contrast, *HSTORE* performs worse on the multi-socket hardware than in the simulation. It is slower than the pessimistic CC schemes and *OCC* from >112 cores. Also, *OCC* and *WAIT DIE* achieve higher throughput on the multi-socket hardware, now similar to *DL DETECT* and *NO WAIT*. Moreover, unexpectedly in this low conflict workload, *MVCC* is significantly slower than *OCC* and the pessimistic CC schemes, which is caused by high overheads of this scheme as we discuss next.

Useful	Time usefully executing application logic and operations on tuples.
Abort	Time rolling back and time of wasted useful work due to abort.
Backoff	Time waiting as backoff after abort (and requesting next transaction to execute).
Ts. Alloc.	Time allocating timestamps.
Index	Time operating on hash index of tables including latching.
Wait	Time waiting on locks for concurrency control.
Commit	Time committing transaction and cleaning up.
CC Mgmt.	Time managing concurrency control other than prior categories, e.g., constructing read set.

Table 3: Time breakdown categories.

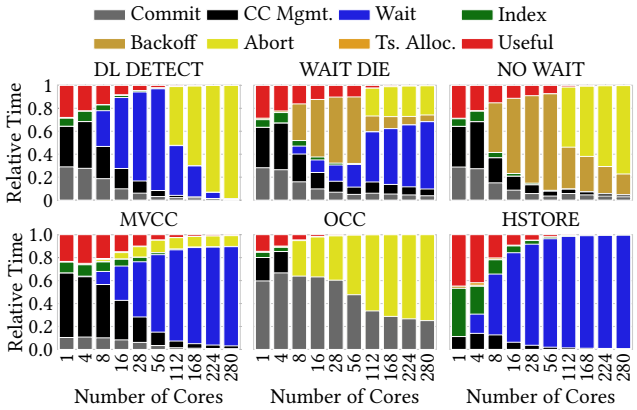


Figure 4: Breakdown of relative time spent for high conflict (4 WH) TPC-C transactions on multi-socket hardware.

Insight: The initial comparison of concurrency control schemes on 1000 cores presents only minor similarities between the simulation and our multi-socket hardware with surprising differences in the behaviour of the CC schemes mandating further analysis.

3.2 A First Time Breakdown

For deeper understanding of the observed behaviour of the CC schemes, we now break down where time is spent in processing the TPC-C transactions on the multi-socket hardware. For this purpose, we apply the breakdown of [27] categorising time as outlined in Table 3. For each CC scheme, Figures 4 and 5 break down the time spent relative to the total execution time of the TPC-C benchmark with a bar for each core count.

The time breakdown of 4 warehouses in Figure 4 neatly shows the expected effect of conflicting transactions and aborts for increasing core counts under high conflict workload. That is, most CC schemes result in high proportions of *wait*, *abort*, and *backoff* as soon as the number of cores exceeds the number of warehouses (>4 cores), yielding nearly no *useful work* at higher core counts. Only the *wait* time of *HSTORE* immediately grows at 4 cores concurrently executing transactions, such that *HSTORE* appears more sensitive to conflicts for this workload.

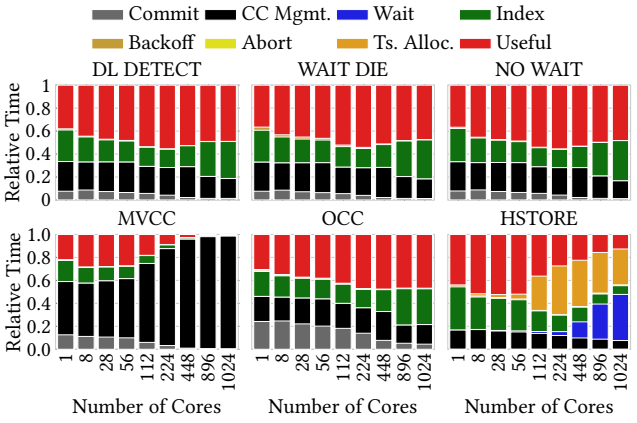


Figure 5: Breakdown of relative time spent for low conflict (1024 WH) TPC-C transactions on multi-socket hardware.

Remarkably, textbook behaviour of the specific schemes becomes visible in the breakdown: Starting with *DL DETECT*, its *wait* time increases with increasing number of concurrent transactions as expected, following the increasing potential of conflicts between concurrent transactions. Different from *DL DETECT*, *WAIT DIE* spends more time *backing off* and *aborting* due to its characteristic aborts after a short wait time (small wait proportion). Instead *NO WAIT* solely *backs off* without waiting, spending even more time on *aborted* transactions. The optimistic *MVCC* waits on locks during validation, such that its break down shows similar *wait* times like *DL DETECT*. Finally, for *OCC* we can see that the high *abort* portion reflects its sensitivity to conflicts while the high *commit* portion stems from high costs for cleaning up temporary versions at commit time.

Having observed this "expected" behaviour of the CC schemes under high conflict, we now analyse the unexpected behaviour under low conflict (1024 warehouses) as shown in Figure 5. Against the expectation, most CC schemes spend considerable amount of time to manage concurrency (black and grey area) such as lock acquisition (except *HSTORE* which we discuss later). For these schemes this results in at most 50% of useful work (red area). Staggeringly, *MVCC* which actually should perform well under low conflicting workloads, spends almost no time with *useful work* despite the low conflict in the workload, i.e., <10% *useful work* from 224 cores. In fact, the low conflict is visible in the overall little time spent *waiting* or *aborting*. Consequently, the slowdown compared to pessimistic CC schemes does not stem from wasted work but from pure internal overhead in execution of this CC scheme under high core counts.

In contrast, we observe for *HSTORE* an increasing impact of *timestamp allocation* and *waiting time*. While *timestamp allocation* is used by the other schemes as well, the relative overhead for *HSTORE* is the highest since lock acquisition in *HSTORE* is cheap. In fact, the authors of [27] did analyse different timestamp allocation methods in their paper but chose *atomic increment* as a sufficiently well performing method that is a generally applicable option when there is no specialised hardware available. However, as we can see this choice is not optimal for multi-socket hardware. Moreover,

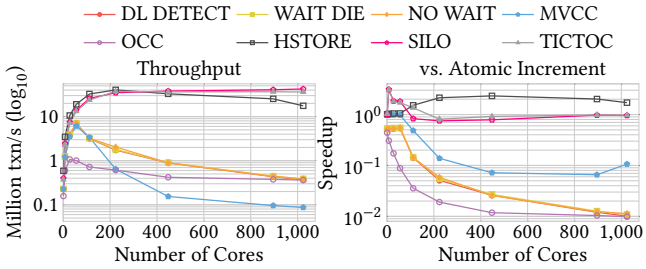


Figure 6: Throughput of TPC-C with 1024 warehouses for timestamp allocation with hardware clock.

we attribute the increasing *waiting time* of *HSTORE* to its coarse-grained partition locking to sequentially execute transactions on each partition. This partition-level locking causes a higher overhead if more cores are used since this leads to more conflicts between transactions as shown in prior work [18, 21].

Insight: The analysed CC schemes behave differently on the real multi-socket hardware than in the simulation of [27]. For the high conflict workload (4 warehouses), the behaviour on real hardware and the simulation appears more similar, for which the time breakdown confirms the expected characteristics for each CC scheme. However, low conflict workload (1024 warehouses) causes an unexpectedly high CC management overhead in most CC schemes and transactions execute only a limited amount of useful work, except for *HSTORE* where waiting and timestamp allocation dominate.

4 A Second Look: Hidden Secrets

In this section, we now take a "second look" on the factors leading to the surprising behaviour of the CC schemes observed in our initial analysis and discover equally surprising insights.

4.1 Hardware Assistance: The Good?

In a first step, we analyse the benefit of hardware-assisted timestamp allocation over using atomic counters for the real multi-socket hardware. As explained earlier, the *atomic increment* is generally applicable but may cause contention, which efficient and specialised hardware may prevent if available. Fortunately, as already mentioned in [27], timestamp allocation can also be implemented using a synchronised hardware clock as supported by the Intel CPUs [14] in our hardware (*rdtsc* instruction with *invariant tsc* CPU feature). Therefore, we can replace the default timestamp allocation via atomic increment with this hardware clock.

In the following experiment, we analyse the benefit of this hardware assistance for timestamp allocation. Figure 6 shows the throughput of the CC schemes for 1024 warehouses with timestamp allocation based on the hardware clock.

On one hand, *HSTORE* greatly benefits from the hardware clock (as expected) achieving peak throughput of ~ 40 M txn/s with an overall speedup over atomic increment of up to 3x. We now also include *SILO* and *TICTOC* in our results which perform like *HSTORE* except for high core counts as we discuss in the time breakdown analysis below. On the other hand, the remaining CC schemes (*DL*

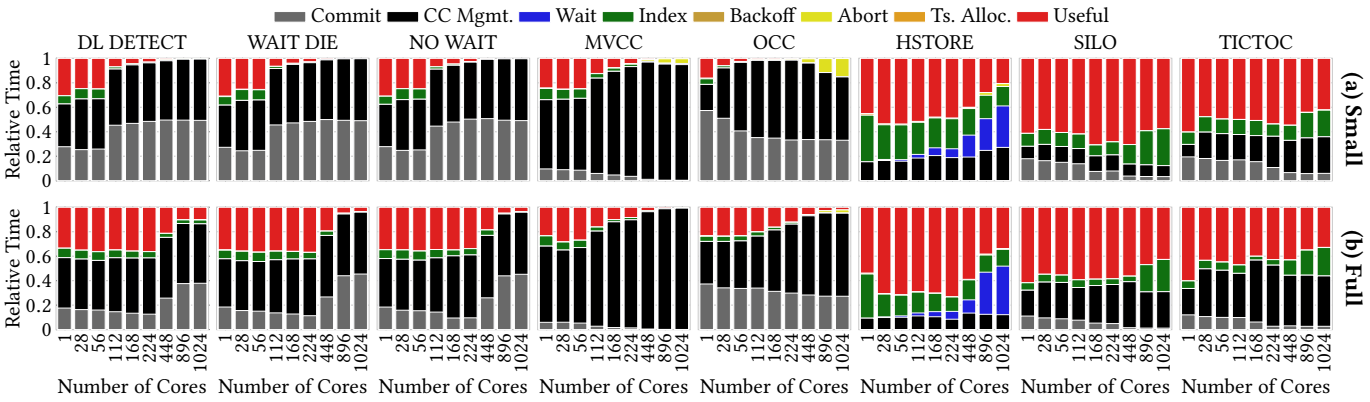


Figure 7: Breakdown of relative time spent processing TPC-C transactions on *small* and *full* schema with 1024 warehouses using timestamp allocation via hardware clock.

DETECT, WAIT DIE, NO WAIT, MVCC, and *OCC*) degrade drastically when using the hardware clock instead of atomic counters. That is, the pessimistic CC schemes *DL DETECT, WAIT DIE*, and *NO WAIT* perform ~ 50% slower within a socket (0.51 - 0.55x speedup for ≤56 cores), after which they degrade to 0.01x speedup at 1024 cores (0.37 - 0.39 M txn/s). Likewise, *MVCC* is stable (~1x speedup) up to 56 cores and its speedup drops to 0.1x when exceeding the single socket. Finally, *OCC* does not benefit from the hardware clock at all (0.44 - 0.01x speedup).

Overall, timestamp allocation based on the hardware clock drastically changes the perspective on the performance of the CC schemes. Now *HSTORE* performs best, meeting the initial observations of [27] (joined by *SILO* and *TICTOC*), whereas the pessimistic schemes, *OCC*, and *MVCC* degrade severely.

For better understanding of these diverse effects of the hardware clock, we again look at the time breakdown shown in Figure 7a (top row). As expected, *HSTORE* now spends no significant time for timestamp allocation anymore (like *SILO* and *TICTOC*). Its *waiting time* still significantly increases as before. This explains the slowdown for >448 cores and corroborates earlier descriptions of *HSTORE*'s sensitivity to conflicts on the partition level as discussed in Section 3.2. An interesting observation is the significant change in the time break down of the other CC schemes. For example, *DL DETECT, WAIT DIE*, and *NO WAIT* show at least double the time spent for *CC Mgmt.* and *committing/cleaning up* (black & grey, bottom two bars) with a sudden increase after 56 cores. *OCC*'s increase of time spent in these categories increases even more drastically with less than 20% of *useful work* at any core count. Only *MVCC* does not show a significant change in this breakdown, since its *useful* time spent was low already.

Profiling these CC schemes reveals contention, that previously was on atomic counters, now results in higher thrashing of latches, despite a latch per row and low conflicts in the workload with 1024 warehouses. Notably, our profiling further reveals more interesting details of the individual CC schemes: The *pthread_mutex* employed in *DL DETECT, WAIT DIE, NO WAIT*, and *OCC* sharply degrades due to NUMA sensitivity of hardware transactional memory [4] used for lock elision and its fallback to robust but costly queuing synchronisation [9] as well as costly interaction with the scheduler of the OS.[3] In contrast, *MVCC* uses an embedded flag as spin latch which is not as sensitive to NUMA but also not robust [6]. Hence, this type of latch shows a slower but also continuous degrading of performance.

Insights: Hardware-assisted timestamp allocation via specialised clocks alleviates contention and leads to better scalability for *HSTORE* (as well as *SILO* and *TICTOC*). However, while introducing hardware-assisted clocks also shifts the overhead in the other schemes, it does not necessarily improve their overall performance as contention moves and puts pressure on other components (e.g., latches), even leading to further degradation of the overall performance.

4.2 Data Size: The Bad?

In the context of this surprisingly high overhead, our second look at the paper [27] brings the following statement to our attention: "Due to memory constraints [...], we reduced the size of [the] database" [27]. Consequently, we are wondering if the staggering overhead is potentially caused by absence of useful work to execute rather than the abundance of overhead in the CC schemes, due to the reduced data size imposed by limited memory capacity of the simulator in [27]. Therefore, we revert the benchmark to the full TPC-C database in the following experiment and report on the surprising effect of the larger data volume.

Figure 8 shows the throughput for the full schema with 1024 warehouses and speedup in comparison to the small schema based on the previous experiment (cf. Figure 6). We measure quite diverse throughput of the CC schemes. Yet, the speedup indicates that two major effects of the increased data volume appear in the same clusters as in the previous experiment but with inverse outcome. The first cluster of *HSTORE, SILO*, and *TICTOC* is slower with the full schema, i.e., 0.2-0.6x, 0.3-0.5x, and 0.2-0.5x, respectively. The second cluster, consisting of the previously "slower" CC schemes, improves inversely to the previously described thrashing points. That is, *DL DETECT, WAIT DIE*, and *NO WAIT* have a speedup of

[3] pthread_mutex is specific to *libc* and OS as well as configurable.

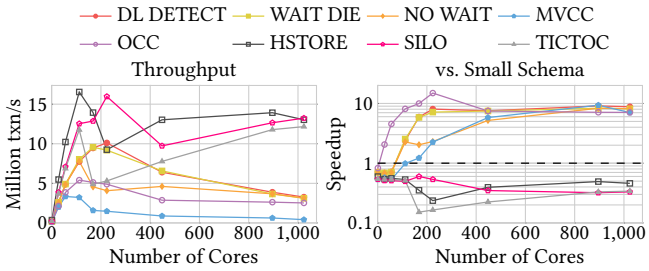


Figure 8: Throughput of TPC-C with 1024 warehouses for small schema size like in the simulation versus full schema size both executed on multi-socket hardware.

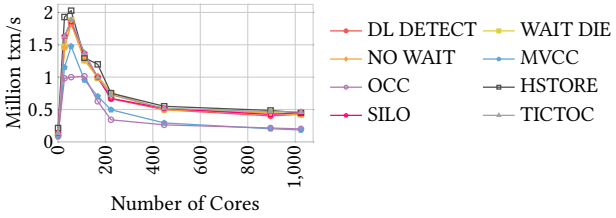


Figure 9: Throughput of TPC-C including inserts with full schema size on multi-socket hardware.

0.7x up to 56 cores, after which they benefit from the full schema with speedups of 2.4-9.1x, 2.5-8.3x, and 2.0-9.3x, respectively. *MVCC* has a speedup of 0.5-0.6x until 56 cores, breaks even (1x) at 112 cores, and then improves with a speedup of 1.2-9.3x. *OCC* has a speedup of 0.8 at 1 core and broadly improves with the full schema with 2.1-14.9x speedup.

The time breakdown in Figure 7b (lower row), presents insights on the causes. As for the CC schemes in the first cluster, *HSTORE* has increased *useful* work, whereas for *SILO* and *TICTOC CC Mgmt.* increases, both indicate increased cost of data movement, as *HSTORE* directly accesses tuples and the other two create local temporary copies in the CC manager. The second cluster also has an increase of *useful* work to some extent, presenting less staggering overhead of CC management at low core counts. Importantly, the sudden increase of *commit* for *DL DETECT*, *WAIT DIE*, and *NO WAIT* is delayed, indicating that latches thrash only from 448 cores (while previously already from 112 cores). For *OCC* the time spent on *commit* also decreases with the larger data volume, but the increase of *CC Mgmt.* due to larger temporary copies still diminishes *useful work*. Only for *MVCC* the time break down does not change significantly.

We attribute these observations to two effects of the larger data volume: The heavier data movement slows down data-centric operations (e.g., tuple accesses or temporary copies), but in turn alleviates pressure on latches preventing thrashing.

Insight: The effect of larger data volumes in the full schema changes the perspective on the CC schemes again, most notably on the differences between the individual CC schemes. Moreover, also the relation of *useful work* and overhead within each scheme changes. Both are caused by larger data volume reducing performance of data movement, but also alleviating pressure on latches.

4.3 Inserts: Facing Reality!

Since the simulator of [27] had limited memory capacity and excluded the simulation of important OS features such as memory management, the TPC-C implementation of DBx1000 did not include insert operations and for comparability we initially excluded these as well. For the last experiment in this section, we now complete the picture of concurrency control on real hardware.

Accordingly, Figure 9 shows the throughput of TPC-C transactions including inserts (as well as all before-mentioned changes) for 1024 warehouses. As we can see, the inserts drastically reduce throughput of all CC schemes and introduce heavy degradation at the socket boundary (56 cores). Even more interesting, all CC schemes perform similarly with inserts included in the transactions. Indeed, profiling indicates execution of insert operations are the hotspot of the TPC-C transactions now, but the causes are orthogonal to concurrency control. The two major hotspots are (1) catalogue lookups to locate tuple fields and (2) memory allocation for new tuples during insert operations.

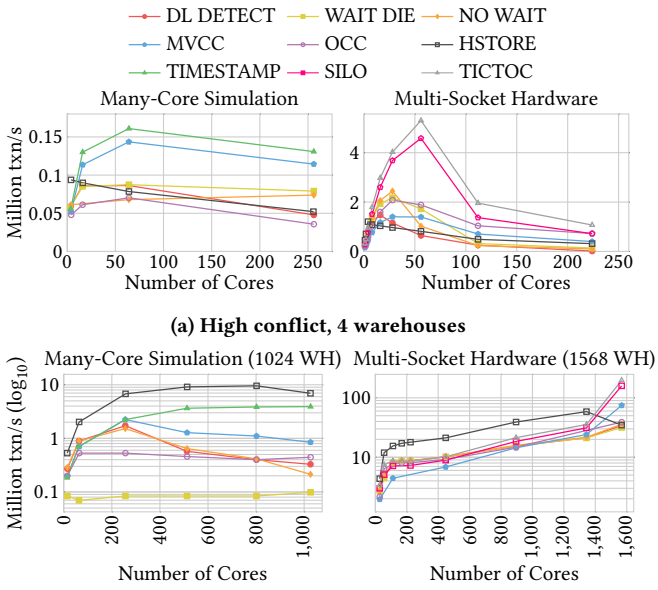
Profiling details show that catalogue lookups cause frequent accesses to L1 and L3 caches. For tuple allocation, profiling details indicate significant time spent in the memory allocator and for OS memory management including page faults. These hotspots are amplified by NUMA in our multi-socket system, since the catalogue is centrally allocated and memory management in Linux is contention- and NUMA-sensitive as well [5]. Consequently, such impact on performance only becomes visible in its full extent on large systems like ours.

Insight: Inserts themselves do significantly affect the performance of the CC schemes in this benchmark. Yet, in all schemes performance is now greatly overshadowed by orthogonal hotspots most notably cache misses of the catalogue and memory allocation during inserts.

5 A Final Look: Clearing Skies

Finally, we take a last step to provide a more optimal handling of inserts in DBx1000 to get a clear view on the characteristics of the CC schemes on large multi-socket hardware. To clear this view, we remove previously identified obstacles and further optimise DBx1000 as well as the implementation of the TPC-C transaction based on state-of-the-art in-memory DBMS for large multi-socket hardware [13, 17, 18].

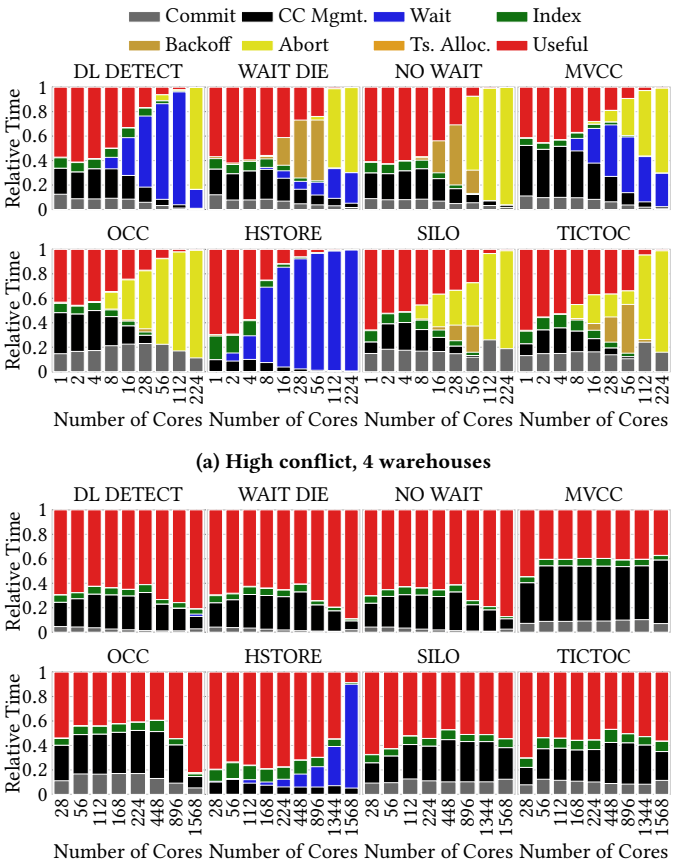
Our optimisations address thrashing (cf. Section 4.1) with a queuing latch and exponential backoff [13]. We optimise data movement (cf. Section 4.2) with reordering and prefetching of tuple and index accesses, a flat perfect hash index, and NUMA-aware replication of the read-only relations [18]. Additionally, for the hotspots identified in Section 4.3, we transform expensive query interpretation (especially catalogue lookups) into efficient query compilation as done by state-of-the-art in-memory DBMS [17] and introduce a thread-local memory allocator that pre-allocates memory, as done in commercial in-memory databases today [11]. Memory alignment is also an interesting trade-off for memory management. On one hand, alignment to cache line boundaries prevents false sharing



(a) High conflict, 4 warehouses

(b) Low conflict, 1024/1568 warehouses

Figure 10: Throughput of TPC-C in original many-core simulation [27] without full schema & inserts and our optimised implementation with full schema & inserts on multi-socket hardware.



(a) High conflict, 4 warehouses

(b) Low conflict, 1568 warehouses

Figure 11: Breakdown of relative time spent processing TPC-C transactions with optimised DBx1000 using full schema and inserts on multi-socket hardware.

and may generally be required by some CPUs. On the other hand, this alignment may amplify memory consumption, as records are allocated as multiples of cache lines, e.g., 64 bytes. DBx1000 aligns to 64 bytes and our allocator does so as well now, because false sharing obliterates performance without alignment. Additionally, we reduce CC overhead for read-only relations and update the deadlock prevention mechanisms to state-of-the-art as recommended by [13].

5.1 The Final Results

With the above optimisations in place, we are finally able to take a clear look at concurrency control for OLTP on large multi-socket hardware. In the following experiment, we repeat our first assessment (cf. Section 3) of the concurrency control schemes under OLTP workload with high and low conflict. Though, this time we utilise our optimised implementation and take the full TPC-C schema as well as inserts in the transactions. Notably, we exercise the whole 1568 cores in this experiment, for which we keep the one-to-one relation of cores to warehouses for the low conflict workload (1568 warehouses), as the TPC-C workload induces significant conflict when concurrent transactions exceed the number of warehouses (cf. Section 3). Accordingly, Figure 10 presents the final throughput for the high conflict and low conflict TPC-C workload. In addition, Figure 11 again details the performance of the CC schemes on the multi-socket hardware with time breakdowns for both workloads.

Starting with throughput of the high conflict workload in Figure 10a (top row), we again observe similar results as reported in our first assessment. The many-core simulation and the multi-socket hardware results show different but reasonable behaviour due to the respective hardware characteristics. The only difference is that now our optimisations further offset throughput on the multi-socket hardware. Additionally, we now include the advanced CC schemes *SILO* and *TICTOC* whose peak throughput remarkably outperform the originally covered CC schemes with 4.6 and 5.3 M txn/s, respectively. Yet, those two CC schemes similarly degrade at high core counts converging to the performance of the other CC schemes from 56 cores (>1 socket).

For the other CC schemes, there are minor similarities of the individual throughput curves of the CC schemes between the many-core simulation and the multi-socket hardware. Focusing on the relative performance of the CC schemes other than *SILO* and *TICTOC*, reveals significant improvement of *OCC* and decrease of *MVCC*. Additionally, the pessimistic schemes converge at high core counts only degrading at different points and rates. Finally, *H-Store* still only performs well for small core counts (≤ 4) and remains slow beyond. Moreover, considering the time breakdown for the high conflict TPC-C workload in Figure 11a, we again observe textbook

behaviour as in the early time breakdown in Section 3.2 with fractions of *wait, backoff*, and *abort* characteristic for the individual CC schemes, though the amount of *useful* generally improves and *commit* as well as *CC Mgmt.* decrease through our optimisations.

Next, we analyse the low conflict workload using our optimised implementation. Figure 10b reveals that under this workload all CC schemes broadly provide scalable performance with fewer differences as the schemes show in the many-core simulation. That is, up to two sockets the throughput of all CC schemes steeply grows. Then the throughput continues to grow linear up to 1344 cores at a lower growth rate. At the full scale of 1568 cores, the behaviour of the CC schemes differs. *TICTOC*, *SILO*, and *MVCC* make a steep jump reaching 197 M txn/s, 159 M txn/s, and 75 M txn/s, respectively. Also, the growth rate of the pessimistic locking schemes increases but not as much yielding 34 M txn/s for *DL DETECT*, 32 tnx/s for *WAIT DIE*, and 36 M txn/s for *NO WAIT*. *OCC* stays linear achieving 39 M txn/s. In contrast, *HSTORE* degrades from 59 M txn/s at 1344 cores to 35 M txn/s at 1568 cores.

Now with this clear view, we can make out different characteristics of the CC schemes on the large multi-socket hardware, that are visible in their throughput as well as in their time breakdown as shown in Figure 11b. Under high conflict, the schemes *SILO* and *TICTOC* are the clear winners, although they do also not scale to high core counts (similar to the other schemes). Under low conflict, *HSTORE* performs the best before the number of concurrent transactions (cores) equals the number of partitions (warehouses). *HSTORE* degrades beyond this point, due to its simple but coarse partition locking, which is identical to the behaviour in the simulation. In detail, HSTORE's sensitivity to conflicts becomes obvious in the steep increase of *wait* time in the time breakdown.

Under low conflict, *TICTOC* follows as second fastest with *SILO* close by. Both provide significantly lower throughput than *HSTORE* until the tipping point at 1344 cores from which they outperform *HSTORE* by a large margin due to efficient fine-grained coordination, as indicated by their stable amount of *Commit* and *CC Mgmt.* For the other CC schemes, the view is diverse as their relation changes with the NUMA distance between the participating cores. After exceeding 8 sockets (448 cores/2 chassis) the pessimistic schemes fall behind the advanced optimistic CC schemes (*TICTOC & SILO*) and eventually also behind *OCC* and *MVCC*. This degrading is unrelated to conflicts (no *wait* time) but correlates with increasing NUMA distances. Consequently, for the low conflict OLTP workload, it appears that pessimistic locking is beneficial when access latencies (NUMA effects) are low, whereas the temporary copies of optimistic CC can hide these latencies, but these temporary copies come at the cost of additional data movement, slowing down throughput at close NUMA distance. To this end, *HSTORE* and *TICTOC* implement these two approaches as well, but they are more efficient, e.g., as *HSTORE* locks less frequently. Notably, there is no difference among the pessimistic CC schemes with different mechanisms against deadlocks, as there is low conflict in the workload, and thus few deadlocks.

Insight: After spending considerable engineering effort bringing state-of-the-art in-memory design to DBx1000, we shed new light on concurrency control on 1000 cores. First, we unveil remarkable peak throughput of the newer CC schemes, *TICTOC* and *SILO*, on high conflict workload, while also presenting textbook behaviour of all CC schemes in the time breakdown. Second, we brighten the grim forecast of concurrency control on 1000 cores for low conflict workload from the simulation of [27]. In fact, under low conflict all CC schemes scale nearly linearly to 1568 cores with a maximum of 200 million TPC-C transactions per second.

6 Discussion and Conclusion

In this paper, we analysed in-memory DBMS on today's large multi-socket hardware with 1568 cores, revisiting the results of the simulation in [27], which led us to several surprising findings: (1) A first attempt of running their prototype on today's multi-socket hardware presented broadly different behaviour of the CC schemes. To our surprise, the low contention TPC-C workload with at most one warehouse per thread revealed most concurrency schemes not only stopped scaling after 200 cores but also were very inefficient spending not even half of their time on useful work. (2) Based on these results, we decided to take a second deeper look into the underlying causes and made several additional discoveries. First, DBx1000 uses atomic increments to create timestamps in the default setting. This was a major bottleneck on the multi-socket hardware. Second, the default benchmark settings of DBx1000 used a TPC-C database which was significantly reduced in size and did not implement any inserts in the transactions. Changing these default setting shifted the picture of our initial assessment completely: while replacing the atomic counter with a hardware clock removed the timestamp creation bottleneck, enabling the original database size and insert operations, however, led to an even darker picture as in our first look. In this second look, we saw that all CC schemes completely collapsed when scaling to more than 200 cores, resulting in devastating 0.5 million txn/s when all cores were used. (3) Finally, we took this challenge and spent significant engineering efforts on the DBx1000 code base to optimise all components from memory management over transaction scheduling to locking. This cleared up the dark skies we faced before and allowed most CC schemes to perfectly scale, providing up to 200 million txn/s on 1568 cores. Even more surprisingly, now, the CC schemes behave very similar with no clear winner.

We speculate that this is due to the fact that most schemes are now memory-bound, emphasising the need to invest in latency hiding techniques such as interleaving with coroutines [15, 22] as an intriguing direction for future work on scalable concurrency control. Having cleared the view on concurrency with this re-evaluation on large hardware, fundamental optimisations like hardware-awareness of OLTP architecture [21] or even adaptive architectures [1] appear exciting for further evaluation of DBMS on such hardware. Also, now that hardware is available, evaluating not only broad concurrency but also utilisation of the full memory capacities is an interesting avenue towards hundred-thousands of TPC-C warehouses on in-memory DBMS.

So, stay tuned for "Part 2 on The Tale of 1000 Cores" :).

Acknowledgments

We would like to express our great gratitude to the authors of [27] for providing DBx1000 as open source making this work possible.

References

[1] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. 2020. Robust Performance of Main Memory Data Structures by Configuration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20),*. ACM, New York, NY, USA, 16. https://doi.org/10.1145/3318464.3389725

[2] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221.

[3] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483.

[4] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. 2016. Investigating the Performance of Hardware Transactions on a Multi-Socket Machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2935796, 121–132. https://doi.org/10.1145/2935764.2935796

[5] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 199–210. https://doi.org/10.1145/2150976.2150998

[6] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization But Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2522714, 33–48. https://doi.org/10.1145/2517349.2522714

[7] Ulrich Drepper. 2007. What Every Programmer Should Know About Memory. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[8] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*. Association for Computing Machinery, Article 8. https://doi.org/10.1145/3302424.3303977

[9] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. 2002. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *AUUG Conference Proceedings*, Vol. 85. AUUG, Inc. Kensington, NSW, Australia, 479 – 495.

[10] Hewlett Packard Enterprise. 2018. *The Unique Modular Architecture of HPE Superdome Flex: How it Works and Why It Matters.* https://community.hpe.com/t5/Servers-The-Right-Compute/The-unique-modular-architecture-of-HPE-Superdome-Flex-How-it/ba-p/7001330#.XnsMbEBFyAg.

[11] Hewlett Packard Enterprise Development LP. 2018. *HPE Superdome Flex, Intel Processors Scale SAP HANA.* https://www.intel.com/content/www/us/en/big-data/hpe-superdome-flex-sap-hana-wp.html.

[12] Hewlett Packard Enterprise Development LP. 2020. *HPE Superdome Flex Server Architecture and RAS.* https://assets.ext.hpe.com/is/content/hpedam/documents/a00036000-6999/a00036491/a00036491enw.pdf.

[13] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 629–642. https://doi.org/10.14778/3377369.3377373

[14] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.*

[15] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714. https://doi.org/10.14778/3236187.3236216

[16] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[17] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3284966

[18] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 691–706. https://doi.org/10.1145/2723372.2746480

[19] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.

[20] Tirthankar Lahiri and Markus Kissling. 2015. *Oracle's In-Memory Database Strategy for OLTP and Analytics.* https://www.doag.org/formes/pubfiles/7378967/2015-K-DB-Tirthankar_Lahiri-Oracle_s_In-Memory_Database_Strategy_for_Analytics_and_OLTP-Manuskript.pdf.

[21] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. 2016. Characterization of the Impact of Hardware Islands on OLTP. *The VLDB Journal* 25, 5 (2016), 625–650. https://doi.org/10.1007/s00778-015-0413-2

[22] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Interleaving with Coroutines: A Systematic and Practical Approach to Hide Memory Latency in Index Joins. *The VLDB Journal* 28, 4 (2019), 451–471. https://doi.org/10.1007/s00778-018-0533-6

[23] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *The Proceedings of the VLDB Endowment* 8, 12 (2015), 1442–1453. https://doi.org/10.14778/2824032.2824043

[24] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[25] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[26] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, Blazej Filipiak, and Sri Sakthivelu. 2020. Intel Memory Latency Checker v3.8. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[27] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[28] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935