DT-DB4MLKD-B
Moderne Datenbanksysteme für maschinelles Lernen und Wissensentdeckung
Silentium!
Run–Analyse–Eradicate the Noise out of the DB/OS Stack

University of Bamberg
Bamberg, Germany
johann-martin.hoefer@uni-bamberg.de

ABSTRACT

This is a rework of the original Silentium!-Paper for the DT-DB4MLKD-B Seminar. When multiple tenants compete for resources, database performance tends to suffer but sub-millisecond latencies are crucial. This paper studies how to make query latencies deterministic in the face of noise, showcases controlled experiments with with an in-memory database engine in a multi-tenant setting, where noisy interference from the system stoftware stack is successfully eradicated to the point of running close to bare-metal on the underlying hardware. It is shown that query latencies comparable to the database engine running as the sole tenant, but without noticeably impacting the workload of competing tenants is achievable. This paper discusses these results in context of building a custom OS for database workloads and point out the adequate quality of an existing and expertly configured OS.

Keywords: Low-latency DB; bounded-time query processing; DB-OS co-engineering; tail latency

1 INTRODUCTION

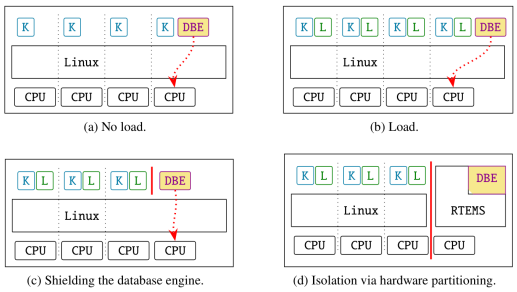
The OS is often regarded as a critical factor in scalable service stack development. While a general-purpose OS provides hardware support, drivers and system abstractions, they are a cause of jitter in network bandwidth, disk I/O and CPU [Ar09; SDQ10; Xu13].This also affects cloud-hosted database engine (DBE) performance [Ki15]. While historically, database and operating-systems research have been highly interwoven, the communities have parted ways in the past, and are just now rediscovering potential synergy effects (e.g. [Ca20; Mü20]). This has sparked immense interest in devising novel system architectures [KSL13].This however is connected to large efforts of creation and maintenance. So here we look at established OS solutions for ow-latency/high determinism workloads, as they arise in real-time scenarios and similarly in cloud settings where latency effects can lead to systemic problems [DB13]. Therefore the employment, root-cause identification, analysis and addressing of existing open-source components is logical. By vertical, cross-cutting engineering, the stack is tailored towards the needs of DBEs, eradicating interference and reducing noise-induced latencies in query evaluation. First results show purposeful employment of existing architectural measures effects jitter to a large degree. These results are obtained through controlled experiments with in-memory DBE running multi-tenant software stack scenarios with a focus on DBEs as a specific use case where deterministic latencies are essential [BL01].

2 OVERVIEW

2.1 Sources of Noise

In this section and beyond, by the term kernel we refer not to the database but to the operating systems kernel. The three main sources of noise, as observed by an unprivileged user space workload (as compared to system services or the kernel) are (1) competing processes and system services (2) non-disableable CPU performance optimisations such as caches or pipelines and (3) contention of implicitly shared resources (memory bus etc). Such *systemic* noise is often undistinguishable from *intrinsic* noise of the application (run-time variations through data-dependent code paths or application-specific optimisations etc.) **Processes and system services** Multi-tasking operating systems manage M scheduleable entities competing for N processors, with $M>N$. Linux uses a completely fair scheduling (CFS) [Ma10] policy for regular processes and supports round robin and FIFO (soft) real-time scheduling. Kernels can pre-empt most userland activities and place threads into the schedule that perform on behalf of the kernel (e.g. to support cross-CPU migration) and enjoy higher priority than regular processes whether they are governed by real-time policies or not. This factor interplay creates noise compared to uninterrupted, continuous flow of execution of a single job. *CPU Noise* Even given uninterrupted code execution, pipelined and superscalar execution may lead to temporal divergence to straightforward execution of assembly instructions. Caching mechanisms (foremost cache hierarchy interacting with memory references) cause varying latencies in accessing memory, effectively adding noise. *Shared Resources* Executing Workloads are not entirely isolated from another, but interact via shared resources, accessed via system buses. While deterministic from a *system-global view*, delays caused by competing requests manifest as noise viewed from the *individual process*.

2.2 Experimental System Configurations



(a) No load. (b) Load. (c) Shielding the database engine. (d) Isolation via hardware partitioning.

Figure 1: DBE Scenarios

System software stack configurations are visualized in *Figure 1*. In **No Load** a single DBE executes on an otherwise quiet system while the payload is pinned to one CPU to avoid perturbations. Standard system services and kernel threads required by the OS proper can execute on all CPUs, including the CPU dedicated to the database workload.

In **Load** additional tenants (through synthetic workloads) put the system at capacity while the database task can still be pre-empted by the kernel, or by incoming interrupts.

In **Shielding** the CPU distributes all existing tasks and kernel threads to the rest of the system and thus prevents use of the shielded CPU by any non explicitly assigned processes. Main memory, buses, cashes etc. remain shared resources though.The kernel can also still pre-empt the running userland task (for instance when timers expire) and then latencies can arise from administrative duties performed, or in system calls issued by the task.

The strongest form of isolation considered is **Partitioning** based on the Jailhouse hypervisor [Ra17]. Jailhouse can partition system hardware resources by establishing independent and strictly isolated computing domains through leveraging underlying system architecture, including essential virtualisation mechanism for system partitioning. It comes at a negligible performance overhead, as it does neither (para-)virtualise or emulate resources, nor schedule its partitions (guests) among CPUs. This architecture can find application in multi-tenant database scenarios, described in [MKN12]. For **Bare-Metal Operation**(s), which data centers, cloud and high performance data processing systems often employ due to them benefiting from bounded tail latencies, an ARM core and a x86 server class CPU are used for experimentation. This is to build a realistic database deployment that reduces systemic noise, stemming from multicore effects, to the bare minimum (long pipelines, large caches, and strong interference on buses) allowing for the exploration of intrinsic variations with the database workload. The in-memory database engine *DBToaster* [Ko14], a portable serverless DBE running on C++/STL with no other libraries or system services, can be deployed by moderate effort without having to rely on an OS proper. To allow the STL to run properly the DBE is ported to *RTEMS* (real-time executive for multiprocessor systems) [BS14]. To reduce operating system noise as far as possible, we essentially limit *RTEMS* to providing only a console driver, and execute the database engine in a single thread, which eliminates the need for a scheduler. This configuration is supposed to reduce any OS noise to the bare minimum, and is comparable to a bare-metal, main-loop style binary.

3 EXPERIMENTS

The experiments are conducted with the in-memory DBE *DBToaster*, which compiles SQL queries to C++, which in turn is compiled to the target platform. Resulting in a single-threaded DBE executable, incrementally updating a SQL view given a tuple stream, making it a low refresh latency SQL-to-code compiler. Typical application would include stream processing, such as algorithmic trading, network monitoring or clickstream analysis. To be able to discuss the run-time results properly the focus is on a subset of queries. In particular, queries that display high levels of intrinsic variability in latency, or variance in computational effort due to nested correlated sub-queries and multi-joins, are excluded, as they aren't suited for stream processing.

The queries considered are listed in *Figure 2*. The *Finance queries* are the incremental *countone*, designed as a baseline, the queries *axfinder*(AXF) and *pricespread*(PSP) which each compute a join, a selection, aggregation and for AXF also a group-by on the input stream. To execute these queries on hardware limited by memory, thee base data set of 100 tuples is iterated over 5000 times, thus yielding 500.000 data points. The *TPC-H queries* are generated with *dbgen* set to a scale factor 4. Queries *Q6, Q1* and *Q11a* from *Figure 2* are used. The queries perform selections, aggregations, and in the case of *Q11a* also a join.

CT
```
SELECT count (1) FROM bids;
```

axfinder
```
SELECT b.broker_id,
  SUM(a.volume*(-1*b.volume)) AS axfinder
FROM bids b, asks a
WHERE b.broker_id = a.broker_id
  AND ((a.price+((-1) * b.price)>1000)
  OR (b.price+((-1) * a.price)>1000))
GROUP BY b.broker_id;
```

TPCH Q11a
```
SELECT ps.partkey,
  SUM(ps.supplycost * ps.availqty)
  AS querylla
FROM  partsupp ps, supplier s
WHERE ps.suppkey = s.suppkey
GROUP BY ps.partkey;
```

TPCH Q1
```
SELECT returnflag, linestatus,
  SUM(quantity) AS sum_qty, SUM(extendedprice) AS sum_base_price,
  SUM(extendedprice*(1-discount)) AS sum_disc_price,
  SUM(extendedprice*(1-discount)*(1+tax)) AS sum_charge,
  AVG(quantity) AS avg_qty, AVG(extendedprice) AS avg_price,
  AVG(discount) AS avg_disc, COUNT(*) AS count_order
FROM lineitem
WHERE shipdate<=DATE('1997-09-01')
GROUP BY returnflag, linestatus;
```

pricespread
```
SELECT SUM(a.price + (-1*b.price))
          AS psp
FROM bids b, asks a
WHERE (b.volume > 0.0001 *
  (SELECT SUM(b1.volume) FROM bids b1))
  AND (a.volume > 0.0001 *
  (SELECT SUM(a1.volume) FROM asks a1));
```

TPCH Q6
```
SELECT SUM(l.extendedprice*l.discount)
          AS revenue
FROM lineitem l
WHERE l.shipdate>=DATE('1994-01-01')
  AND l.shipdate<DATE('1995-01-01')
  AND ( l.discount BETWEEN (0.06-0.01)
          AND (0.06+0.01) )
  AND l.quantity<24;
```

Figure 2: SQL queries used in the experiments

The execution platform for the experiments for the x86 architecture is a *Dell PowerEdge T440* with a single *12 core Intel®™ Xeon®™ Gold 5118 CPUs* and *32 GiB* of main memory. For Linux, kernel version 5.4.38 as baseline. *Symmetric multithreading* (SMT) is deactivated on the target, as is *Intel®™ Turbo Boost®™*. The CPU is configured to the highest possible *P-State* (performance setting) that guarantees a stable core frequency of 2.29 GHz. For the *shielding* scenario CPU namespaces are used that can be dynamically reconfigured during system operation. For the *Jailhouse* setup a single CPU and *1GiB* of main memory from Linux is released and assigned to a new computational domain. There *RTEMS + DBToaster binary* runs parallel to Linux. To partition last-level caches an exclusive *5 MiB* of *Level 3 Cache* to the *RTEMS + DBToaster* domain are assigned. This mitigates noise (cache pollution) of neighboured CPUs, as the *Level 3 Cache* is shared across all cores [In15] For the *ARM* reference platform a *BeagleBone Black* with single-core *Sitara AM3358, 32 bit ARM Cortex-A8* processor and *512 MiB* main memory are used.

DBToaster logs a time-stamp for every N input tuples processed. Allowing for the computation of the *latency per N input tuples processed* averaged over N tuples. Two time measurements are used. First time stamps are obtained by *POSIX API*, allowing for nanosecond resolution, but creating considerable overhead in the microsecond range. Second, through *DBToaster* a x86 *time stamp counter* (TSC) is used to count ticks. All tuples are pre-load prior to stream processing to exclude noise caused by I/O. To simulate tenant load 6 synthetic workloads are used via *stress-ng*.

4 EXPERIMENT RESULTS

4.1 Finance Queries

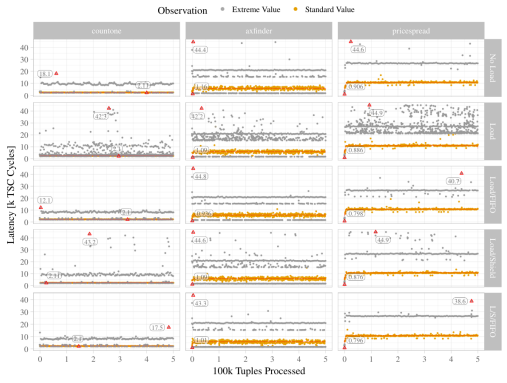


Figure 3: Noise and Determinism in Finance Query Latencies

Visualized in *Figure 3* are the observed latencies for processing each out of the *500.000* input tuples. Red triangles mark minima and maxima for each scenario. All measure points deemed *"extreme"*, which means exceeding the *99.95-percentile* or falling below the *0.05-percentile* are coloured grey. All other data points are considered normal and coloured ochre. And as a consistency check a red line is computed as a sliding mean window over *1.000 tuples*. While for performance measurements outliers have no noticeable influence, they are of paramount importance for real-time, bounded latency scenarios. Almost all latencies are centred around the sliding mean value. Important to note is, that a few outliers exceed the mean by a factor of about *four*. Maximum observed latency is essential for application and cannot be compensated by the fact, that this does not happen on average.

The average performance of the simulated other tenants is essentially identical regardless of measurement setup. This shows improved determinism for a given workload does not necessarily decrease average throughput for non-real-time loads. While there is no direct relation to query complexity and noise, there is a relation between query complexity and average performance. This is visible in *Figure 3* by comparing the *coountone* to the *axfinder* and the *pricespread* mean average. As the complexity rises from left to right, so do the latencies. The horizontal bands observable in *Figure 3*s latencies can be attributed to *DBToaster*s main execution paths. Two main execution paths emerge as the consequence of the *orderbook adapter*, which distinguishes between the two types of input data, *bids* and *asks*. Also when the internal dynamic data structures of *DBToaster* grow in size, additional *DBToaster*-intrinsic latencies incur. Vertical spreading around these bands is a visual noise measure.

By comparing against the "Load" scenario, it is apparent that the isolation mechanisms substantially reduce the observed jitter typically to the level of an otherwise unloaded system. The strongest isolation, *shielding+FIFO*, even beats the *No Load* scenario in terms of maximum values. The amount of noise decreases in order *Load/Shield*, *Load/FIFO*, and *Load/Shield/FIFO*. While the measurements show a noticeable reduction of noise when using more advanced isolation techniques, the reduction of maximal latencies comprises only a factor of two.

4.2 Noise and Determinism in TPC-H Queries

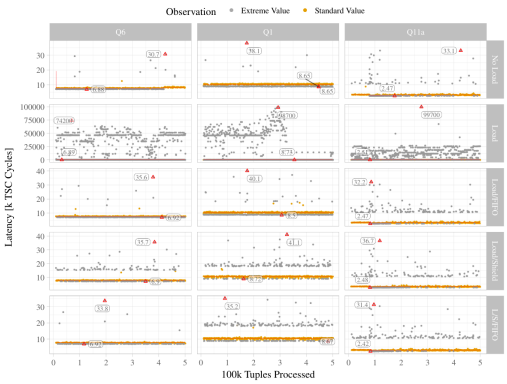


Figure 4: TPC-H Queriy Latencies

TPC-H query latency measurement observations are identical to *Finance Queries*, the behaviour of the queries under high load differs considerably from the *No Load* and isolated case. The difference in maximal latencies comprises more than three decimal orders of magnitude. While such high variance has grave consequences for real-time systems, it is not even observable when throughput measurements are averaged.

4.3 The Role of CPU Noise

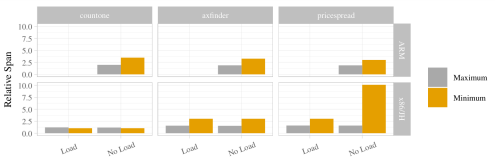


Figure 5: Span of observations relative to median query latency of finance queries on bare-metal, on a high-end (x86) and low-end (ARM) CPU

To a major extent, the previous experiments concern the control of noise introduced by the operating system and the presence of other tasks that compete for CPU time and other shared resources. Especially in the scenario using CPU isolation and combined with

a real-time scheduling policy it eliminates a substantial fraction of this noise. It is prudent to ask how much of the remaining noise is caused by the executing CPU itself, and can thus be seen as an effective lower bound on any systemic noise.

To achieve this, a run as close to *bare-metal* as possible is needed, to reduce the OS overhead. These measurements are performed on an *ARM processor*, deemed powerful enough to execute reasonable database operations, but that uses substantially fewer performance optimisations than *x86 server-class CPU*s (and, thus, suffers from less intrinsic noise). The choice for an ARM CPU is not just driven by simplicity, though: Processors of this type are the most frequent choice in embedded systems and IoT devices, where low latency data processing is a common requirement (for instance sensor-based systems that derive action decisions by combining previously measured values stored in a database with current data points). The measurements are therefore representative for this large class of systems that is expected to gain even more importance in future applications. Of course, measurements on CPUs with drastically different capabilities cannot be directly compared. Instead, it is important to consider the relative difference between average and maximal latencies, and the span within measurements.

The summary for a second set of measurements shown in the bottom part of *Fig5* represents bare-metal results obtained on the *x86 CPU*, but this time driven by an *RTEMS kernel* running inside a *Jailhouse* cell. Since the system is equipped with a total of *12 core*s (compared to the *single-core ARM*), and only one of the cores is needed to run the database workload, the measurement is extended with an additional aspect that quantifies the aptitude of the setup to decouple latency-critical database operations performed by one tenant from other, perhaps throughput-oriented operations performed by other tenants. The spread is, as *Figure 5* shows, almost identical between the scenarios.

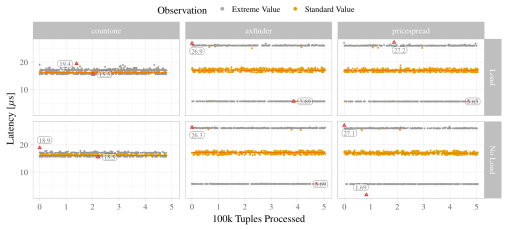


Figure 6: Latency time series for finance queries on an RTEMS-based near bare-metal CPU provided by the Jailhouse hypervisor

This is also reflected in the time series shown in *Figure 6*, which demonstrates that the results of the two configurations do not deviate in any substantial way. Since the isolation provided by Jailhouse does not only address execution timing, but also extends to other security and privacy related aspects of database workload processing, we deem this configuration a suitable basis for multi-tenant systems with strong separation guarantees.

5 CONSEQUENCES & CONCLUSION

The experimental results show, that building a database stack on *vanilla* Linux with custom settings can achieve competitiveness close to *bare-metal*. The main source of noise in the experiments was due to interruptions to measure time and noise itself.

This suggests, that it may be unnecessary design a DB-aware OS from scratch. Extending and enhancing existing systems may be more pragmatic. Research on novel OS for DBE is remains valid, but the study for actual reasons behind noise observed with existing OS is important. Only through the identification and understanding of root causes can solutions be found. In other fields of study, such as embedded real-time systems similar problems to the multi-tenant DBE use are researched and may give insight needed and promising solutions (almost) ready to deploy. It is important to note, that only CPU noise was eradicated in this experiment and I/O noise was deliberately ignored. Disk-based DBE will probably require more invasive changes to the existing software stack as well as extended support for disks and their management.

Through proper use of standard mechanisms of a OS proper, database query latencies comparable to running an in-memory DBE on raw hardware are achievable, but the point where measuring time becomes the largest source of noise is already at hand. By leveraging techniques established for mixed-criticality systems applied to the database domain e.g. shielding one workload (the DBE) without impairing the other workloads the challenges ahead, such as in-memory DBE disk support extension, might be tackled.

REFERENCES

[0000] Mauerer W.; Ramsauer R.; Edson R.; Lucas F.; Lohmann D.; Scherzinger S.: Silentium! Run–Analyse–Eradicate the Noise out of the DB/OS Stack, K.-U. Sattler et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2021), Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 397-421

[Ar09] Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing, tech. rep., University of California at Berkeley, 2009.

[BL01] Buchmann, A. P.; Liebig, C.: Distributed, Object-Oriented, Active, Real-Time DBMSS: We Want It All - Do We Need Them (at) All? In. Annual Reviews in Control, pp. 147–155, 2001.

[BS14] Bloom, G.; Sherrill, J.: Scheduling and Thread Management with RTEMS. SIGBED Rev. 11/1, pp. 20–25, 2014.

[Ca20] Cafarella, M. J.; DeWitt, D. J.; Gadepally, V.; Kepner, J.; Kozyrakis, C.; Kraska, T.; Stonebraker, M.; Zaharia, M.: DBOS: A Proposal for a Data- Centric Operating System. CoRR abs/2007.11112/, 2020.

[DB13] Dean, J.; Barroso, L. A.: The Tail at Scale. Commun. ACM 56/2, pp. 74–80, 2013.

[In15] Intel Corporation: Improving Real-Time Performance by Utilizing Cache Allocation Technology,

https://www.intel.com/content/dam/www/public/ us/en/documents/white-papers/cache-allocation-technology-white- paper.pdf (last accessed February 2021), 2015.

[Ki15] Kiefer, T.; Schön, H.; Habich, D.; Lehner, W.: A Query, a Minute: Evaluating Performance Isolation in Cloud Databases. In: Performance Characterization and Benchmarking. Traditional to Big Data, pp. 173–187, 2015.

[Ko14] Koch, C.; Ahmad, Y.; Kennedy, O.; Nikolic, M.; Nötzli, A.; Lupei, D.; Shaikhha, A.: DBToaster: Higher-Order Delta Processing for Dynamic, Fre- quently Fresh Views. VLDB J. 23/2, pp. 253–278, 2014.

[KSL13] Kiefer, T.; Schlegel, B.; Lehner, W.: Experimental Evaluation of NUMA Effects on Database Management Systems. In: Datenbanksysteme für Business, Technologie und Web (BTW) 2025. BTW'13, pp. 185–204, 2013.

[Ma10] Mauerer, W.: Professional Linux Kernel Architecture. John Wiley and Sons, 2010.

[MKN12] Mühe, H.; Kemper, A.; Neumann, T.: The Mainframe Strikes Back: Elastic Multi-Tenancy Using Main Memory Database Systems On a Many-Core Server. In: Proceedings of the 15th International Conference on Extending Database Technology. EDBT'12, pp. 578–581, 2012.

[Mü20] Mühlig, J.; Müller, M.; Spincyk, O.; Teubner, J.: mxkernel: A Novel System Software Stack for Data Processing on Modern Hardware. Datenbank-Spektrum 20/3, pp. 223–230, 2020.

[Ra17] Ramsauer, R.; Kiszka, J.; Lohmann, D.; Mauerer, W.: Look Mum, no VM Exits! (Almost). In: Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '17). 2017.

[SDQ10] Schad, J.; Dittrich, J.; Quiané-Ruiz, J.-A.: Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. PVLDB Endow. 3/1, pp. 460– 471, 2010.

[Xu13] Xu, Y.; Musgrave, Z.; Noble, B.; Bailey, M.: Bobtail: Avoiding Long Tails in the Cloud. In: 10th USENIX Symposium on Networked Systems Design and Implementation. NSDI'13, pp. 329–341, 2013.