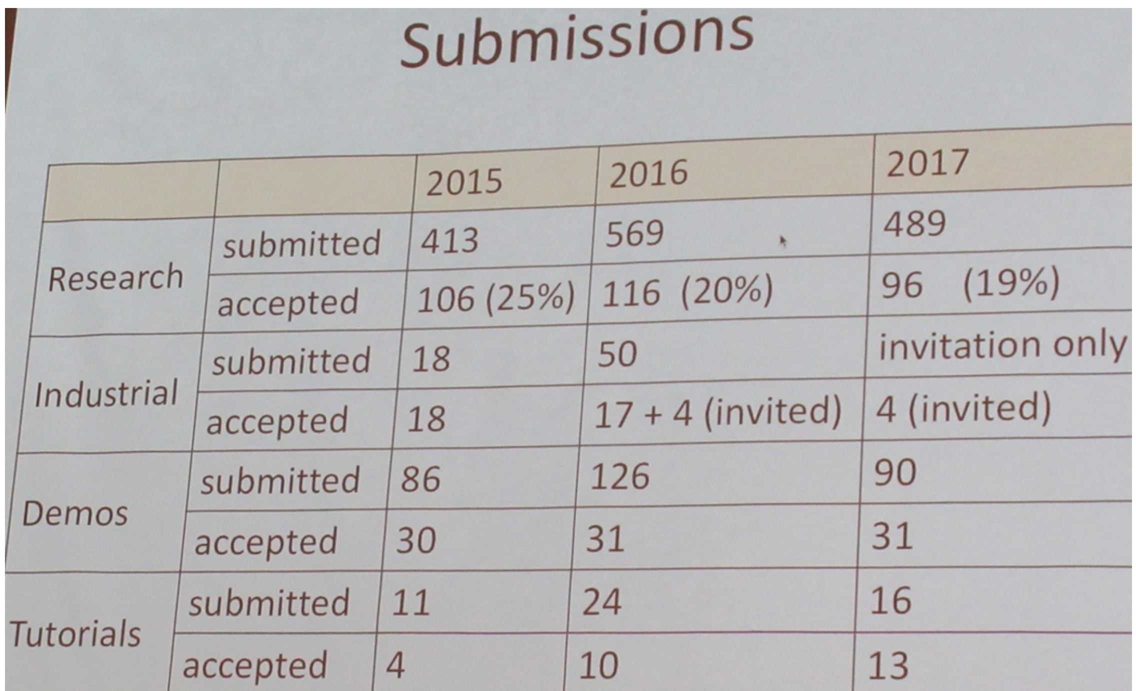
A Top-Down Approach to Achieving Performance Predictability in Database Systems

Harunobu Daikoku

HPCS Lab., University of Tsukuba

SIGMOD 2017 - Conference Overview

- Date: 5/14 (Sun) ~ 5/19 (Fri)
- Venue: Hilton Chicago, IL, USA
- # attendees: approx. 800



Submissions

		2015	2016	2017
Research	submitted	413	569	489
	accepted	106 (25%)	116 (20%)	96 (19%)
Industrial	submitted	18	50	invitation only
	accepted	18	17 + 4 (invited)	4 (invited)
Demos	submitted	86	126	90
	accepted	30	31	31
Tutorials	submitted	11	24	16
	accepted	4	10	13

SIGMOD 2017 @ Chicago, IL

A Top-Down Approach to Achieving Performance Predictability in Database Systems

Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, Thomas F. Wenisch

University of Michigan

* Some of the figures in this document are taken from the original literature.

tl;dr

- Stop focusing only on **raw performances** (e.g. throughput, mean latency).
- Should be looking at **performance predictability** as well.
- **TProfiler**: a performance tracing tool that identifies sources of latency variance in DBMSs.
- Successfully identifies and mitigates major sources of performance unpredictability in MySQL, PostgreSQL and VoltDB.

Performance Predictability

- **Predictability: Variance**
- Why so important?
  - DB-backed web services (latency directly affects **user experience**)
  - Service-Level Agreements (“*if violated, …result in **financial penalties***”)
- How bad is it?

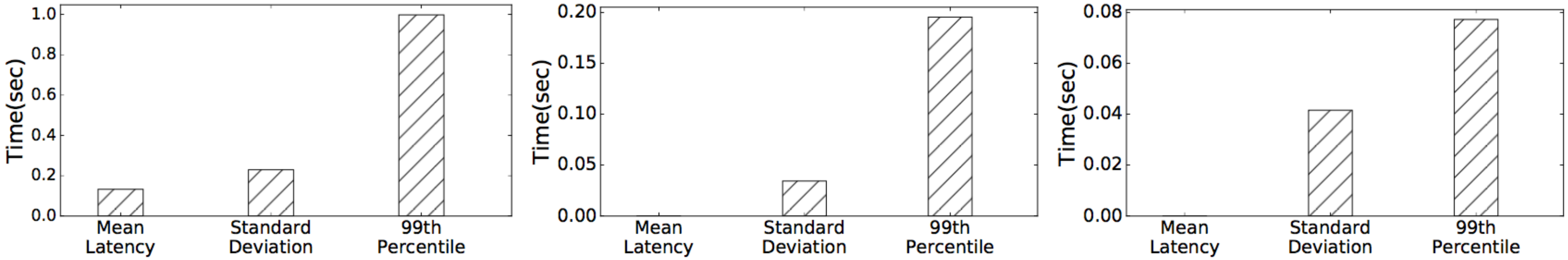


Figure 6: Mean, standard deviation, and 99th percentile latencies in MySQL (left), Postgres (center), and VoltDB (right).

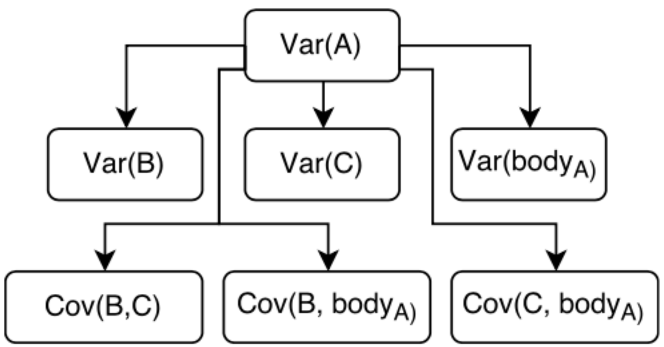
Sources of Unpredictability

- **Avoidable** (internal): caused by internal components of DBMSs (e.g. I/Os, contention, data structures, algorithms)

- Inherent (external): caused by varying amounts of work (e.g. "*a transaction that updates 10 tables inherently involves more work than one that updates only one table*")

TProfiler (VProfiler) - Overview

- Given the source codes of a DBMS (w/ explicit annotations of txns.), identifies sources of latency variance by generating a call graph called “***a variance tree***”



- Open-sourced: https://web.eecs.umich.edu/vprofiler/
  (VProfiler: a generalized version of TProfiler presented @ Eurosys 2017)

TProfiler - Differentiation

- Existing tools (e.g. DTrace [37]) are ignorant of…
  - Transaction-related code sequences inside the codebase
  - Mathematical nature of variance - *"the variance of a parent function is always strictly greater than the variance of its children…"*

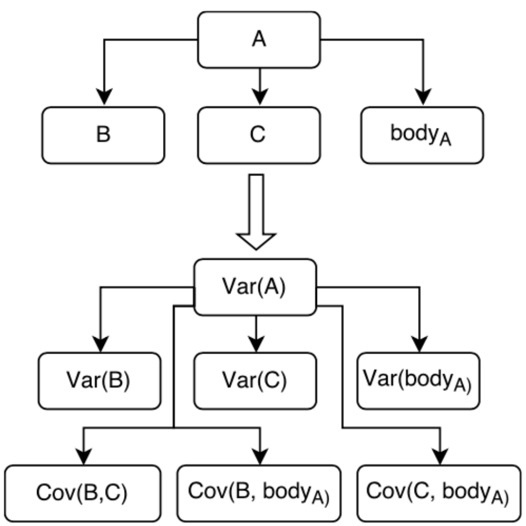
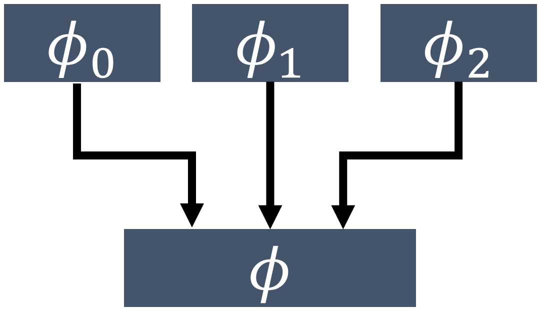


Figure 1: A call graph and its corresponding *variance tree* (here, $body_A$ represents the time spent in the body of A).

$$Var(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} Var(X_i) + 2 \sum_{1 \leq i} \sum_{\leq j \leq n} Cov(X_i, X_j)$$

TProfiler- Scoring Function

- **Considers both variance and depth within the call hierarchy**
- Intuition: "*functions deeper in the call graph implement more specific functionality*", thus are more informative

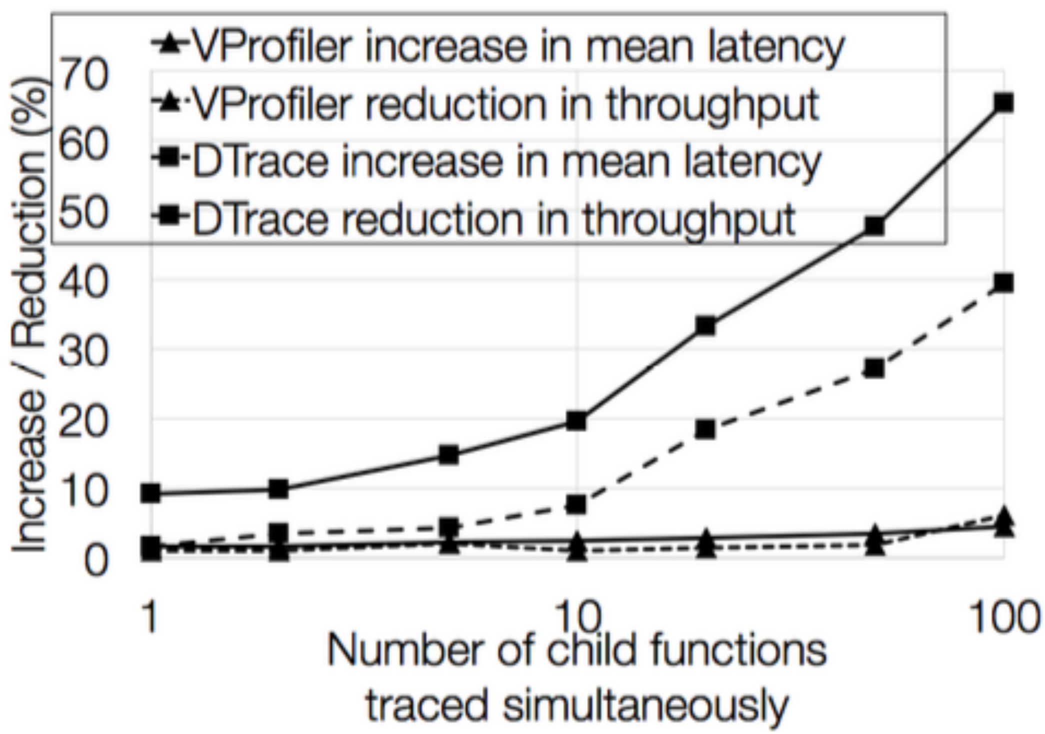


$$specificity(\phi) = (height(call\_graph) - height(\phi))^2$$

$$score(\phi) = specificity(\phi) \sum_i V(\phi_i)$$

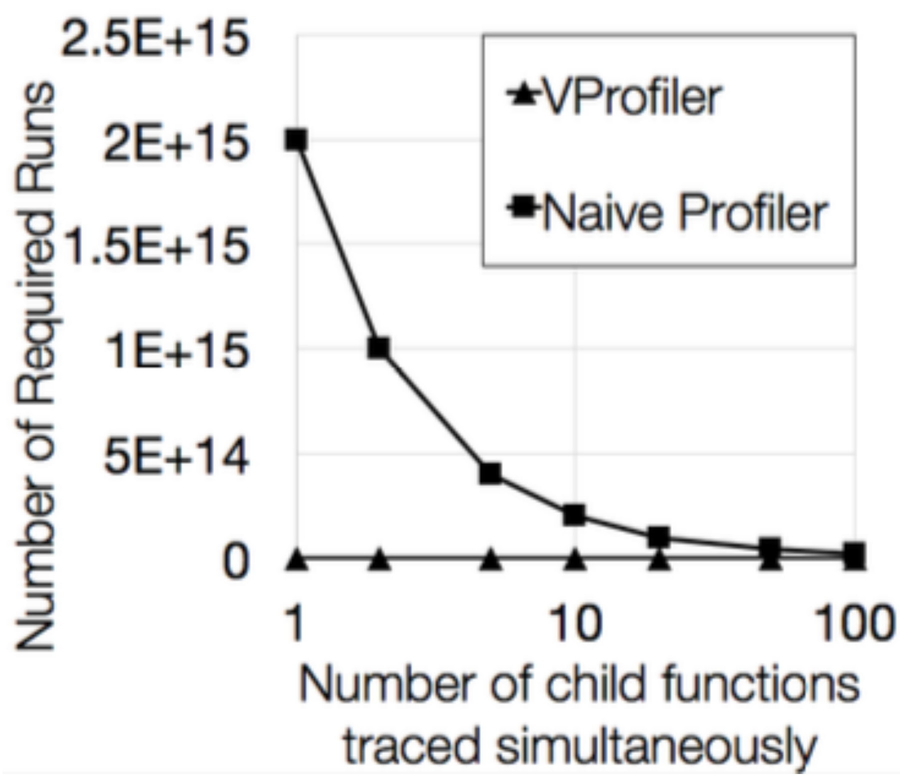
TProfiler vs DTrace [37]

- DTrace: instruments **the binary code** rather than the source code, "*use heavy-weight mechanisms to inject generalized instrumentation code at run-time*"



TProfiler vs Naïve Profiler

- Naïve Profiler: decomposes every single non-leaf functions in a call graph rather than a few important ones.



Case Studies

- Workload: TPC-C
- Analyzed 3 popular open-source DBMSs
  - MySQL 5.6.23 (a thread-per-connection model)
    - **128 WHs** w/ **30 GB** buffer pool (high contention on records)
    - **2 WHs** w/ **128 MB** buffer pool (high contention on the buffer pool)
  - PostgreSQL 9.6 (a process-per-connection model)
    - **32 WHs** w/ **30 GB** buffer pool
  - VoltDB (an event-based server model)

Case Studies – MySQL (128 WHs)

- **`os_event_wait()`**: used to put a thread to sleep when it requested a lock on a record that cannot be granted due to a conflict ([A]: SELECT statements, [B]: UPDATE statements) -> **AVOIDABLE**
- **`row_ins_clust_index_entry_low()`**: inserts a new record into a clustered index, takes varying code paths based on the state of the index -> **INHERENT**

Config	Function Name	Percentage of Overall Variance
128-WH	`os_event_wait` [A]	37.5%
128-WH	`os_event_wait` [B]	21.7%
128-WH	`row_ins_clust_index_entry_low`	9.3%

Case Studies – MySQL (2 WHs)

- **`buf_pool_mutex_enter`**: acquires the lock of the LRU list that manages buffer pages -> **AVOIDABLE**
- **`btr_cur_search_to_nth_level`**: traverses an index tree, varies with the depth -> **INHERENT**
- **`fil_flush()`**: flush redo logs (WAL) -> **INHERENT** (can be mitigated with faster I/O devices)

2-WH	`buf_pool_mutex_enter`	32.92%
2-WH	`img_btr_cur_search_to_nth_level`	8.3%
2-WH	`fil_flush`	5%

Case Studies - PostgreSQL

- **`LWLockAcquireOrWait()`**: acquires a single global lock (WALWriteLock) to ensure that only one txn. is flushing at a time -> **AVOIDABLE** (I/O acceleration or parallel logging)
- **`ReleasePredicateLocks()`**: releases predicate locks (for avoiding phantom problems) -> **INHERENT** (negligible)

Function Name	Percentage of Overall Variance
`LWLockAcquireOrWait`	76.8%
`ReleasePredicateLocks`	6%

Case Studies - VoltDB

- VoltDB: an event-based system
- Each event waits in a queue before a worker thread is assigned
- **99.9%** of latency variance comes from the varying waiting time of the event queues -> **AVOIDABLE**

  (adjust # worker threads and control the queue size)

Mitigation Ideas

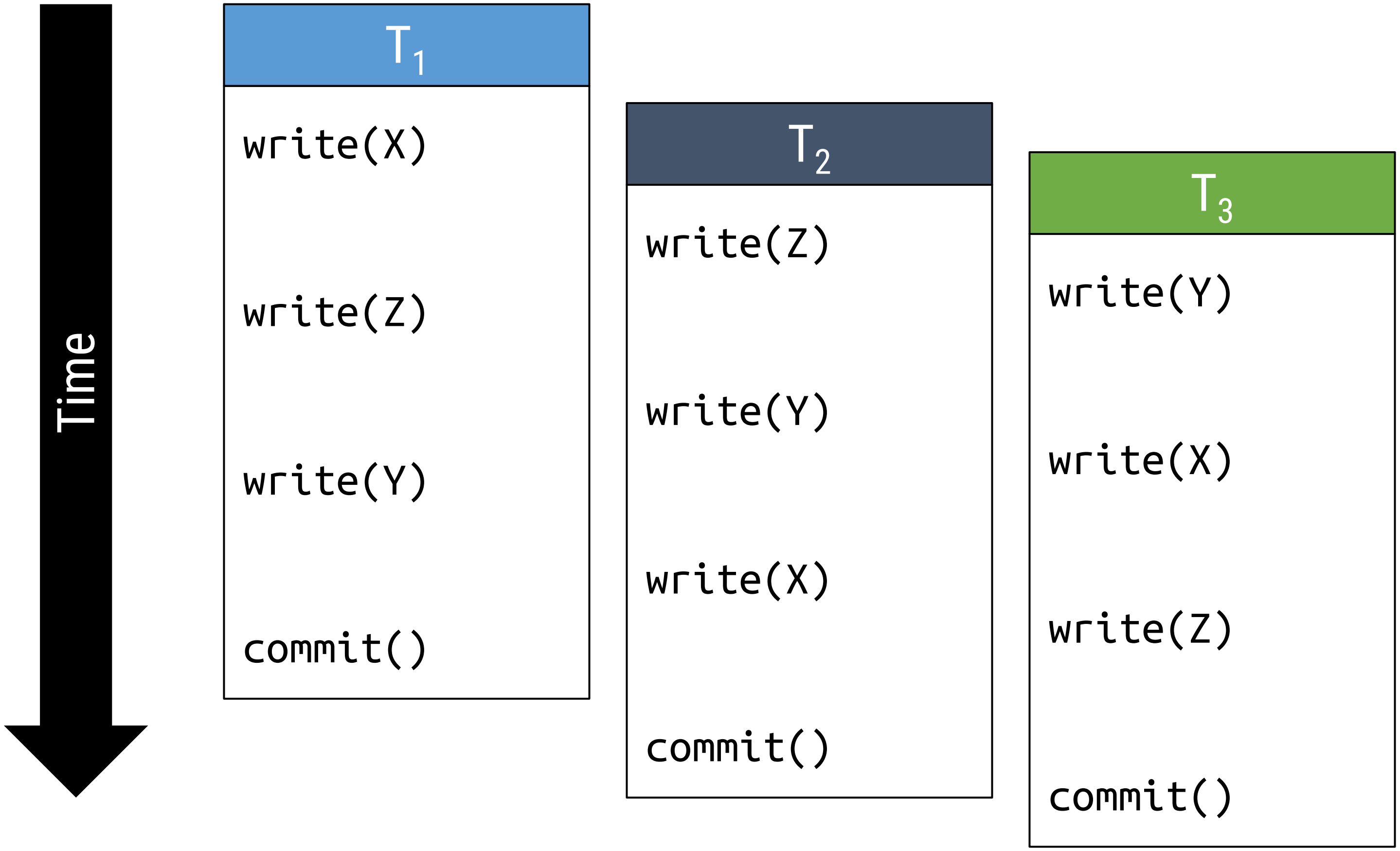
- MySQL
  - `os_event_wait` -> **schedule txns. in a variance-aware manner (VATS)**
  - `buf_pool_mutex_enter` -> **update LRU list lazily (LLU)**
- PostgreSQL
  - `LWLockAcquireOrWait` -> **parallelize WAL**
- VoltDB
  - Event queuing time -> **adjust # worker threads**

Mitigation Ideas

- MySQL
  - `os_event_wait` -> **schedule txns. in a variance-aware manner (VATS)**
  - `buf_pool_mutex_enter` -> **update LRU list lazily (LLU)**
- PostgreSQL
  - `LWLockAcquireOrWait` -> **parallelize WAL**
- VoltDB
  - Event queuing time -> **adjust # worker threads**

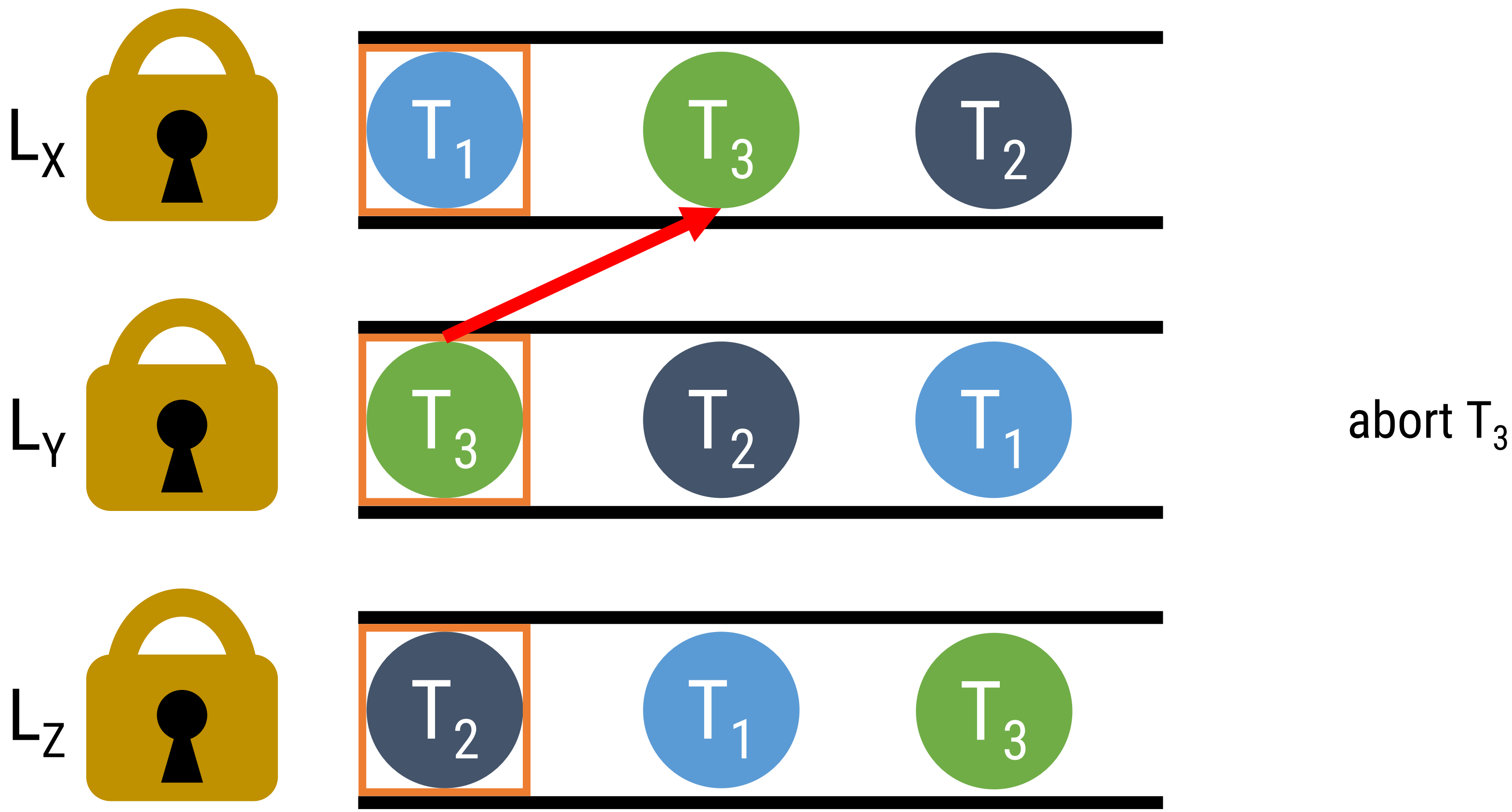
VATS vs FCFS - Example

- Protocol: Strict 2-Phase Locking + Wait-Die Deadlock Prevention



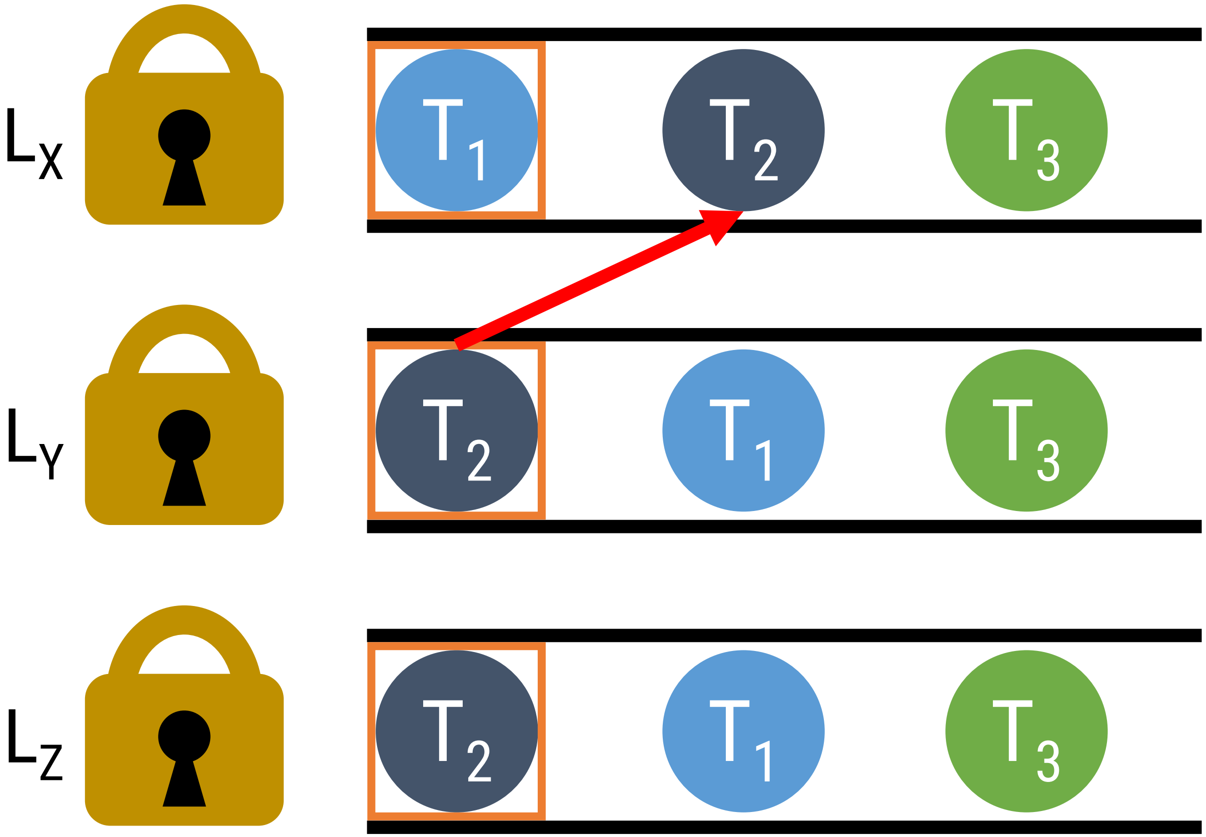
VATS vs FCFS – FCFS (1/5)

FCFS (First-Come-First-Served): Grants locks to the txns. at the head



VATS vs FCFS – FCFS (2/5)

FCFS (First-Come-First-Served): Grants locks to the txns. at the head



abort T_2

VATS vs FCFS – FCFS (3/5)

FCFS (First-Come-First-Served): Grants locks to the txns. at the head

commit T_1

VATS vs FCFS – FCFS (4/5)

FCFS (First-Come-First-Served): Grants locks to the txns. at the head

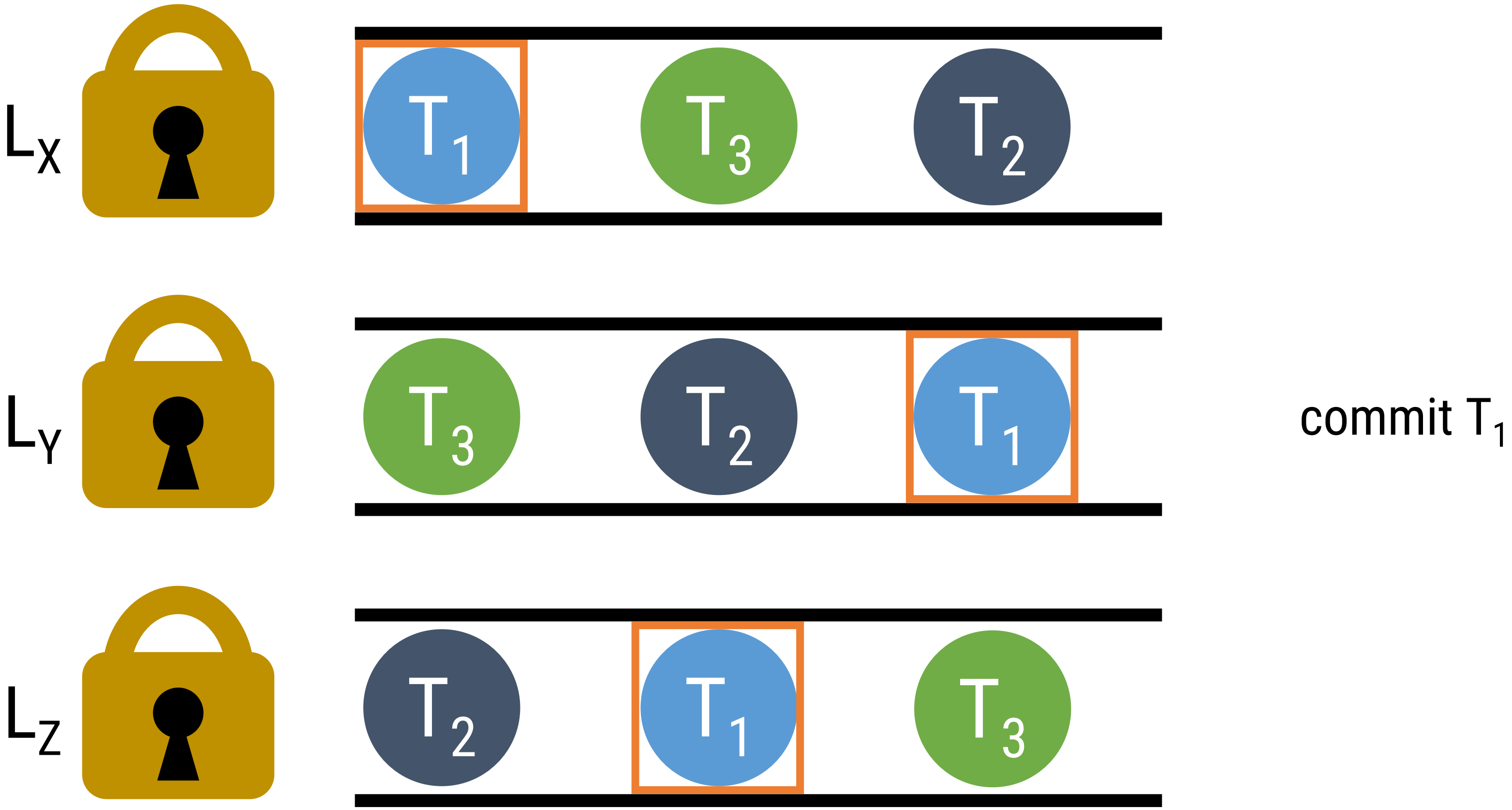
commit T_3

VATS vs FCFS – FCFS (5/5)

FCFS (First-Come-First-Served): Grants locks to the txns. at the head

VATS vs FCFS – VATS (1/3)

VATS: Grants lock to the eldest txns.



VATS vs FCFS – VATS (2/3)

VATS: Grants lock to the eldest txns.

commit T_2

VATS vs FCFS – VATS (3/3)

VATS: Grants lock to the eldest txns.

commit T_3

VATS

- Loss function
  - Variance: not suited (adding a large delay to every txn. can achieve a near-zero variance, but significantly increase mean latency)
  - **Lp norm: indirectly reduce both mean and variance latencies**

    (l_i: latency of txn. i,   p: 2 in practice)

$$L_p = ||\langle l_1, \cdots, l_n \rangle||_p = (\sum_{i=1}^{n} |l_i|^p)^{1/p}$$

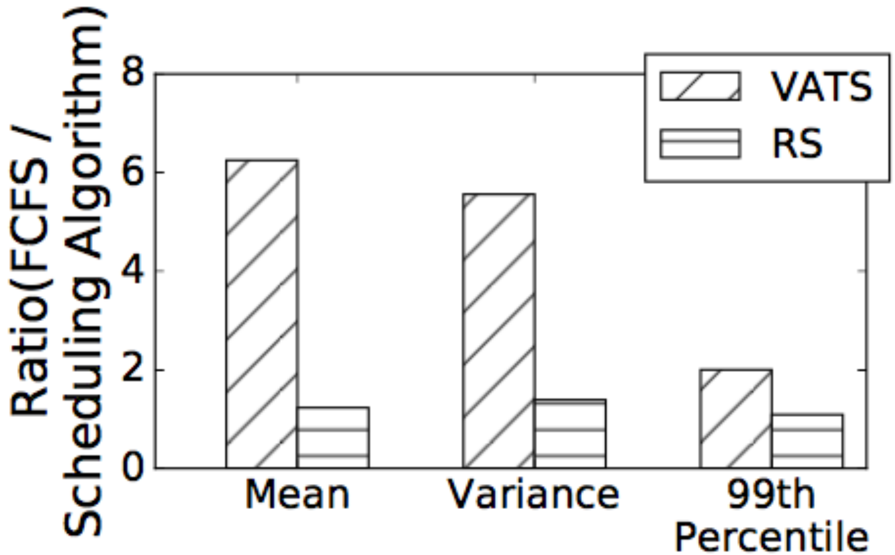
- **Lp norm of VATS scheduler is optimal against all schedulers**

  (Theorem 1, proof in Section 5.3)

VATS – Experiment (1/2)

- Workload: TPC-C

RS: Randomized Scheduling



System	Name of the Identified Function	Original contribution to overall variance	Modification	Modified lines of code or config	Ratio of overall latency variances (Orig. / Modified)	Ratio of overall 99^{th} latencies (Orig. / Modif.)	Ratio of overall mean latencies (Orig. / Modif.)
MySQL	`os_event_wait`	59.2%	replace FCFS with VATS	189	5.6x	2.0x	6.3x
MySQL	`buf_pool_mutex_enter`	32.92%	replace mutex with spin lock	46	1.6x	1.4x	1.1x
MySQL	`fil_flush`	5%	parameter tuning	2	1.4x	1.2x	1.2x
Postgres	`LWLockAcquireOrWait`	76.8%	parallel logging	355	1.8x	1.3x	2.4x
VoltDB	`[waiting in queue]`	99.9%	add # of worker threads	1	2.6x	1.4x	5.7x

VATS – Experiment (2/2)

- SEATS [62]: airline ticketing system (highly contended)
- TATP [68]: caller location system (“*not as contended as TPC-C*”)
- Epinions [48]: customer reviewing system
- YCSB [30]: no lock contentions

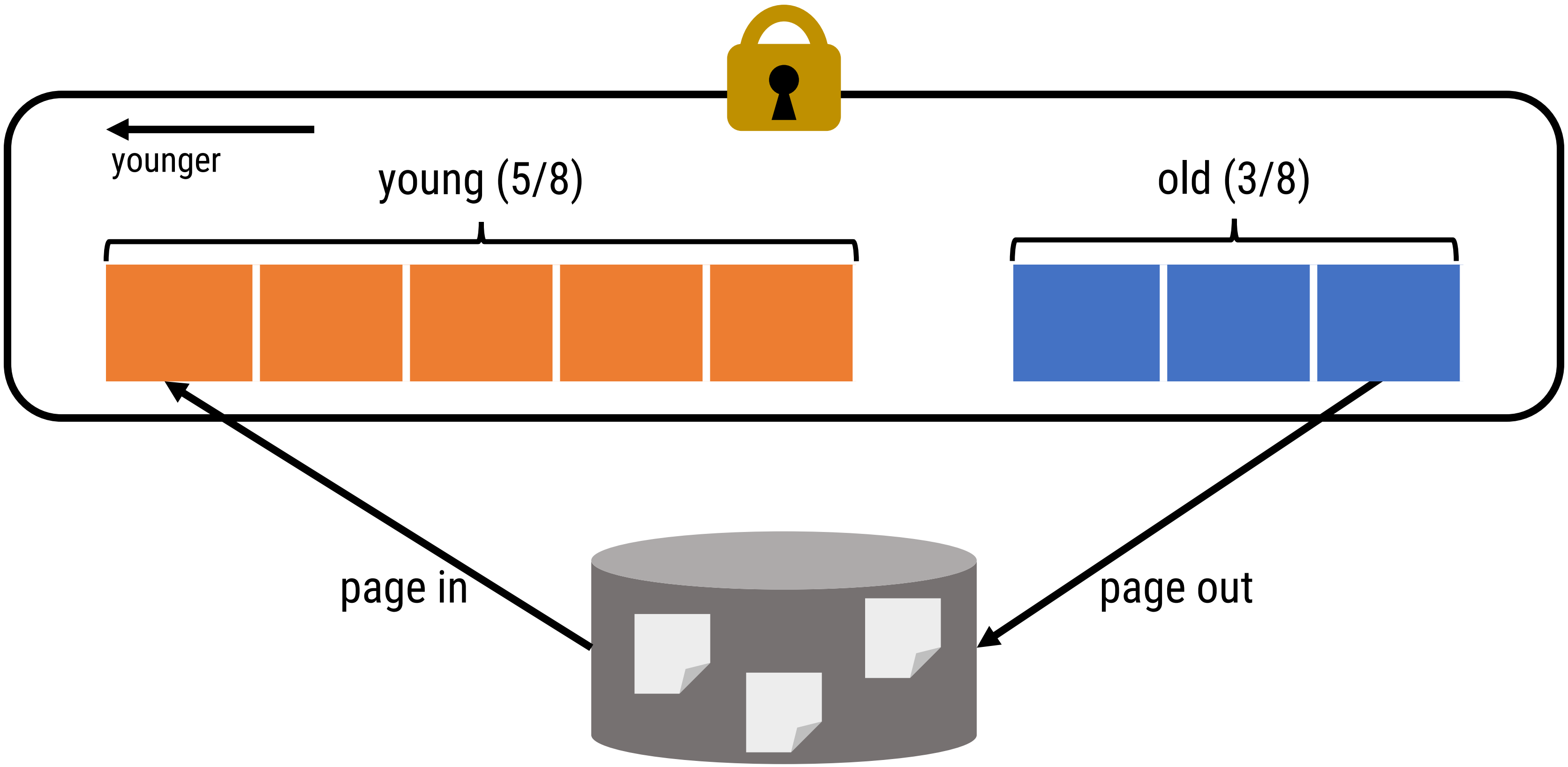
	Workload	Mean Latency	Variance	99th Percentile
Contended	TPCC	6.3x	5.6x	2.0x
	SEATS	1.1x	1.3x	1.1x
	TATP	1.2x	1.6x	1.3x
	Avg	**2.9x**	**2.8x**	**1.5x**
No Contention	Epinions	1.4x	2.6x	1.0x
	YCSB	1.0x	1.1x	1.1x

Mitigation Ideas

- MySQL
  - `os_event_wait` -> **schedule txns. in a variance-aware manner (VATS)**
  - `buf_pool_mutex_enter` -> **update LRU list lazily (LLU)**
- PostgreSQL
  - `LWLockAcquireOrWait` -> **parallelize WAL**
- VoltDB
  - Event queuing time -> **adjust # worker threads**

(Relaxed) LRU Buffer Pool in MySQL

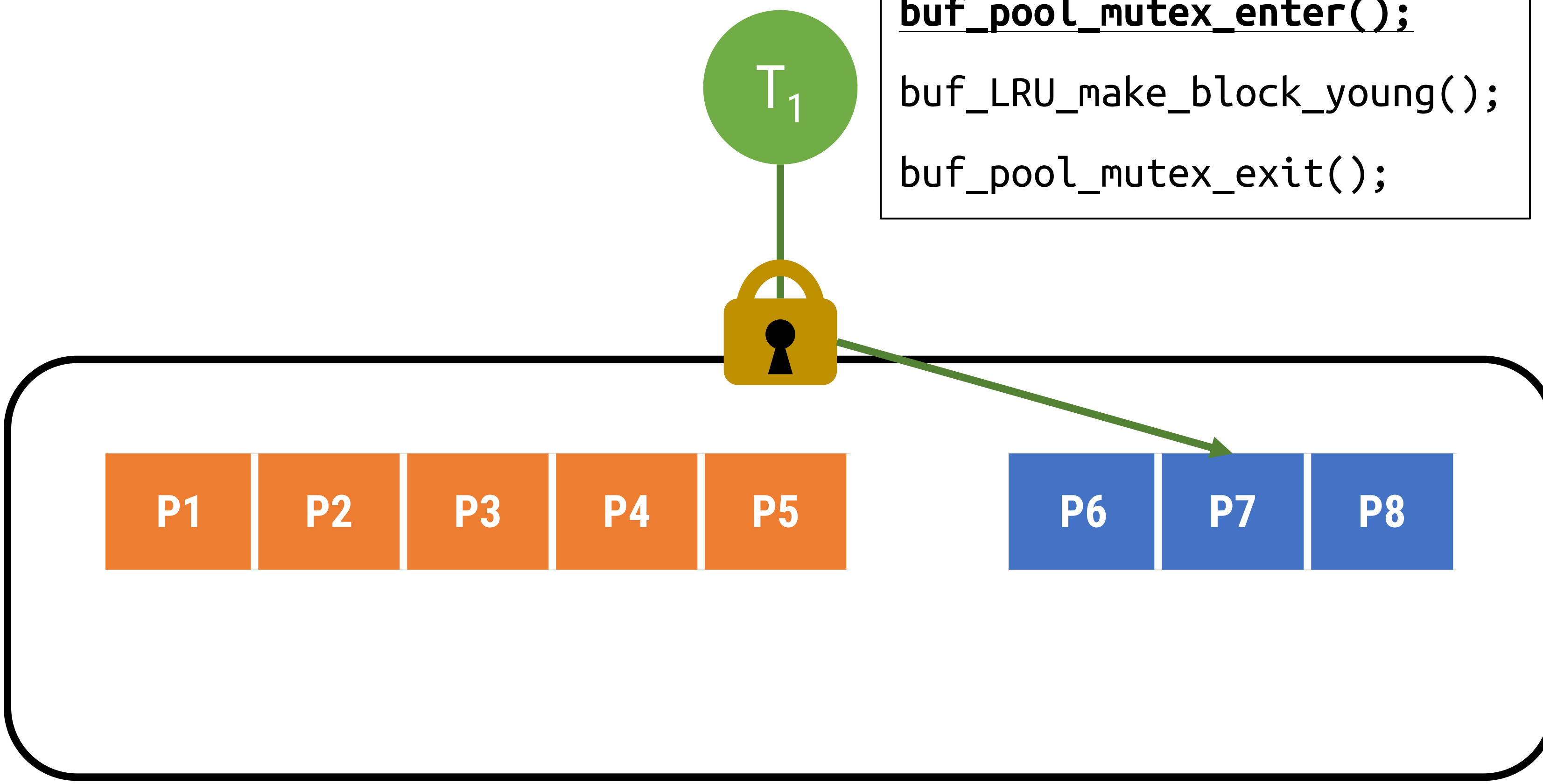
- Consists of two sub-lists: young & old



(Relaxed) LRU Buffer Pool in MySQL

- No precise LRU ordering within the “young” sub-list

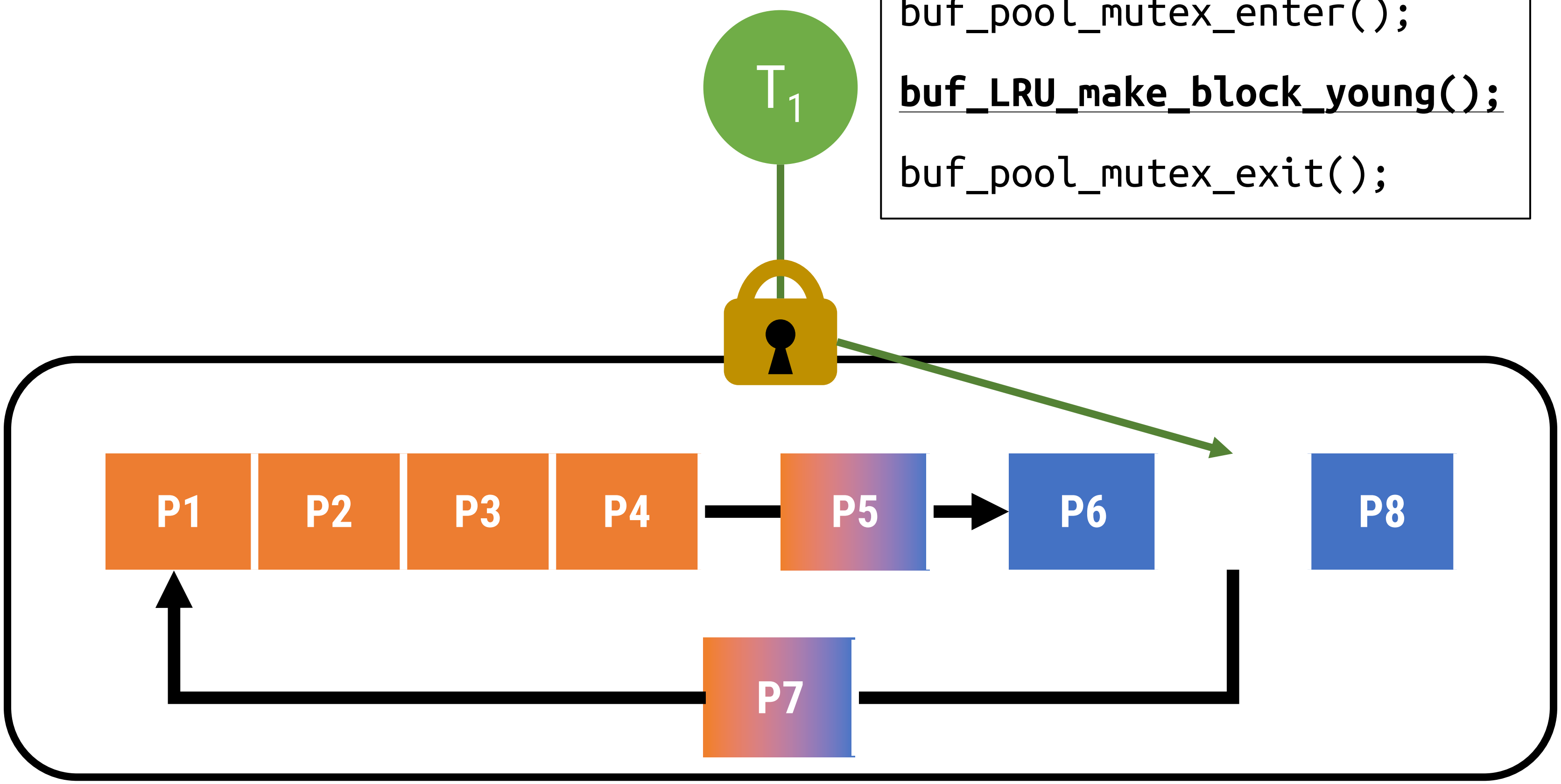
```
buf_pool_mutex_enter();
buf_LRU_make_block_young();
buf_pool_mutex_exit();
```



(Relaxed) LRU Buffer Pool in MySQL

- No precise LRU ordering within the "young" sub-list

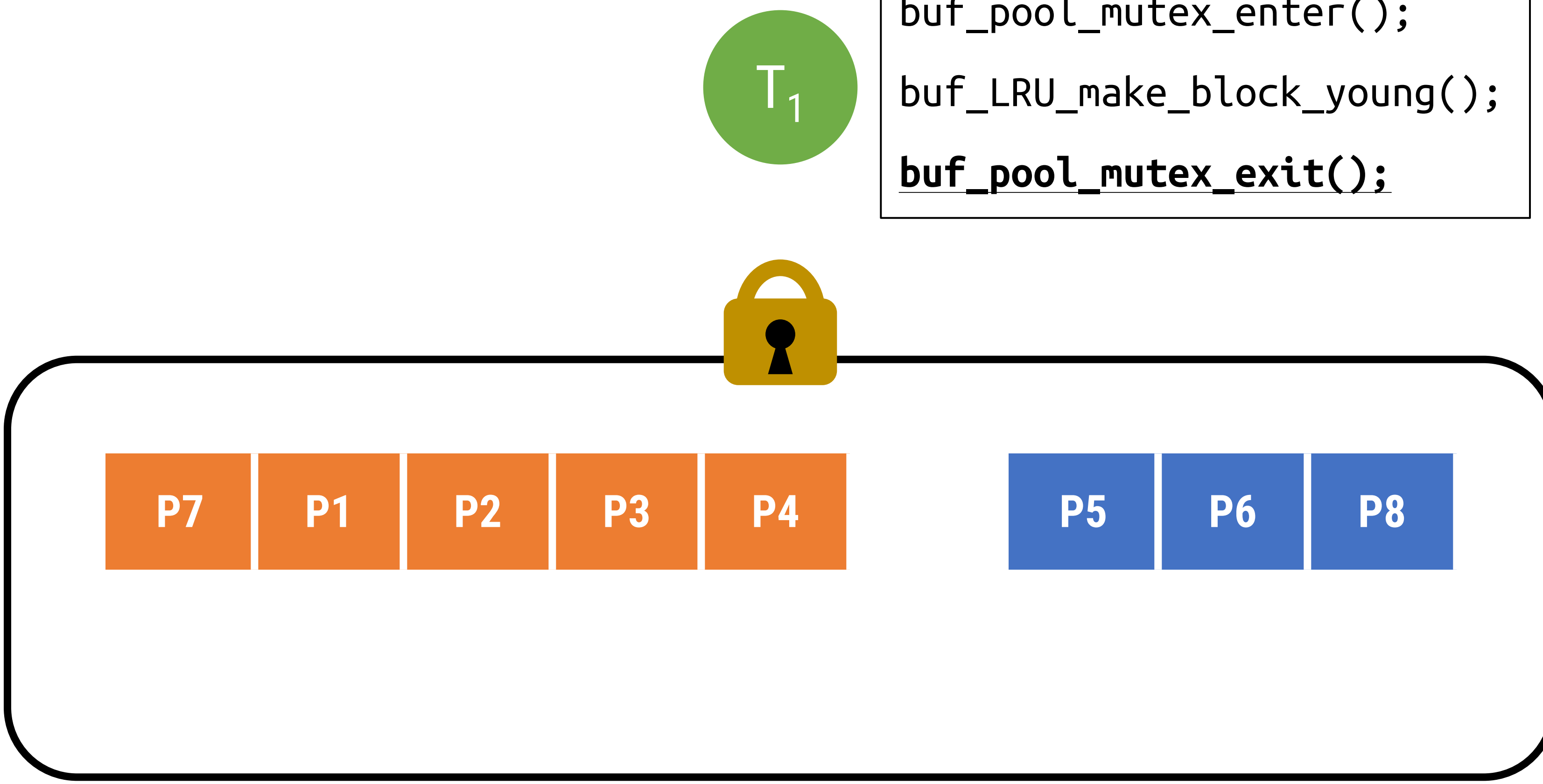
```
buf_pool_mutex_enter();
buf_LRU_make_block_young();
buf_pool_mutex_exit();
```



(Relaxed) LRU Buffer Pool in MySQL

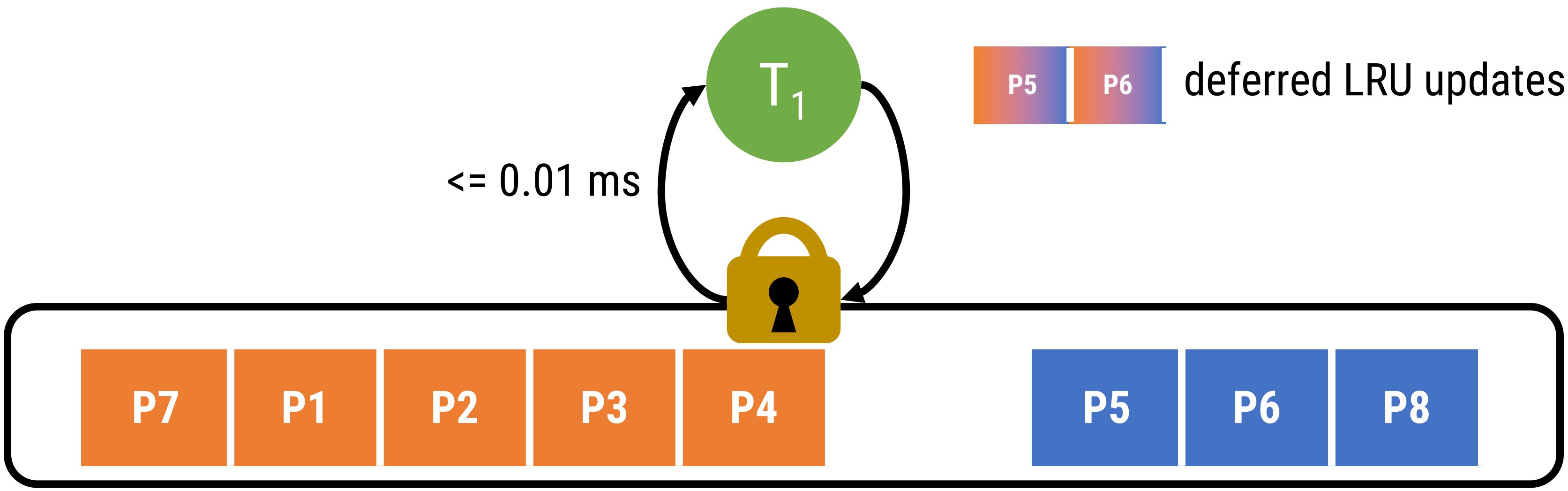
- No precise LRU ordering within the “young” sub-list

```
buf_pool_mutex_enter();
buf_LRU_make_block_young();
buf_pool_mutex_exit();
```



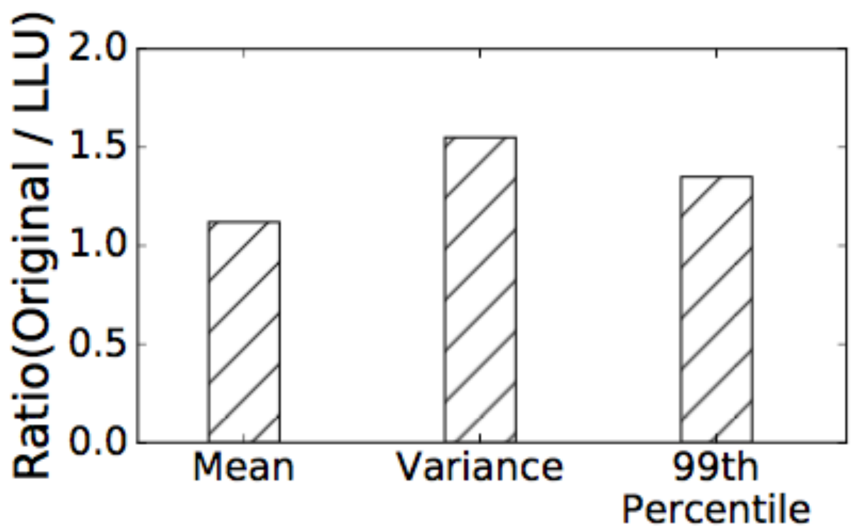
Lazy LRU Update (LLU)

- The mutex can be a bottleneck when the working sets is larger than 5/8 of the buffer pool -> **Further relax LRU ordering**
  - Replace the mutex with a spin lock w/ timeout
  - If failed to acquire the lock within 0.01 ms, defer the update until successfully acquire the lock for another update



Lazy LRU Update (LLU) - Experiment

- Workload: TPC-C (2-WH)



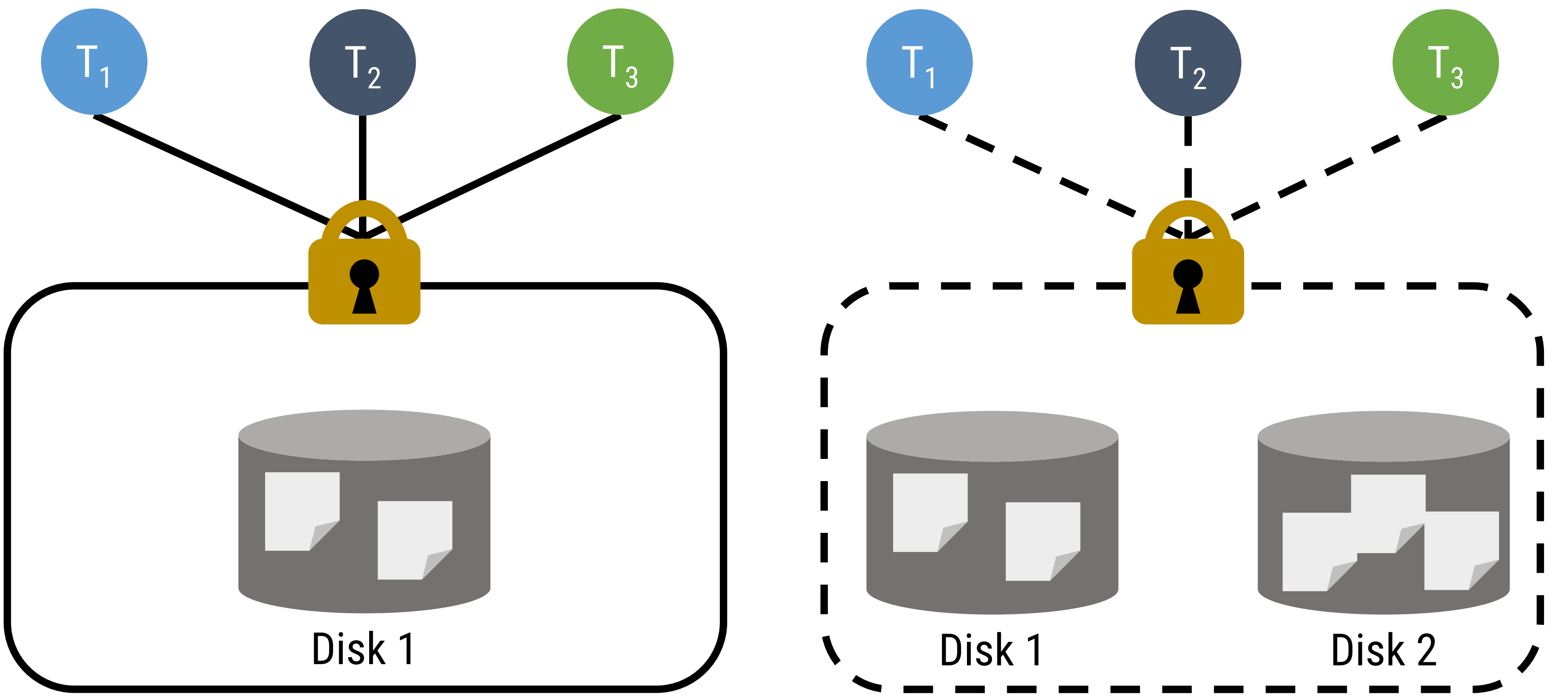
System	Name of the Identified Function	Original contribution to overall variance	Modification	Modified lines of code or config	Ratio of overall latency variances (Orig. / Modified)	Ratio of overall 99^{th} latencies (Orig. / Modif.)	Ratio of overall mean latencies (Orig. / Modif.)
MySQL	`os_event_wait`	59.2%	replace FCFS with VATS	189	5.6x	2.0x	6.3x
MySQL	`buf_pool_mutex_enter`	32.92%	replace mutex with spin lock	46	1.6x	1.4x	1.1x
MySQL	`fil_flush`	5%	parameter tuning	2	1.4x	1.2x	1.2x
Postgres	`LWLockAcquireOrWait`	76.8%	parallel logging	355	1.8x	1.3x	2.4x
VoltDB	`[waiting in queue]`	99.9%	add # of worker threads	1	2.6x	1.4x	5.7x

Mitigation Ideas

- MySQL
  - `os_event_wait` -> **schedule txns. in a variance-aware manner (VATS)**
  - `buf_pool_mutex_enter` -> **update LRU list lazily (LLU)**
- PostgreSQL
  - `LWLockAcquireOrWait` -> **parallelize WAL**
- VoltDB
  - Event queuing time -> **adjust # worker threads**

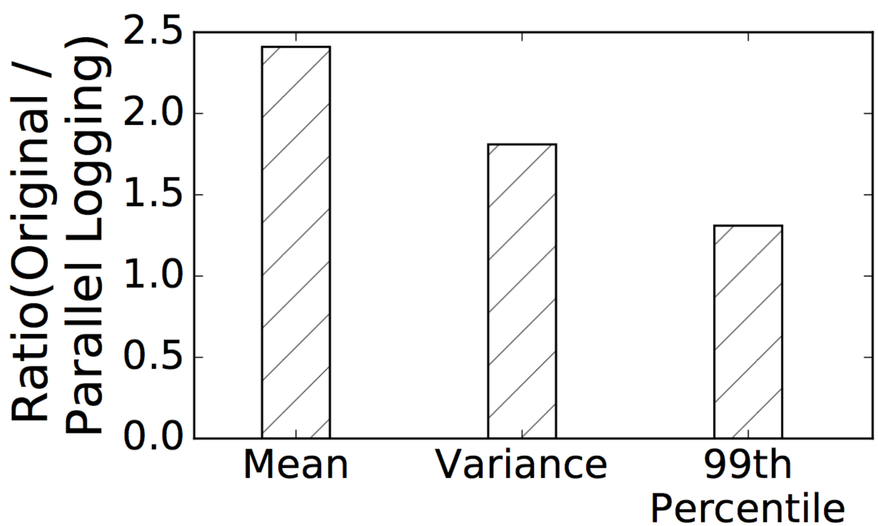
Simple Parallel WAL - Overview

- Uses two hard disks for storing two sets of logs
- Only acquires the global lock when both sets are in use



Simple Parallel WAL - Experiment

- Workload: TPC-C



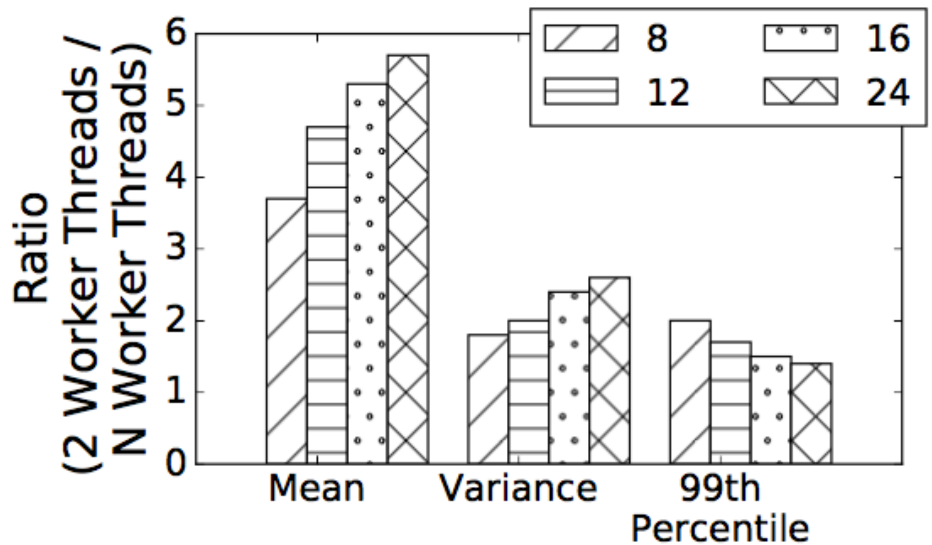
System	Name of the Identified Function	Original contribution to overall variance	Modification	Modified lines of code or config	Ratio of overall latency variances (Orig. / Modified)	Ratio of overall 99^{th} latencies (Orig. / Modif.)	Ratio of overall mean latencies (Orig. / Modif.)
MySQL	os_event_wait	59.2%	replace FCFS with VATS	189	5.6x	2.0x	6.3x
MySQL	buf_pool_mutex_enter	32.92%	replace mutex with spin lock	46	1.6x	1.4x	1.1x
MySQL	fil_flush	5%	parameter tuning	2	1.4x	1.2x	1.2x
Postgres	LWLockAcquireOrWait	76.8%	parallel logging	355	1.8x	1.3x	2.4x
VoltDB	[waiting in queue]	99.9%	add # of worker threads	1	2.6x	1.4x	5.7x

Mitigation Ideas

- MySQL
  - `os_event_wait` -> **schedule txns. in a variance-aware manner (VATS)**
  - `buf_pool_mutex_enter` -> **update LRU list lazily (LLU)**
- PostgreSQL
  - `LWLockAcquireOrWait` -> **parallelize WAL**
- VoltDB
  - Event queuing time -> **adjust # worker threads**

Adjusting # Workers - Experiment

Default # threads: 2

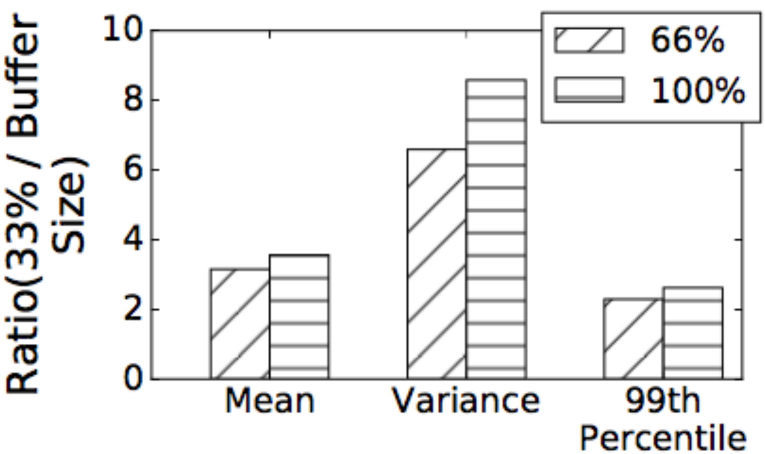


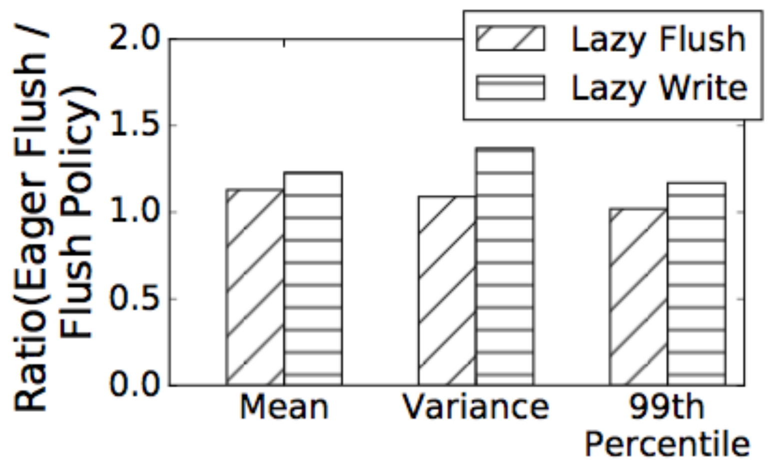
System	Name of the Identified Function	Original contribution to overall variance	Modification	Modified lines of code or config	Ratio of overall latency variances (Orig. / Modified)	Ratio of overall 99^{th} latencies (Orig. / Modif.)	Ratio of overall mean latencies (Orig. / Modif.)
MySQL	`os_event_wait`	59.2%	replace FCFS with VATS	189	5.6x	2.0x	6.3x
MySQL	`buf_pool_mutex_enter`	32.92%	replace mutex with spin lock	46	1.6x	1.4x	1.1x
MySQL	`fil_flush`	5%	parameter tuning	2	1.4x	1.2x	1.2x
Postgres	`LWLockAcquireOrWait`	76.8%	parallel logging	355	1.8x	1.3x	2.4x
VoltDB	`[waiting in queue]`	99.9%	add # of worker threads	1	2.6x	1.4x	5.7x

Variance-Aware Tuning (MySQL)

- buffer pool size:
  33% (default), 66%, 100%
  of the entire DB size

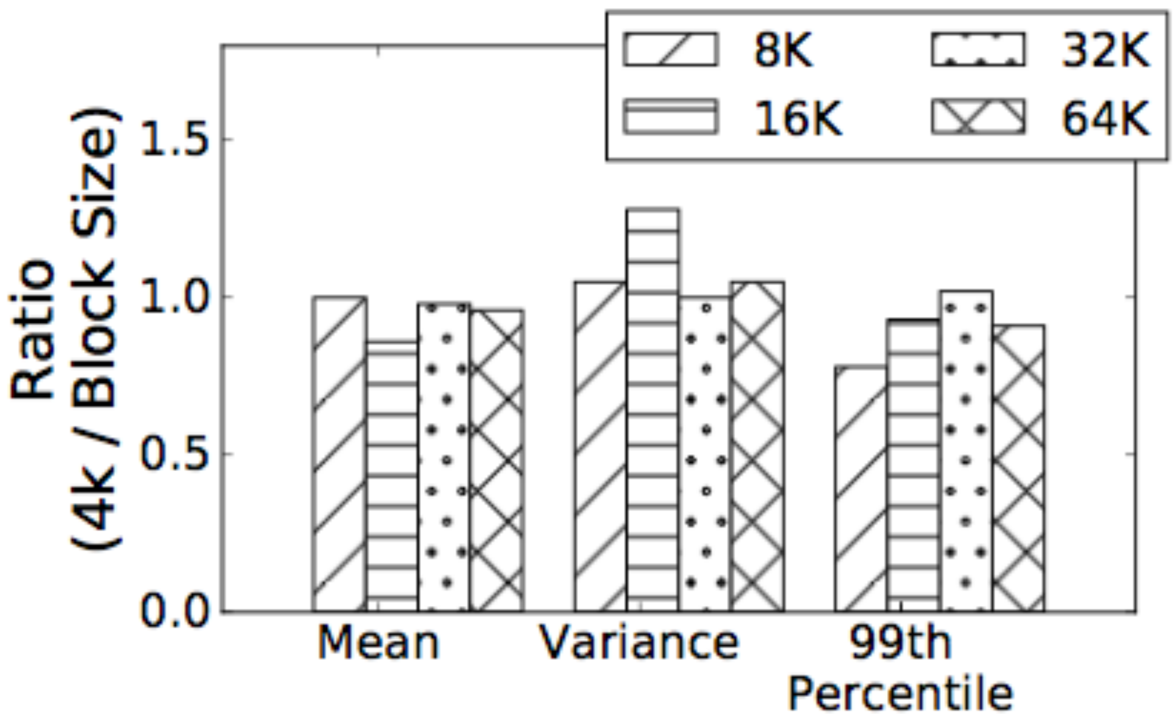
- log flushing policies:
  eager flush, lazy flush,
  lazy write





Variance-Aware Tuning (PostgreSQL)

- I/O block (log buffer) size: 8 (default), 16, 32, 64 KB
- logs may occupy only a small portion of a large block



Real-World Adoption

- VATS has been adopted by MariaDB, and now is its default scheduling policy - https://github.com/MariaDB/server/pull/248

Summary

- **Performance predictability** is getting more important than ever before for modern (OLTP) workloads.
- **TProfiler** has identified major sources of performance unpredictability in MySQL, PostgreSQL, and VoltDB.
- The default FCFS scheduler in MySQL is one major source of performance unpredictability, and VATS scheduler successfully improves predictability, as well as mean latencies.