



LGWR Beyond The Docs

What Oracle didn't Write to Disk Part 3: Exadata Pipelined Log Writes

SOUG Day 02/2026

02 October 2026

Zurich

Christoph Lutz
Swisscom

Blog: <https://www.0xChris.dev>

Twitter: [@chris_skyflier](https://twitter.com/chris_skyflier)

Bluesky: [@christophlutz.bsky.social](https://bsky.app/profile/christophlutz.bsky.social)

Github: [Christoph-Lutz](https://github.com/Christoph-Lutz)





LGWR Features – Timeline

9i	Log File Sync Timeout Log Parallelism (multiple public redo strands)	PART 1
10g	Private Redo Strands & In-Memory Undo	
11g	Adaptive Log File Sync	
12c	Adaptive Scalable LGWR (multiple LG worker processes)	PART 2
19c	Exadata X8M and X9M only (PMEM): Fast Log File Sync (Fast Sync)	PART 1
26ai	Exadata only: Pipelined Log Writes (backported to 19.22+)	PART 3
...	Edge Cases & Surprises	PART 4

There's lots of innovation in LGWR, but unfortunately, all these features are mostly undocumented ...



Before we start ...

This is a low-level technical presentation about internal and undocumented behavior

Beware that:

- Things can change across different versions and patch levels
- My observations, findings and interpretations may be inaccurate or wrong
- Some of the techniques shown in this presentation are dangerous – use them at your own risk!
- Examples and demos were tested in the following environments:
 - **Oracle 19.26** (Exadata 25.2.6, OEL 8.10, kernel 5.15.0-315.196.5.2, bpftrace-0.16)

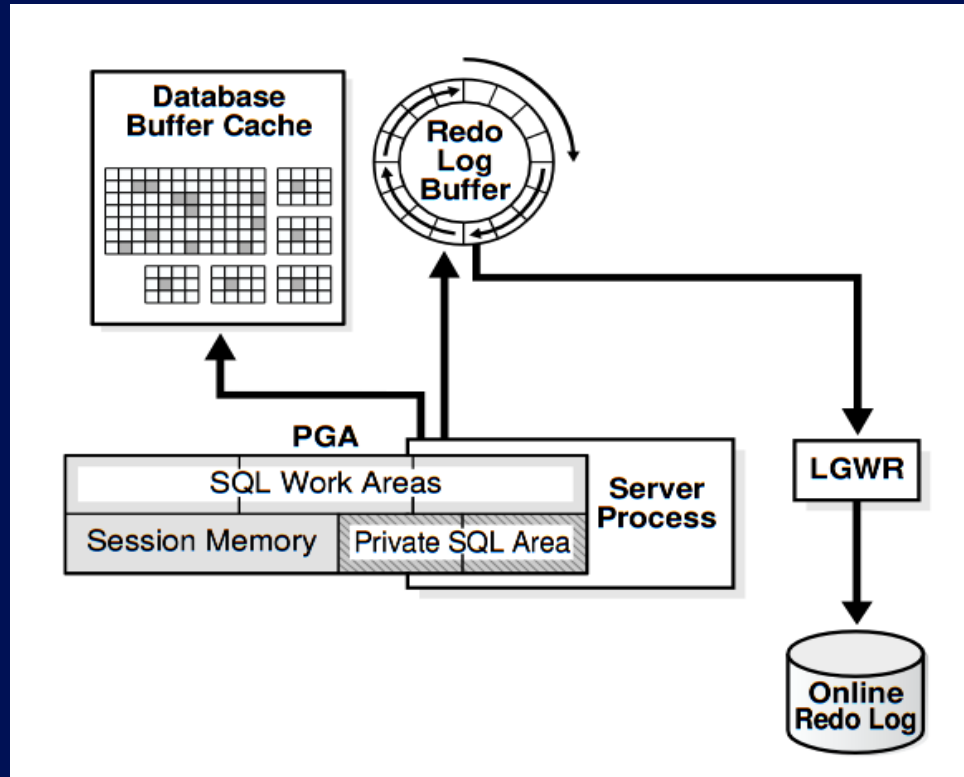




LGWR Fundamentals



LGWR & Redo Log Buffer



The redo log buffer is a **circular buffer** in the SGA that stores redo entries describing changes made to the database.

[...]

The background process **log writer process (LGWR)** writes the redo log buffer to the active online redo log group on disk.

This is not wrong, but it leaves out an awful lot of important details ...

Figure source: [Oracle Corp., Database 19c, Database Concepts, Section 15-21: Redo Log Buffer](#)



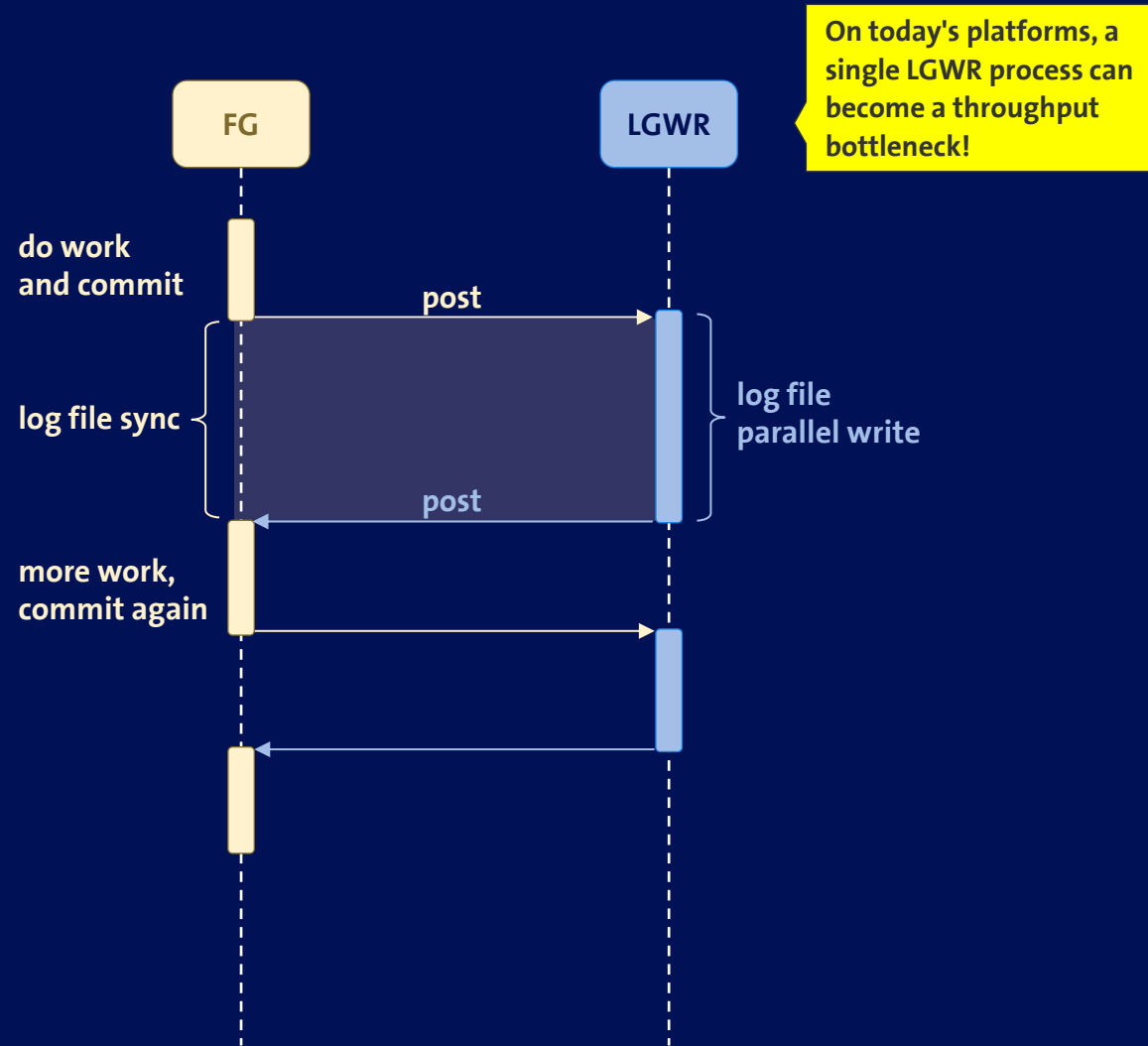
Commit Performance – Key Factors

Commit performance depends on:

1. The **speed of writing** redo records from memory to permanent storage
2. The **delay of receiving confirmation** that the changes are safely persisted

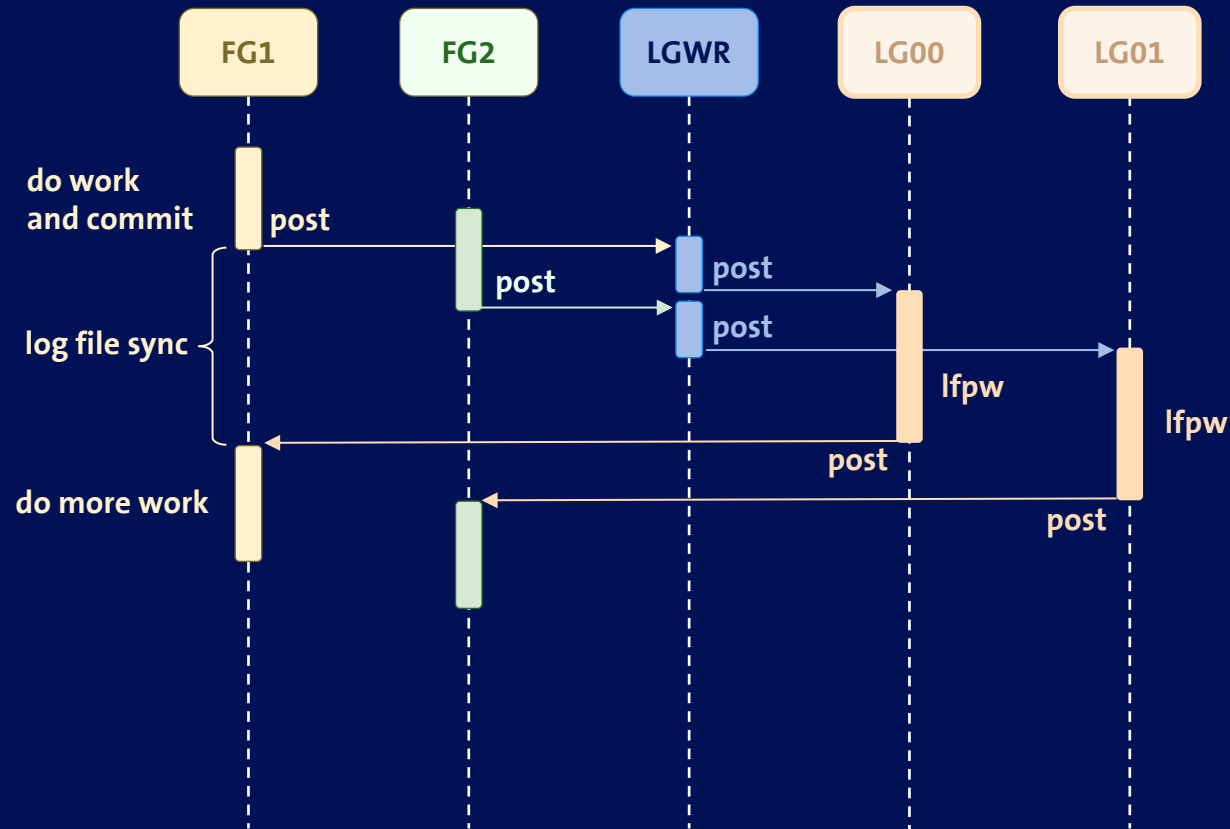


Single LGWR





Adaptive Scalable LGWR (Multiple LGWR) – Pipelined (Overlapped) Redo Write



LGWR coordinates redo write requests across LG workers

LG workers carry out the redo write operations

LGWR uses pipelined writes on all platforms, not only on Exadata.

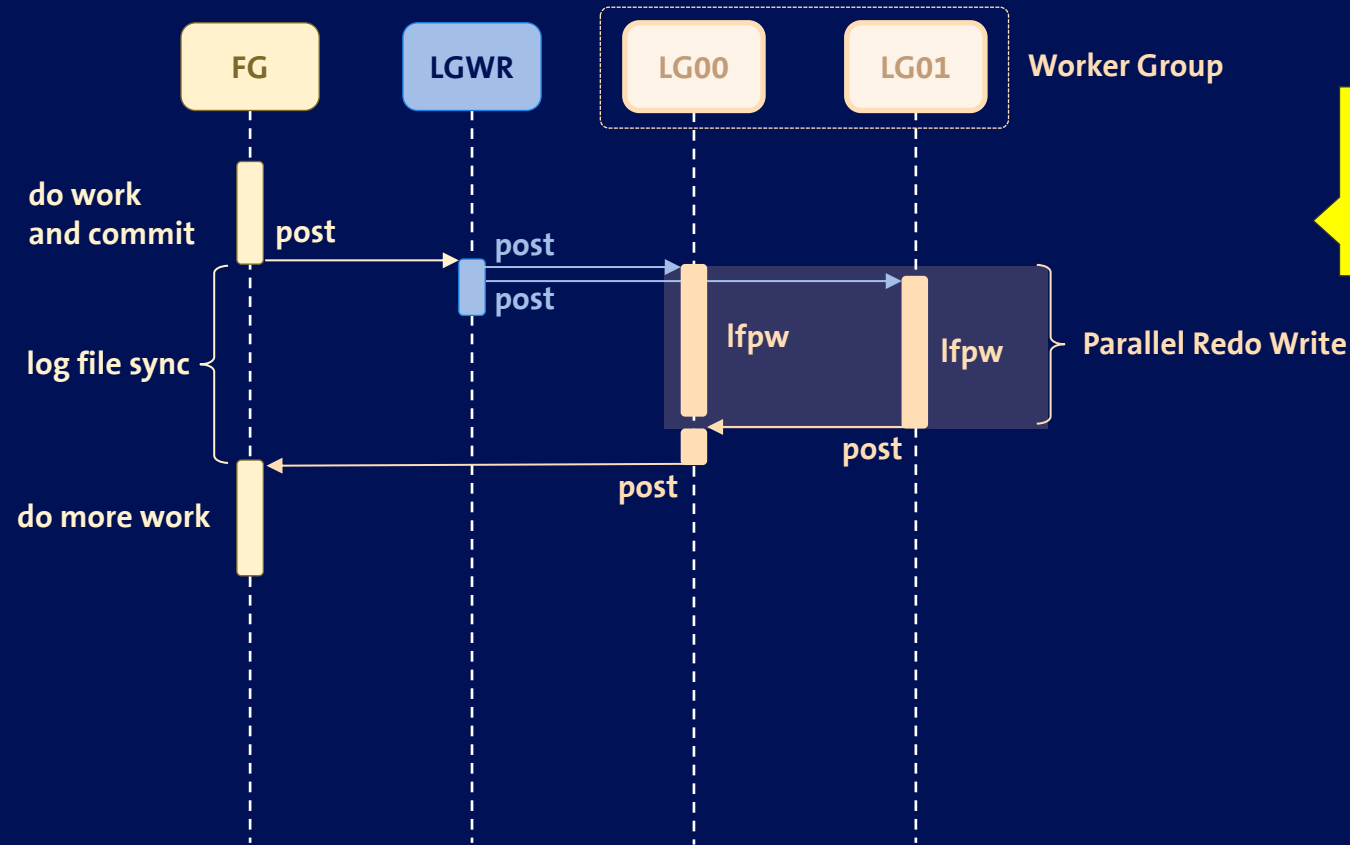
Confusingly, "Pipelined Log Writes" are a marketing name for an optimization that is only available on Exadata (more details later)!

LG workers process multiple redo write requests concurrently!





Adaptive Scalable LGWR (Multiple LGWR) – Parallel Redo Write



LG workers can also process different parallel redo writes concurrently (pipelined + parallel).

LG workers process the same redo write request in parallel!





Latency VS Throughput

Adaptive Scalable LGWR dynamically adapts to workload at runtime!





Adaptive Scalable LGWR – Drawback: Heuristics

The LGWR decision making is **based on heuristics**.

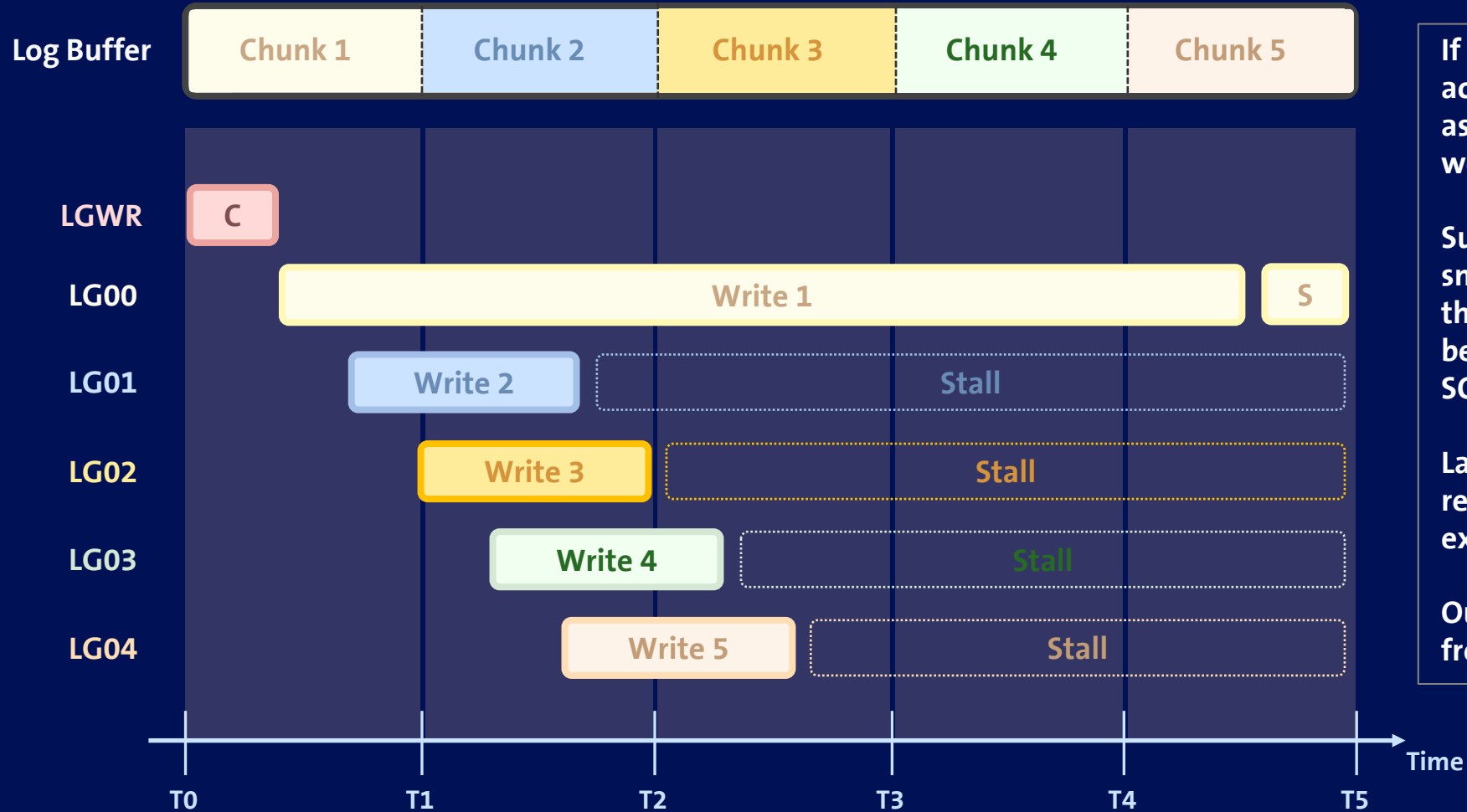
The heuristics are based on factors that **do not always reflect current conditions**:

- Historical sample as a reference point to switch to scalable mode
- Assume 50% overlapped redo writes in the decision to switch back to single mode

The heuristics **react slowly to changes** in workload!



Adaptive Scalable LGWR – Drawback: Pipeline Stalls



If a large amount of redo data accumulates in the log buffer, LGWR assigns a large logical write to a worker.

Subsequent redo writes may be much smaller and therefore complete before the large write – the pipeline stalls because Oracle can't advance the on-disk SCN unless the large write completes!

Large writes are susceptible to response time outliers, further exacerbating the problem!

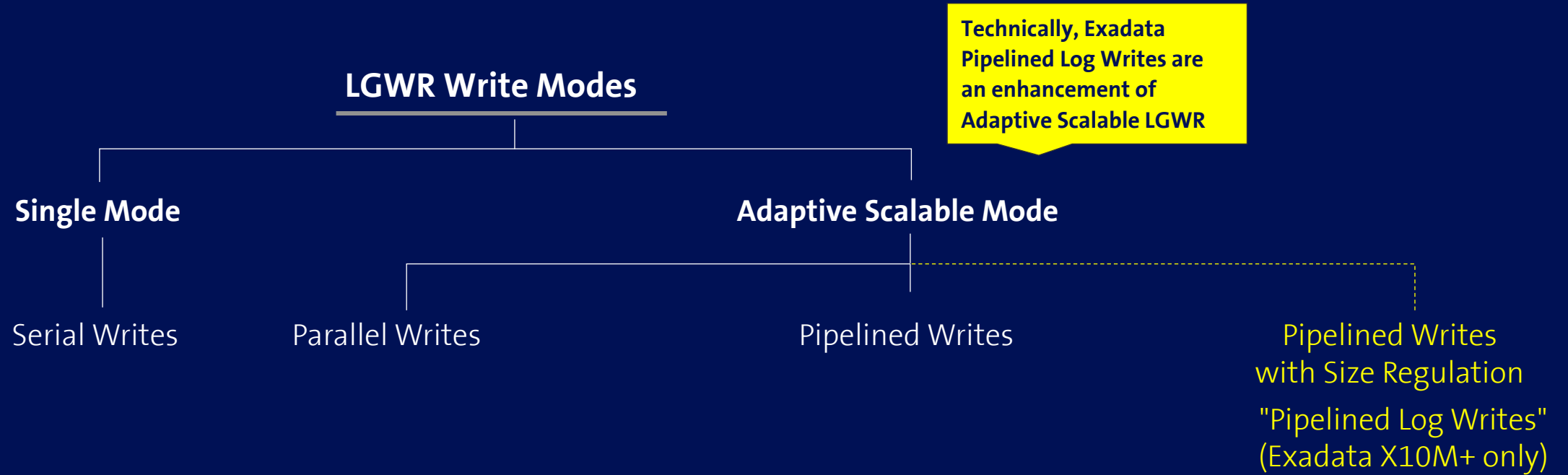
Outlier redo writes tend to increase in frequency with larger redo writes.



Pipelined Log Writes



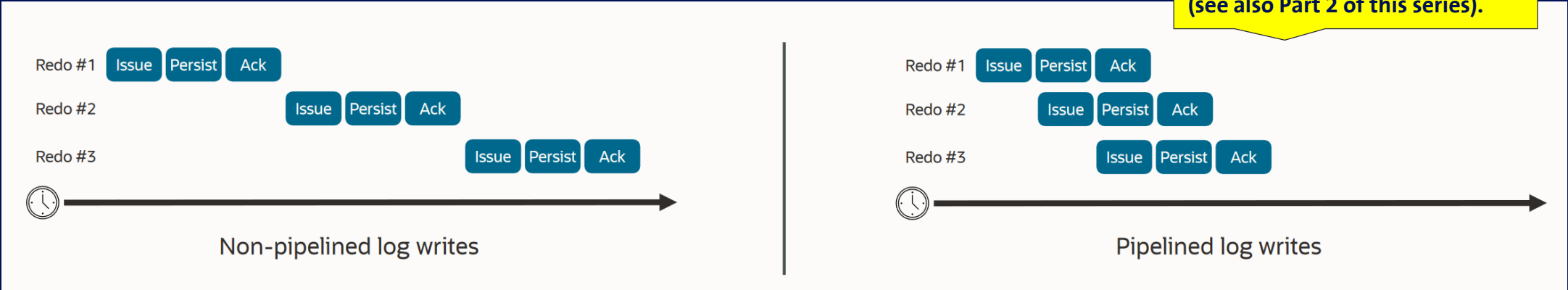
Adaptive Scalable LGWR Modes





Pipelined Log Writes – Overview

The Oracle docs are misleading, LGWR has been using pipelined / overlapped writes since 12c (see also Part 2 of this series).



Only available on Exadata X10 or newer with:

- Exadata System Software 24.1 or higher
- Oracle 26ai
- Oracle 19.22 or higher



Pipelined Log Writes are only enabled if

This refers to the CPU_COUNT at instance startup ("static" CPU_COUNT). The minimum value of 48 is hardcoded in kcrfw_olrw_init_params!

CPU_COUNT $\geq$ 48

(ksbcpueffthrcnt_startup_)

The CPU_COUNT=48 minimum requirement is not documented!





Pipelined Log Writes Disabled

LGWR Tracefile with CPU_COUNT < 48

```
Created 8 redo writer workers (2 groups (0 to 1) of 4 each)
```

```
*** 2025-09-03T13:13:08.312771+02:00 (CDB$ROOT(1))
```

```
Platform for ORLs is Exadata X10M and later
```

```
*** 2025-09-03T13:13:20.670017+02:00 (CDB$ROOT(1))
```

```
Overlapped redo/writes by log writer are disabled (size=32768, stall size=2048, poke=100%, LGWR mode=heuristic)
```

**Adaptive Scalable LGWR
in heuristic mode**



Pipelined Log Writes Enabled

LGWR Tracefile with CPU_COUNT >= 48

```
Created 8 redo writer workers (2 groups (0 to 1) of 4 each)
```

```
*** 2025-09-09T22:11:39.191886+02:00 (CDB$ROOT(1))
```

```
Platform for ORLs is Exadata X10M and later
```

```
*** 2025-09-09T22:11:52.745168+02:00 (CDB$ROOT(1))
```

```
OLRW: Minimum calculated target size for all zones in thick mode adjusted from 1536 to 716
```

```
Adaptive overlapped redo writes using evaluative log writer, are enabled (size=0, stall size=2048, poke=80%)
```



Pipelined Log Writes – LGWR Trace Information

LGWR Tracefile

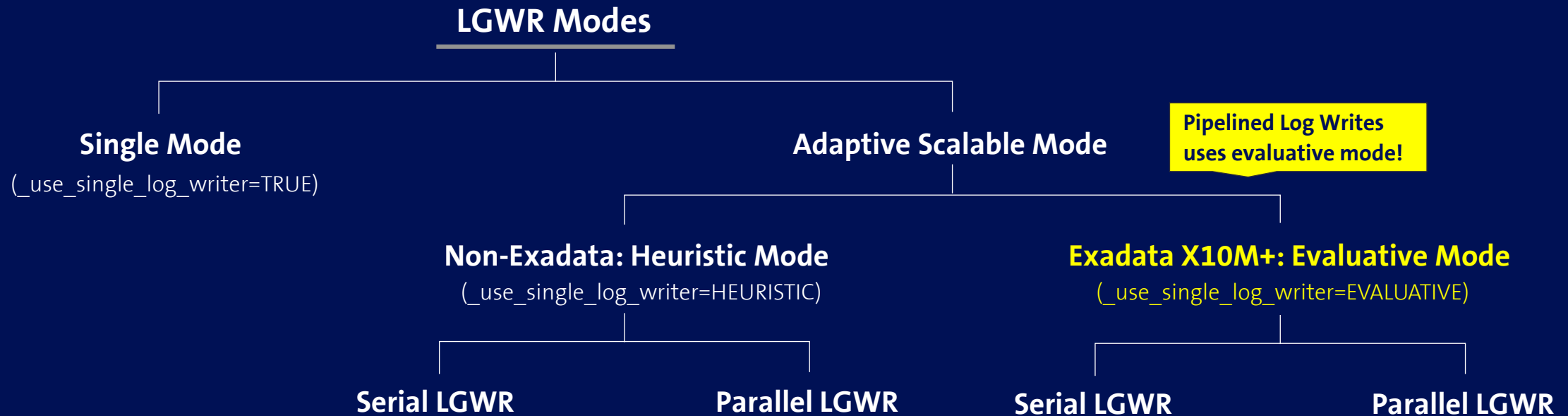
```
*** 2025-10-13T21:45:38.313548+02:00 (CDB$ROOT(1))  
Adaptive scalable LGWR enabling workers  
  
*** 2025-10-13T21:51:38.318961+02:00 (CDB$ROOT(1))  
Adaptive scalable LGWR disabling workers
```

With Pipelined Log Writes, these messages no longer appear, unless trace event 10468 is set at level 1024





Adaptive Scalable LGWR Modes – Parameter `_use_single_log_writer`



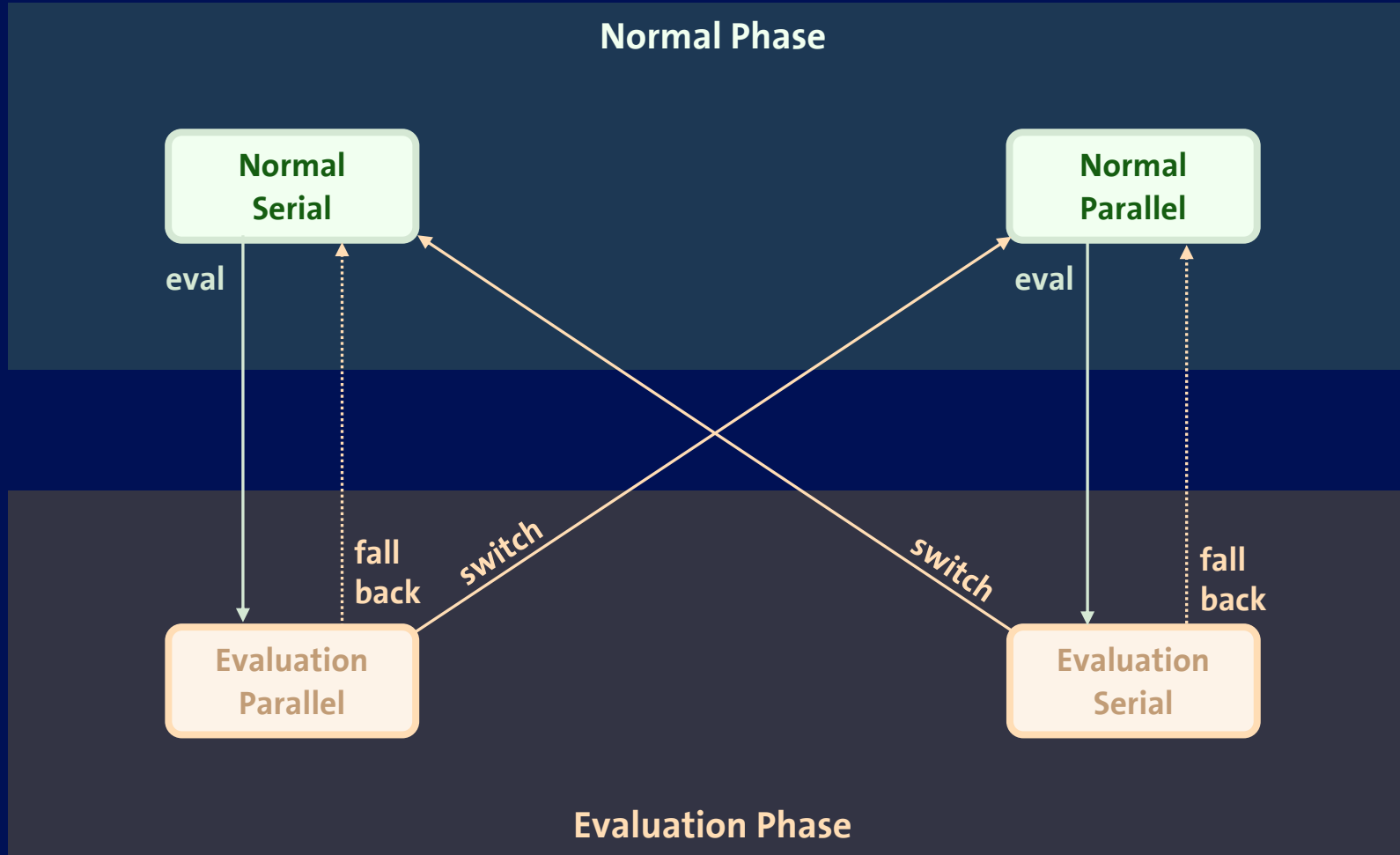


Pipelined Log Writes = ALWE + OLRW

(Adaptive LGWR Evaluation + Overlapped Redo Writes)



Normal & Evaluation Phase – State Transitions



Note: Phase transitions are performed in `kcrfw_slave_adaptive_check_phase`



Pipelined Log Writes – Normal & Evaluation Phase Questions

1. **When** exactly and how frequently do we enter the evaluation phase?
2. **How long** does the evaluation phase last?
3. **What** is checked in the evaluation phase (threshold)?
4. When exactly does the evaluation trigger a LGWR **mode switch**?



The evaluation phase is entered every

15 sec

`_adaptive_scalable_log_writer_evaluation_interval`



The evaluation phase runs until

1 %

of the normal-phase redo blocks have been processed

Parameter: `_adaptive_scalable_log_writer_evaluation_fraction` (default: $100/10000 = 1\%$)



A LGWR mode switch from serial to parallel occurs if:

**eval redo
rate**



**normal
redo rate**



A LGWR mode switch from parallel to serial occurs if:

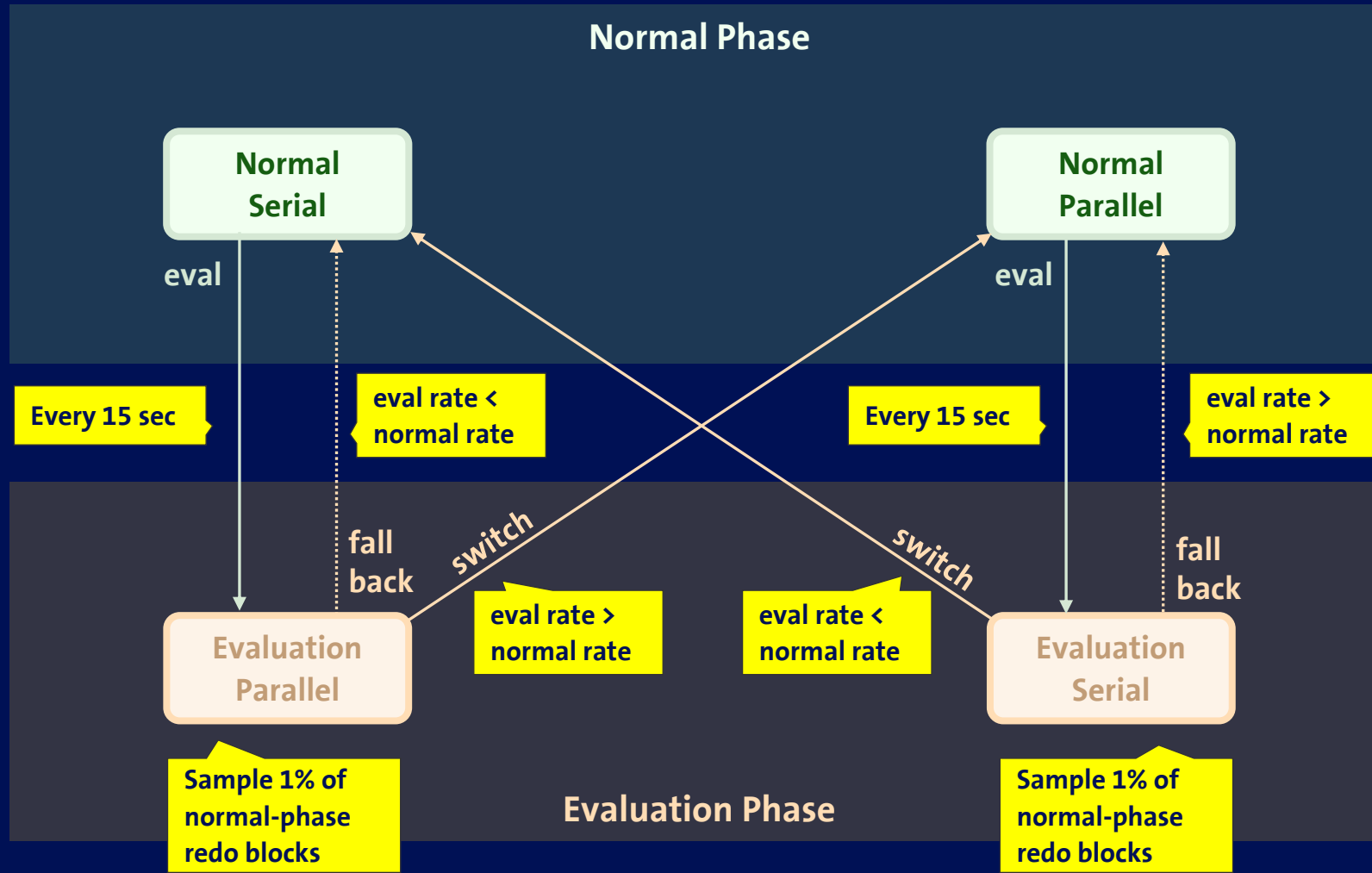
**eval redo
rate**



**normal
redo rate**



Normal & Evaluation Phase – State Transitions



Note: Phase transitions are performed in `kcrfw_slave_adaptive_check_phase`



Overlapped Redo Writes – Normal Phase Computations

```
normal_blocks_per_sec = (tot_blocks / tot_time) * 1e6
```

```
evaluation_blocks = tot_blocks * (eval_fraction / 1e4)
```

tot_blocks:

Total number of redo blocks written during the normal phase measurement interval (15 sec).

tot_time:

Total time (in usec) spent doing redo writes during the normal phase measurement interval (15 sec).

evaluation_blocks:

Number of blocks to evaluate in the evaluation phase (defaults to 1% of the redo blocks written during the normal phase).

eval_fraction:

`_adaptive_scalable_log_writer_evaluation_fraction` in 100 10000ths (default: 100/10000 = 1%).

Compute the avg redo write throughput (in redo blocks per sec) and the number of blocks to evaluate (1%)!





Overlapped Redo Writes – Evaluation Phase Computations

```
eval_blocks_per_sec = (tot_blocks / tot_time) * 1e6
```

```
eval_blocks_per_sec > normal_blocks_per_sec ?
```

tot_blocks:

Total number of redo blocks written during the evaluation phase measurement interval.

tot_time:

Total time (in usec) spent doing redo writes during the evaluation phase measurement interval.

normal_blocks_per_sec:

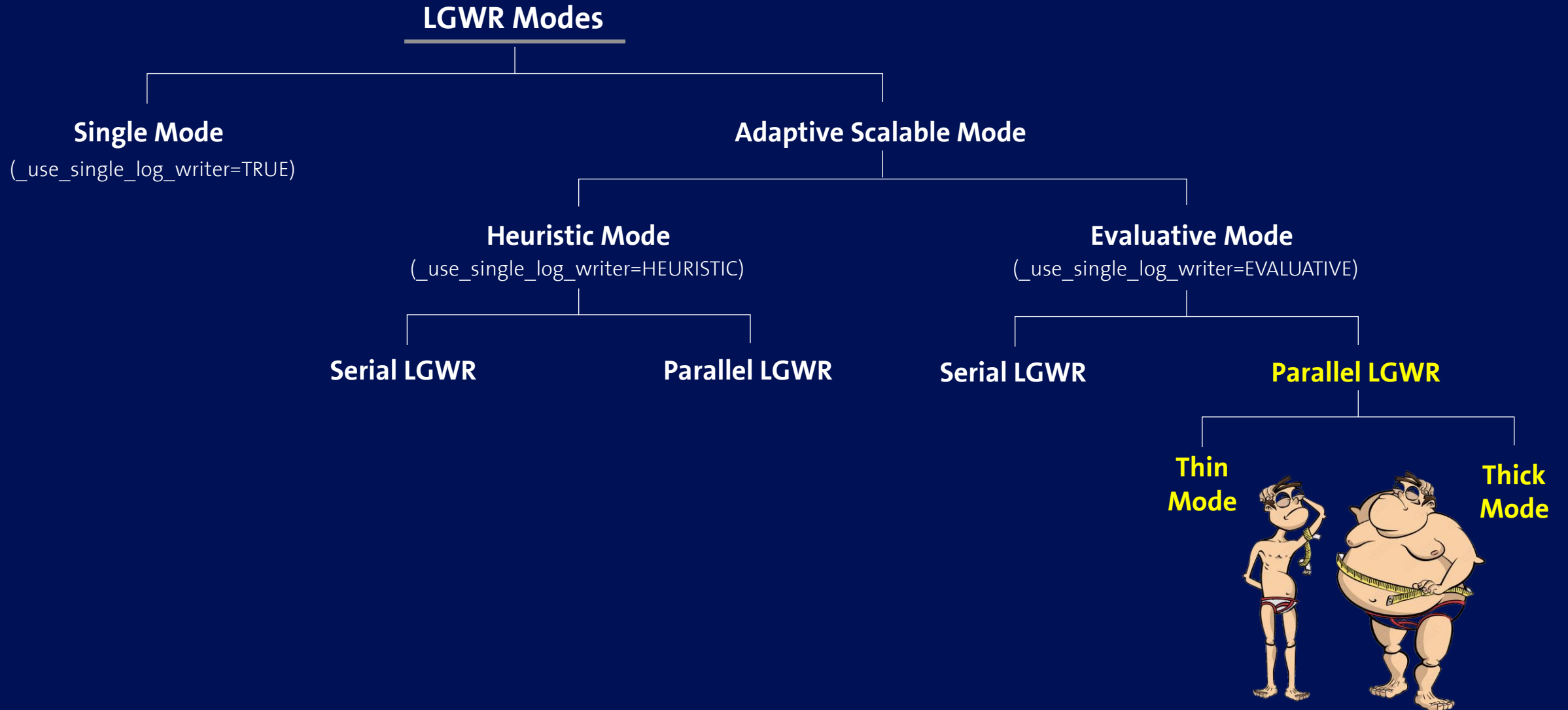
Avg redo write (in blocks/sec) measured during the normal phase.

A mode switch occurs if the redo rate (in redo blocks/sec) during evaluation exceeds the normal-phase rate!



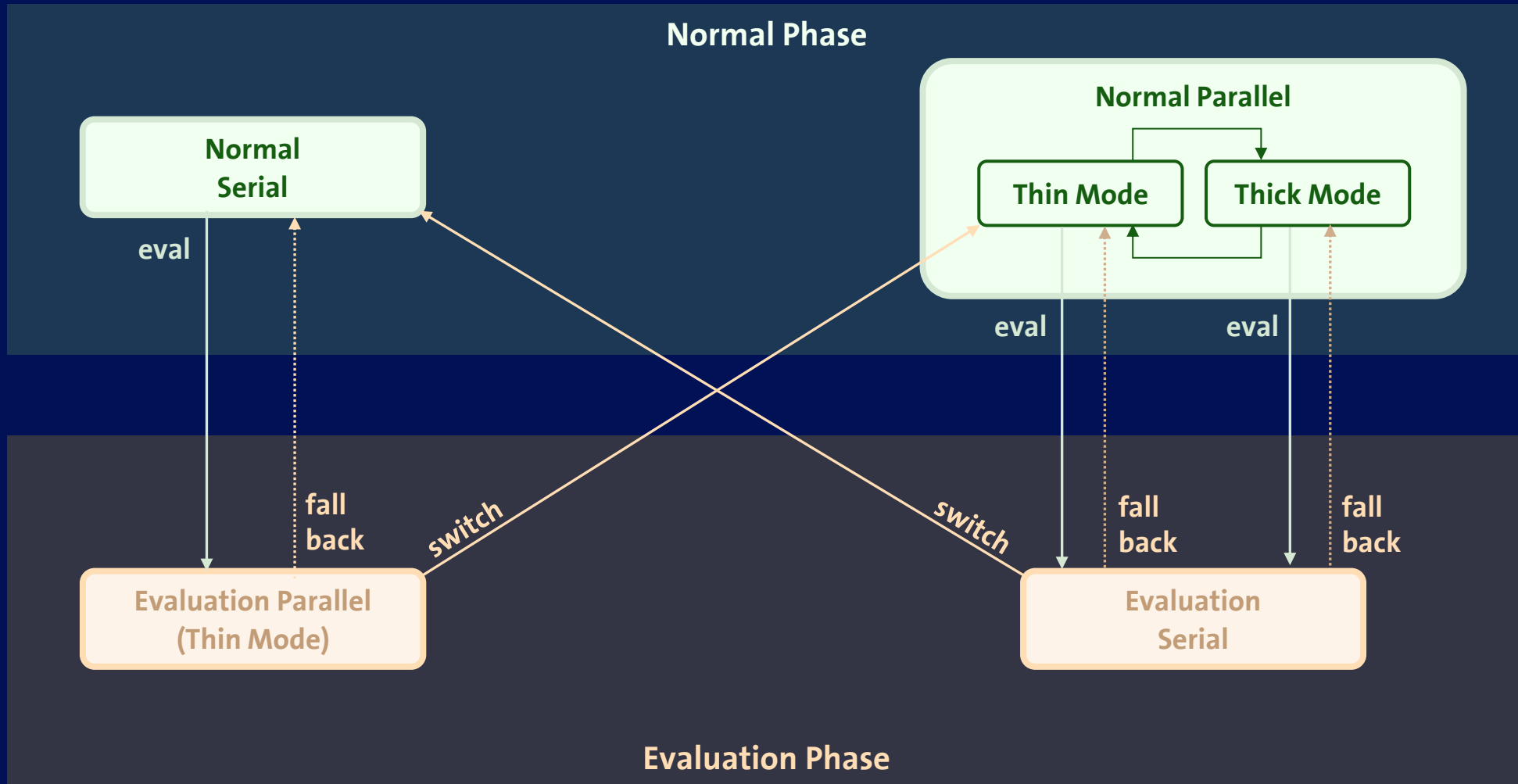


Pipelined Log Writes – Thin & Thick Mode





Pipelined Log Writes – Normal & Evaluation Phase State Transitions



Notes:

- Serial/parallel mode switches are performed in `kcrfw_slave_adaptive_check_phase`
- Thin/thick mode switches are performed in `kcrfw_defer_write`



Pipelined Log Writes – Thin vs Thick Mode

Thin Mode



Optimize for commit latency

Thick Mode



Optimize for write throughput +
stabilize latency under load

Time

LGWR dynamically switches between thin and thick mode at runtime!





Pipelined Log Writes – Thin vs Thick Mode

Thin Mode: Optimize Commit Latency

- Distribute redo write requests across as many LG workers as possible
- Maintain a "small" and consistent redo write size
- Goal: avoid waiting for an LG worker to become available to process a new redo write request (reduce "log file sync" wait time)

Thick Mode: Optimize Redo Write Throughput

- Start optimizing for throughput when all LG workers in thin mode are busy
- Maintain a "large" and consistent redo write size
- Goal: increase throughput while preventing outlier redo write latencies



WHACK-A-MOLE



Pipelined Log Writes – OLRW Thin & Thick Mode Zones





Pipelined Log Writes – Highest Busy Group (hi_busy_grp_num)

A worker group can contain one or more workers. I use the terms loosely and interchangeably throughout this presentation.

Example 1: No worker busy

hi_busy_grp_num = -1



Example 2: 7 workers busy, 9 workers free

hi_busy_grp_num = 6



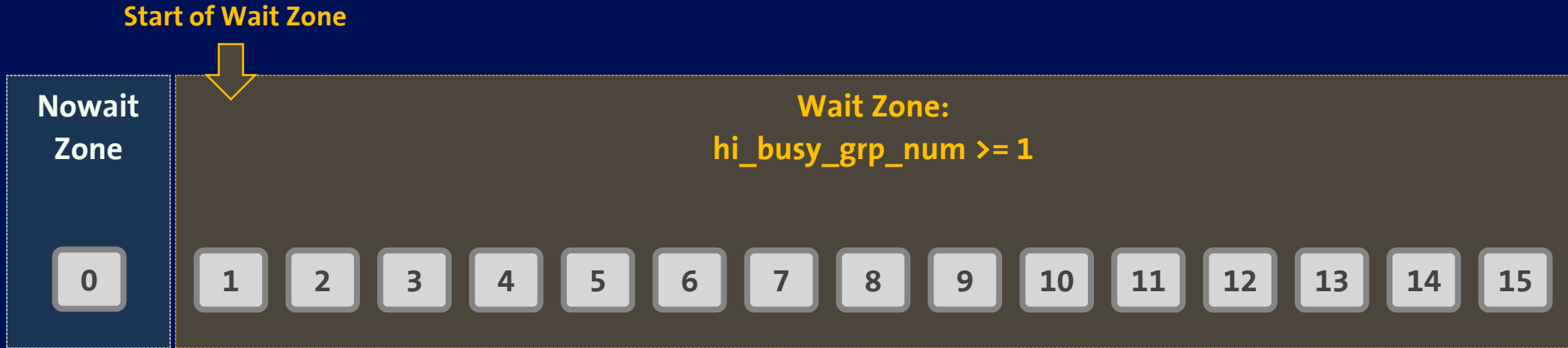
Example 3: All workers busy (hi_busy_grp_num = _max_outstanding_log_writes - 1)

hi_busy_grp_num = 15





Thin Mode – Nowait Zone & Wait Zone



Nowait Zone

- Zero or one worker groups busy
- Start redo **write immediately**

Wait Zone

- At least two worker groups busy
- LGWR defers the redo write until either a **write size threshold** or a **timeout threshold** is reached



Thin Mode Wait Zone – Wait Event "target log write size"

target log write size



LGWR waits until either

write size threshold reached

timeout threshold reached

LGWR dynamically computes the write size and timeout thresholds at runtime!



Target log write size is a busy wait!



LGWR spins and burns CPU while waiting!



Pipelined Log Writes – Thin Mode Configuration

Description:

adaptive overlapped writes thin mode configuration

`_lgwr_thin_mode_config = 1:256:256:100:750`

P1: start of wait zone

P2: minimum write size for wait zone

P3: maximum write size for wait zone

P4: minimum timeout for wait zone (usec)

P5: maximum timeout for wait zone (usec)



Pipelined Log Writes – Thin Mode Timeout Threshold Calculation

The timeout threshold increases as the number of LG workers increases, and vice versa.

`_lgwr_thin_mode_config`

P1: start of wait zone = 1

P2: minimum write size for wait zone = 256

P3: maximum write size for wait zone = 256

P4: minimum timeout for wait zone (usec) = 100

P5: maximum timeout for wait zone (usec) = 750

$$\text{step_inc} = \frac{\text{wait_max_tmo} - \text{wait_min_tmo}}{\text{max_outstanding_log_writes} - \text{wait_zone_start} - 1}$$

$$\text{timeout} = \text{wait_min_tmo} + \text{step_inc} * (\text{hi_busy_grp_num} - \text{wait_zone_start})$$

Example:

`max_outstanding_log_writes` = 16

$\text{step_inc} = (750 - 100) / (16 - 1 - 1) = 650 / 14 = 46$

$\text{timeout} = 100 + 46 * (6 - 1) = 100 + 46 * 5 = 376$

`hi_busy_grp_num` = 6



Fewer LG workers busy:
Decrease timeout

More LG workers busy:
Increase timeout



Pipelined Log Writes – Thin Mode Write Size Threshold Calculation

With the default settings,
the "step increment" is 0, so
write_sz always equals min_
write_sz!

`_lgwr_thin_mode_config`

P1: start of wait zone = 1

P2: minimum write size for wait zone = 256

P3: maximum write size for wait zone = 256

P4: minimum timeout for wait zone (usec) = 100

P5: maximum timeout for wait zone (usec) = 750

$$\text{step_inc} = \frac{\text{max_write_sz} - \text{min_write_sz}}{\text{max_outstanding_log_writes} - \text{wait_zone_start} - 1}$$

$$\text{write_sz} = \text{min_write_sz} + \text{step_inc} * (\text{hi_busy_grp_num} - \text{wait_zone_start})$$

Example:

`max_outstanding_log_writes` = 16

`step_inc` = $(256 - 256) / (16 - 1 - 1) = 0 / 14 = 0$

`write_sz` = $256 + 0 * (6 - 1) = 256 + 0 * 5 = 256$

`hi_busy_grp_num` = 6





Switch from Thin to Thick mode when:

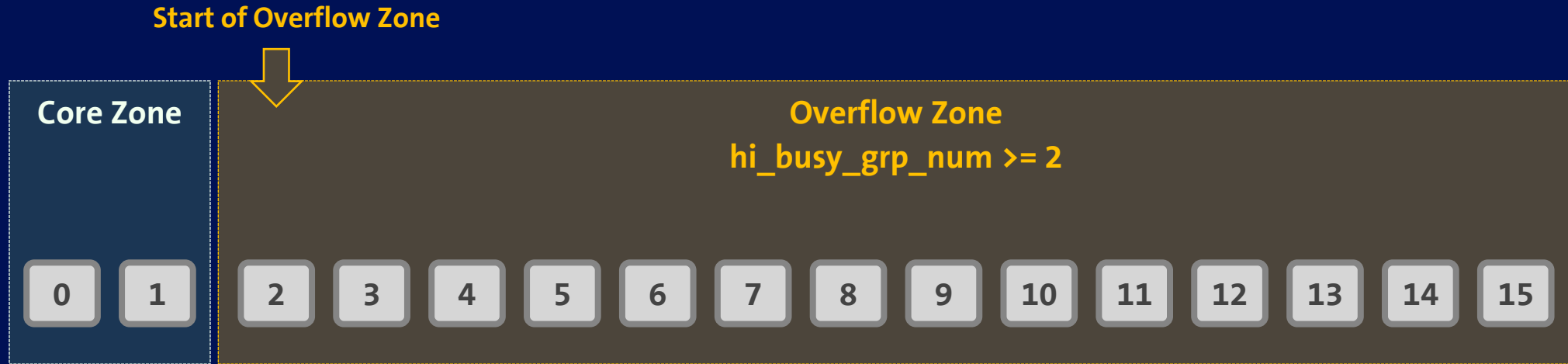
**all worker groups are busy
three times in a row**

3 x





Pipelined Log Writes – Thick Mode Zones



Core Zone

- Zero, one, or two worker groups busy
- **Decrease the target write size** by 64 redo blocks (32 KB) on each write

Overflow Zone

- At least three worker groups busy
- **Increase the target write size** by 64 redo blocks (32 KB) on each write

As the target write size increases, the number of busy workers tends to decrease, eventually falling back into the core zone!





Thick Mode Wait Zone – Wait Event "target log write size"

target log write size



LGWR waits until either

write size threshold reached

timeout threshold reached

The write size changes dynamically depending on thick mode zone:

- Core zone: decrease by 64 redo blocks (32 KB)
- Overflow Zone: increase by 64 redo blocks (32 KB)

The wait occurs in `kcrfw_defer_write` before `lgwr` issues a new redo write request

1500 us (1.5 ms)

(fixed by default settings in `_lgwr_thick_mode_config`)



Pipelined Log Writes – Thick Mode Configuration

Description:

adaptive overlapped writes thick mode configuration

`_lgwr_thick_mode_config = 1:2:1536:8192:1500`

This value is automatically adjusted at instance startup, depending on the following settings:

`_lgwr_thin_mode_config`
`_max_outstanding_log_writes`
(derived from `CPU_COUNT`)



P1: start of fallback zone

P2: start of overflow zone

P3: minimum target size for all zones

P4: maximum target size for all zones

P5: maximum timeout (usec)



Pipelined Log Writes – Thick Mode: Minimum Write Size Adjustment

$$\frac{2 * (\text{max_outstanding_log_writes} - \text{wait_zone_start}) * (\text{wait_max_sz} + \text{wait_min_size})}{(\text{fallback_zone_end} * 5) + 5}$$

Example:

cpu_count = 48
max_outstanding_log_writes = 8
wait_zone_start = 1
fallback_zone_end = 1
wait_max_sz = 256

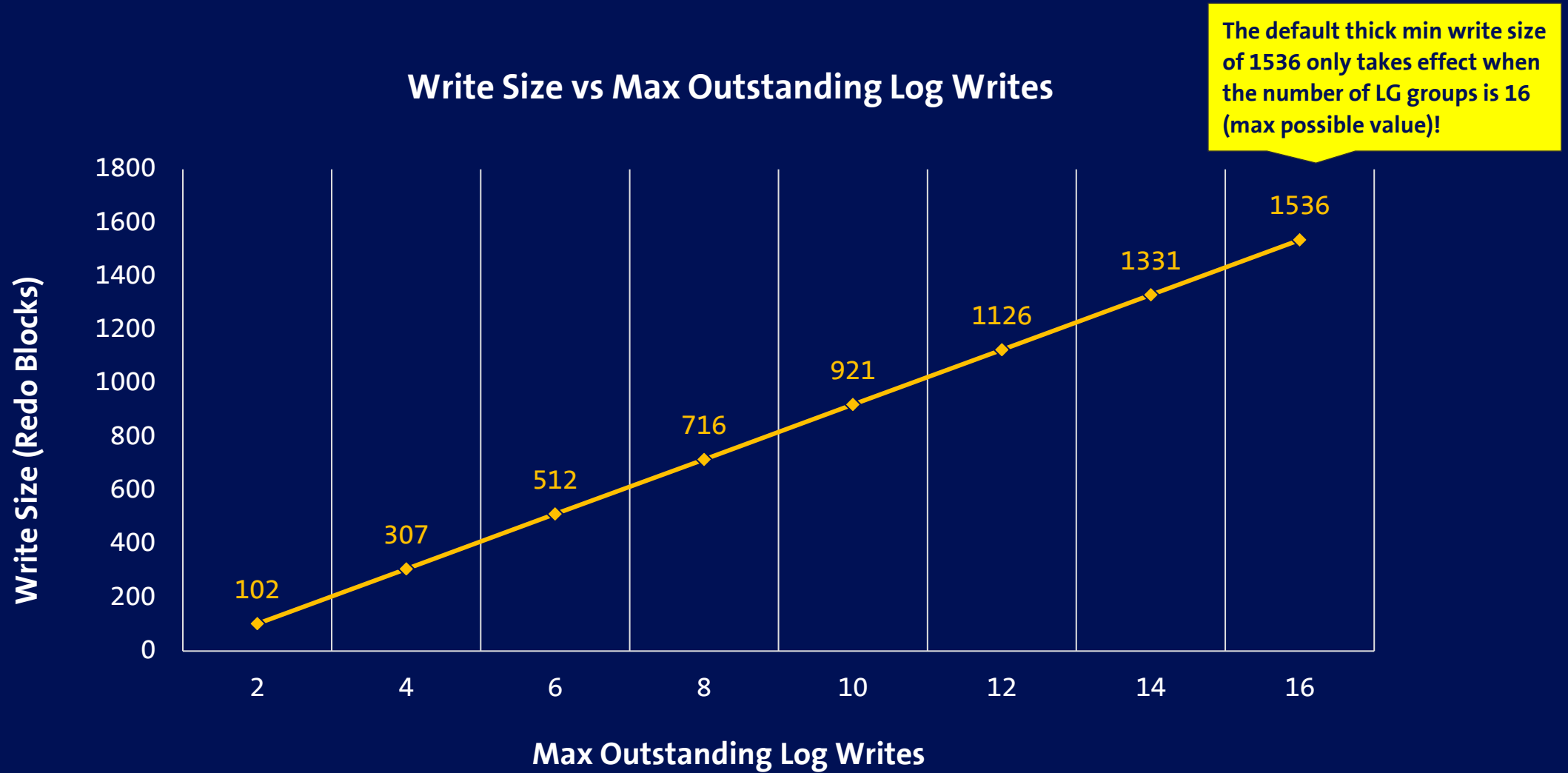
$$\text{thick_write_sz_adj} = ((8 - 1) * 2 * (256 + 256)) / ((1 * 5) + 5) = \mathbf{716.8}$$

Additional adjustments are automatically made in kcrfw_olrw_init_params at instance startup (s. also [Appendix A](#)).

OLRW: Minimum calculated target size for all zones in thick mode adjusted from **1536** to **716**



Pipelined Log Writes – Thick Mode: Min Write Size Adjustment



Fall back to thin mode when
write_sz <= thick_min_sz three times in a row



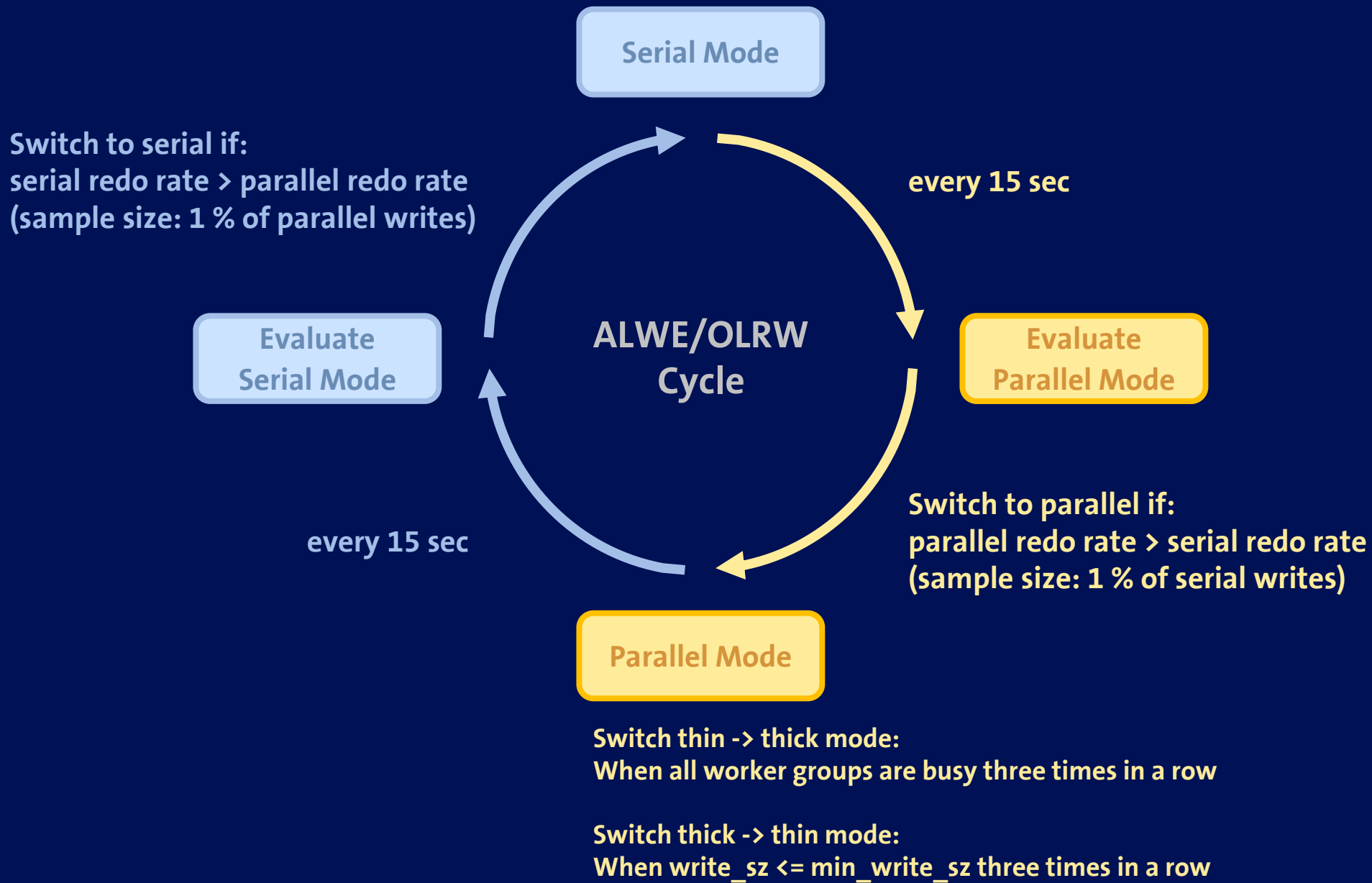
Fallback Zone



Pipelined Log Writes – Fallback Zone: $\text{write_sz} \leq \text{thick_min_sz}$



- Case 1 – In Core Zone: increment the arbiter and switch to thin mode if arbiter == 3
- Case 2 – In Overflow Zone: Increment the write size by 64 redo blocks (32 KB)





Pipelined Log Writes – ALWE Preconditions

Phase transitions only occur if the following preconditions are met (checked in `kcrfw_slave_adaptive_check_phase`):

Normal phase -> evaluation phase

- Evaluation interval > 15 sec
- Number of redo writes ≥ 32

Evaluation phase -> normal phase

- Number of redo writes ≥ 16
- Number of redo blocks left to evaluate ≤ 0
(where number of blocks to evaluate = 1% of blocks written during normal phase)



Additional Observations & Special Cases



Pipelined Log Writes – Switch from Thin to Thick Mode: Initial Write Size

When LGWR switches from thin to thick mode, the initial thick mode write size is calculated as follows:

$$\text{write_sz} = (\text{thin_wait_max_sz} * \text{max_outstanding_log_writes} + \text{thick_min_sz}) / 2$$

thin_wait_max_sz:

Minimum write size for wait zone in thin mode.

max_outstanding_log_writes:

Maximum number of outstanding log writes (= number of LG worker groups)

thick_min_sz:

Minimum write size for all zones when in thick mode.

Note: this is the minimum write size that is automatically adjusted at startup!

Example:

thin_wait_max_sz = 256

max_outstanding_log_writes = 8

thick_min_sz (adjusted) = 716

write_sz = (256 * 8 + 716) / 2 = **1382**



Pipelined Log Writes – Switch from Thin to Thick Mode: Initial Write Size (Trace)

Event 10468, Level 0x18000

2025-11-29T23:09:38.658178+01:00

OLRW: **thin:** wait zone: hi_busy_grp_num=7 **arbiter=1**, tmo_usec=748, **write_sz=256**

OLRW: thin: about to spin: tmo_usec=748, write_sz=256, hi_busy_grp_num=7

OLRW: thin: finished spin:, tmo_usec=748, write_sz=256, tmo_usec_elapsed=841, hi_busy_group_num=7, is_target_sz=F, nowait_grps_are_idle=F, timeout=T

OLRW: **thin:** wait zone: hi_busy_grp_num=7 **arbiter=2**, tmo_usec=748, **write_sz=256**

OLRW: thin: about to spin: tmo_usec=748, write_sz=256, hi_busy_grp_num=7

OLRW: thin: finished spin:, tmo_usec=748, write_sz=256, tmo_usec_elapsed=814, hi_busy_group_num=7, is_target_sz=F, nowait_grps_are_idle=F, timeout=T

OLRW: **thin: switching to thick mode:** hi_busy_grp_num=4294967295, **arbiter=3**, tmo_usec=1500, **write_sz=1382**

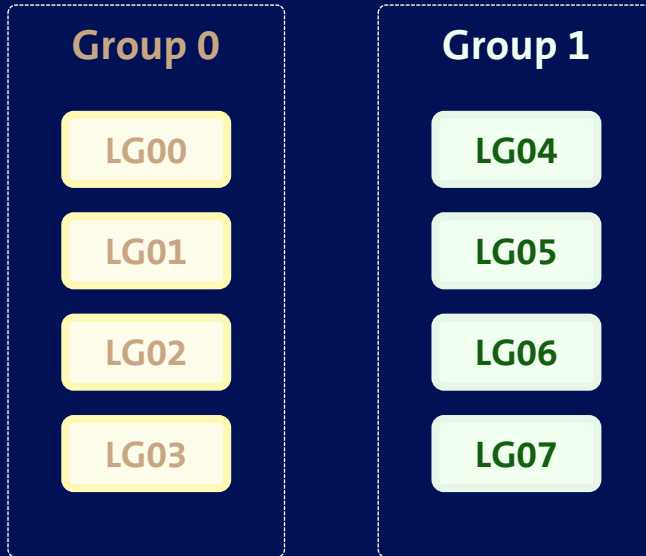
OLRW: thick: about to spin: tmo_usec=1500, write_sz=1382, hi_busy_grp_num=4294967295

OLRW: thick: finished spin:, tmo_usec=1500, write_sz=1382, tmo_usec_elapsed=2181, hi_busy_group_num=7, is_target_sz=F, nowait_grps_are_idle=F, timeout=T



OLRW LG Groups and Workers Remap & Recomputation (8 Workers)

At Instance Startup



After Instance Startup





OLRW Worker Remap Example: CPU_COUNT=48

OLRW: Remapping 4 to 1 slaves_per_group

```
OLRW: remap_existing: num_groups_pre_remap = 2, interim_numslave = 8, interim_numgroups_pre_remap = 2,  
      interim_numgroups_post_remap = 8, current_group_size = 4, target_group_size = 1
```

```
OLRW: remapping LGWR worker group 0
```

```
OLRW: remapping slave: id 2(groupid 0, slaveid 2) -> id 2(groupid 2, slaveid 0)
```

```
OLRW: remapping slave: id 0(groupid 0, slaveid 0) -> id 0(groupid 0, slaveid 0)
```

```
OLRW: remapping slave: id 1(groupid 0, slaveid 1) -> id 1(groupid 1, slaveid 0)
```

```
OLRW: remapping slave: id 3(groupid 0, slaveid 3) -> id 3(groupid 3, slaveid 0)
```

```
OLRW: remapping LGWR worker group 1
```

```
OLRW: remapping slave: id 4(groupid 1, slaveid 0) -> id 4(groupid 4, slaveid 0)
```

```
OLRW: remapping slave: id 5(groupid 1, slaveid 1) -> id 5(groupid 5, slaveid 0)
```

```
OLRW: remapping slave: id 6(groupid 1, slaveid 2) -> id 6(groupid 6, slaveid 0)
```

```
OLRW: remapping slave: id 7(groupid 1, slaveid 3) -> id 7(groupid 7, slaveid 0)
```

```
OLRW: slave (groupid 2, slaveid 0) remap wait complete
```

```
OLRW: slave (groupid 1, slaveid 0) remap wait complete
```

```
OLRW: slave (groupid 3, slaveid 0) remap wait complete
```

```
OLRW: slave (groupid 0, slaveid 0) remap wait complete
```

```
OLRW: slave (groupid 6, slaveid 0) remap wait complete
```

```
OLRW: slave (groupid 7, slaveid 0) remap wait complete
```

```
OLRW: slave (groupid 4, slaveid 0) remap wait complete
```

```
OLRW: slave (groupid 5, slaveid 0) remap wait complete
```

```
OLRW: Starting of new slaves not needed, numgroups 8, optimal_numgroups 8
```

```
OLRW: Remap successful, num_groups 8, slaves_per_group 1
```

These messages will appear
in the alert log with event
10468 set at level 32768
(0x8000) at instance startup.



OLRW Worker Recomputation Example: CPU_COUNT=384

OLRW: Remapping 6 to 4 slaves_per_group

```
OLRW: remap_existing: num_groups_pre_remap = 2, interim_numslave = 12, interim_numgroups_pre_remap = 2,  
interim_numgroups_post_remap = 3, current_group_size = 6, target_group_size = 4
```

```
OLRW: remapping LGWR worker group 0
```

```
OLRW: remapping slave: id 1(groupid 0, slaveid 1) -> id 1(groupid 0, slaveid 1)
```

```
OLRW: remapping slave: id 0(groupid 0, slaveid 0) -> id 0(groupid 0, slaveid 0)
```

```
OLRW: remapping slave: id 2(groupid 0, slaveid 2) -> id 2(groupid 0, slaveid 2)
```

```
OLRW: remapping slave: id 5(groupid 0, slaveid 5) -> id 5(groupid 1, slaveid 1)
```

```
OLRW: remapping slave: id 3(groupid 0, slaveid 3) -> id 3(groupid 0, slaveid 3)
```

```
OLRW: remapping slave: id 4(groupid 0, slaveid 4) -> id 4(groupid 1, slaveid 0)
```

```
OLRW: remapping LGWR worker group 1
```

```
...
```

```
OLRW: slave (groupid 1, slaveid 1) remap wait complete
```

OLRW: Adjusting numgroups, numgroups = 3, optimal numgroups = 16

```
LGWR slave LG12 created with pid=167, OS pid=25960
```

```
LGWR slave LG13 created with pid=166, OS pid=25979
```

```
LGWR slave LG14 created with pid=169, OS pid=26002
```

```
...
```

```
LGWR slave LG63 created with pid=216, OS pid=26554
```

If needed, Oracle starts additional LG workers during the remap phase at instance startup.

OLRW: Remap successful, num_groups 16, slaves_per_group 4



OLRW LG Groups and Workers Recomputation Rules

OLRW recomputes the number of LG groups and LG workers per group based on the CPU_COUNT:

Number of LG groups

```
_max_outstanding_log_writes:  
  
< 48      : 2  
48-95     : 8  
96-191    : (CPU_COUNT + 8) / 12  
>= 192    : 16
```

Number of LG workers per Group

```
_max_log_write_parallelism:  
  
< 72      : 1  
72-191    : 2  
>= 192    : (CPU_COUNT + 48) / 96
```

Notes:

- OLRW recomputes the number of LG groups and workers per group at instance startup in `kcrfw_olrw_compute_optnumslaves`
- OLRW also remaps existing LG workers to different groups at instance startup in `kcrfw_olrw_init_remap` (s. also next slide)



Pipelined Log Writes & Other Features

Pipelined Log Writes & Data Guard

Data Guard with **LogXptMode SYNC** disables Adaptive Scalable LGWR and Pipelined Log Writes

Technically, disabling means putting the LG workers into standby mode by setting a "standby mode" flag in `kcrfw_slave_pool_setstandbymode`

```
# ./kcrfw_slave_pool_setstandbymode.bt $ORACLE_SID

341646 ora_lgwr_mysid -> kcrfw_slave_pool_setstandbymode: 0x1
341646 ora_lgwr_mysid   lgwr_mode : beg = 0x1, end = 0x0
341646 ora_lgwr_mysid   stdby_mode: beg = 0x0, end = 0x1
341646 ora_lgwr_mysid <- kcrfw_slave_pool_setstandbymode
```



Additional Information & Further Details

- [Appendix A: OLRW Automatic Parameter Adjustments](#)
- [Appendix B: Phase State Transitions \(Detailed\)](#)
- [Appendix C: Log Buffer Write Thresholds](#)
- [Appendix D: LGWR NUMA Awareness](#)
- [Appendix E: ALWE and OLRW Sysstat Counters](#)
- [Appendix F: Tracing Scripts](#)
- [Appendix G: LGWR Trace Event 10468](#)
- [Appendix H: struct alwe](#)
- [Appendix I: struct olrw](#)



Thank you for attending!



SYMPOSIUM ^{L2}



Thank you for attending!

Questions, feedback, comments?
I look forward to hearing from you!



<https://www.0xChris.dev>



christoph.lutz@swisscom.com



[@chris_skyflier](https://twitter.com/chris_skyflier)



[@christophlutz.bsky.social](https://bsky.social/@christophlutz)



[Christoph-Lutz](https://github.com/Christoph-Lutz)



References



References (1/2)

[Alex Blyth et. al., Oracle Database 23ai now available for Exadata, 2024-07-22](#)

[Chi Cao Minh et al., United States Patent 9418129B2, Adaptive High Performance Database Redo Log Synchronization, 2016-08-16](#)

[Chi Cao Minh et al., United States Patent 10706009B2, Techniques to parallelize CPU and IO work of log writes, 2020-07-07](#)

[Christoph Lutz, oracle Github Repository, July 2025](#)

[Christian Craft, Persistent Memory Primer, 2020-05-20](#)

[Christian Craft, Persistent Memory in Exadata X8M, 2020-05-21](#)

[Frits Hoogland, A look into Oracle redo, part 4: the log writer null write, 2018-02-20](#)

[Graham Ivy et al., United States Patent 12346311B2, Adaptively overlapping the writing of redo log records, 2025-07-01](#)

[My Oracle Support, Bug 14823372 - Adaptive "log file sync" picks inaccurate polling interval on RAC \(Doc ID 14823372.8\), 2022-03-13](#)

[Nikolay Savvinov, High log file sync waits? Check log parallelism!, 2014-19-22](#)



References (2/2)

[Oracle Corp., Database 19c, Database Concepts, E96138-09, July 2025](#)

[Oracle Corp., Oracle® Exadata System Software, User's Guide 25.2, F29254-45, July 2025](#)

[Oracle Corp., Oracle 9i Database Reference Release 2 \(9.2\), Parameter LOG PARALLELISM, A96536-02](#)

[Oracle Corp., Oracle 19c Database Reference, E96196-38, July 2025](#)

[Tanel Poder, Understanding LGWR, Log File Sync Waits, and Commit Performance \(via Scribd\), undated](#)

[Tanel Poder, Testing Oracle's Use of Optane Persistent Memory, Part 1 - Low Latency Commits, 2022-04-28](#)

[Tanel Poder, Testing Oracle's Use of Optane Persistent Memory, Part 2 - Fast Log File Sync Waits, 2022-07-15](#)

[Tanel Poder, Oracle Performance Troubleshooting Without OS Access, Part 1: Identifying CPU Scheduling Latency, 2020-02-10](#)

[Wei Ming Hu et al., United States Patent 7039773B2, Method and Mechanism for Efficient Implementation of Ordered Records, 2006-05-02](#)

[Yuhong Zhong, XRP: In-Kernel Storage Functions with eBPF \(Youtube Video eBPF Summit 2022\), 2022-10-04](#)



Appendix A: OLRW Automatic Parameter Adjustments



Pipelined Log Writes – Automatic Adjustments at Instance Startup

- When the instance starts, Oracle performs various sanity checks and, in some situations, also automatically adjusts the configured thin and thick mode parameter values defined by `_lgwr_thin_mode_config` and `_lgwr_thick_mode_config`.
- The checks and adjustments are performed in `kcrfw_olrw_init_params` and cannot be overridden, as they are hardcoded.
- The automatic adjustments can be observed with `bpfttrace` or by enabling trace event 10468 at level 0x8400 (0x8000 + 0x400).

Note that you must enable trace event 10468 in the spfile for tracing ALWE and OLRW:

```
alter system 10468 trace name context forever, level 0x8400' scope=spfile;
```



Pipelined Log Writes – Thin Mode: Start of Wait Zone Adjustment

```
/*
 * If wait_zone_start equals or exceeds
 * max_outstanding_log_writes, it is ad-
 * justed to max_outstanding_lg_writes - 1
 */

$wait_zone_start = ($max_out_log_wr - 1);

if (@thin_cfg_p1[pid] >= $max_out_log_wr) {
    printf("%ld %s      OLRW: Start of wait zone in thin mode\n", pid, comm);
    printf("%ld %s          adjusted from %u to %u\n", pid, comm, @thin_cfg_p1[pid],
          $wait_zone_start);
}
```



Pipelined Log Writes – Thin Mode: Min Write Size Adjustment

```
/*
 * The min write size is set to
 * the smaller of the two min and
 * max size limits, but never less
 * than 8.
 */

$write_sz = (@thin_cfg_p3[pid] < @thin_cfg_p2[pid]) ? @thin_cfg_p3[pid] : @thin_cfg_p2[pid];

$wait_min_sz = ($write_sz > 8) ? $write_sz : 8;

if ($wait_min_sz != @thin_cfg_p2[pid]) {
    printf("%ld %s      OLRW: Minimum write size for wait zone in thin mode adjusted\n", pid, comm);
    printf("%ld %s          from %u to %u\n", pid, comm, @thin_cfg_p2[pid], $wait_min_sz);
}
```



Pipelined Log Writes – Thin Mode: Max Write Size Adjustment

```
/*
 * Enforce a max write size of at
 * least 8, but only when the write_sz
 * is larger than or equal to the con-
 * figured size and the configured size
 * is currently below 8.
 */

$write_sz = (KSFD_MAXIO / REDO_BLOCK_SZ * $max_strands);

$wait_max_sz = (($write_sz >= @thin_cfg_p3[pid]) && (@thin_cfg_p3[pid] < 8)) ? 8 : @thin_cfg_p3[pid];

if ($wait_max_sz != @thin_cfg_p3[pid]) {
    printf("%ld %s      OLRW: Maximum write size for the wait zone\n", pid, comm);
    printf("%ld %s      in thin mode adjusted from %u to %u\n", pid, comm,
           @thin_cfg_p3[pid], $wait_max_sz);
}
```



Pipelined Log Writes – Thin Mode: Min Timeout Adjustment

```
/*  
 * The min timeout is the smaller  
 * of the two min and max timeout  
 * limits.  
 */  
  
if (@thin_cfg_p5[pid] < @thin_cfg_p4[pid]) {  
    $tmo_min_usec = @thin_cfg_p5[pid];  
    printf("%ld %s      OLRW: Minimum timeout for wait zone in thin mode adjusted\n", pid, comm);  
    printf("%ld %s      from %u to %u\n", pid, comm, @thin_cfg_p4[pid], $tmo_min_usec);  
}
```



Pipelined Log Writes – Thick Mode: Overflow Zone Adjustment

```
/*
 * Set the overflow_zone to a
 * minimum value of 1, if it's
 * set to 0.
 * If the overflow zone is greater
 * than or equal to _max_outstanding_
 * log_writes, set it to:
 * max_outstanding_log_writes - 1.
 */

if (@thick_cfg_p2[pid] == 0) {
    $overflow_zone = 1;
    printf("%ld %s      OLRW: Start of overflow zone\n", pid, comm);
    printf("%ld %s          adjusted from %u to %u\n", pid, comm,
          @thick_cfg_p2[pid], $overflow_zone);
}
else {
    if ($max_out_log_wr <= @thick_cfg_p2[pid]) {
        $overflow_zone = $max_out_log_wr - 1;
        printf("%ld %s      OLRW: Start of overflow zone\n", pid, comm);
        printf("%ld %s          adjusted from %u to %u\n", pid, comm,
              @thick_cfg_p2[pid], $overflow_zone);
    }
}
}
```



Pipelined Log Writes – Thick Mode: Fallback Zone Adjustment

```
/*
 * If the overflow zone is less than
 * or equal to the end of the fallback
 * zone, set the fallback zone to:
 * overflow_zone - 1
 */

if (@thick_cfg_p2[pid] <= @thick_cfg_p1[pid]) {
    $fallback_zone = $overflow_zone - 1;
    printf("%ld %s      OLRW: Start of fallback zone\n", pid, comm);
    printf("%ld %s      adjusted from %u to %u\n", pid, comm, @thick_cfg_p1[pid], $fallback_zone);
}
```



Pipelined Log Writes – Thick Mode: Max Write Size Adjustment

```
/*
 * If the max write size is less than
 * or equal to the total size of all
 * strands, use the configured value.
 * However, if the configured value is
 * less than 256, set it to 256 instead.
 */

$thick_max_sz = (KSFD_MAXIO / REDO_BLOCK_SZ * $max_strands);

if (@thick_cfg_p4[pid] <= $thick_max_sz) {
    $thick_max_sz = (@thick_cfg_p4[pid] < 256) ? 256 : @thick_cfg_p4[pid];
}

if($thick_max_sz != @thick_cfg_p4[pid]) {
    printf("%ld %s      OLRW: Maximum write size for all zones\n", pid, comm);
    printf("%ld %s      in thick mode adjusted from %u to %u\n", pid, comm,
           @thick_cfg_p4[pid], $thick_max_sz);
}
```



Pipelined Log Writes – Thick Mode: Min Write Size Adjustment

```
/*
 * The adjustment is calculated according to the (rather complex) formula below,
 * if the parameter _lgwr_thick_mode_config is not set at session level (checked
 * by kspges).
 * If _lgwr_thick_mode_config is set at session level, use the smaller of the
 * min and max write sizes, but never less than 256.
 */

if (@lgwr_thick_mode_cfg_prm[pid] == 0) {
    $thick_min_sz_calc = 1;
    $thick_min_sz      = ($max_out_log_wr - $olrw->wait_zone_start) * 2 *
                        ($olrw->wait_max_sz + $olrw->wait_min_sz) /
                        ($olrw->fallback_zone_end * 5 + 5);
} else {
    $thick_min_sz_calc = 0;
    $thick_min_sz      = (@thick_cfg_p4[pid] < @thick_cfg_p3[pid]) ? @thick_cfg_p4[pid] : @thick_cfg_p3[pid];
    $thick_min_sz      = ($thick_min_sz <= 256) ? 256 : $thick_min_sz;
}

if ($thick_min_sz != @thick_cfg_p3[pid]) {
    printf("%ld %s      OLRW: Minimum %starget size for all zones\n", pid, comm,
           ($thick_min_sz_calc ? "calculated " : ""));
    printf("%ld %s      in thick mode adjusted from %u to %u\n", pid, comm,
           @thick_cfg_p3[pid], $thick_min_sz);
}
```



Pipelined Log Writes – Thick Mode: Max Write Size Adjustment

```
/*
 * If the max write size is less than
 * or equal to the total size of all
 * strands, use the configured value.
 * However, if the configured value is
 * less than 256, set it to 256 instead.
 */

$thick_max_sz = (KSFD_MAXIO / REDO_BLOCK_SZ * $max_strands);

if (@thick_cfg_p4[pid] <= $thick_max_sz) {
    $thick_max_sz = (@thick_cfg_p4[pid] < 256) ? 256 : @thick_cfg_p4[pid];
}

if($thick_max_sz != @thick_cfg_p4[pid]) {
    printf("%ld %s      OLRW: Maximum write size for all zones\n", pid, comm);
    printf("%ld %s          in thick mode adjusted from %u to %u\n", pid, comm,
           @thick_cfg_p4[pid], $thick_max_sz);
}
```



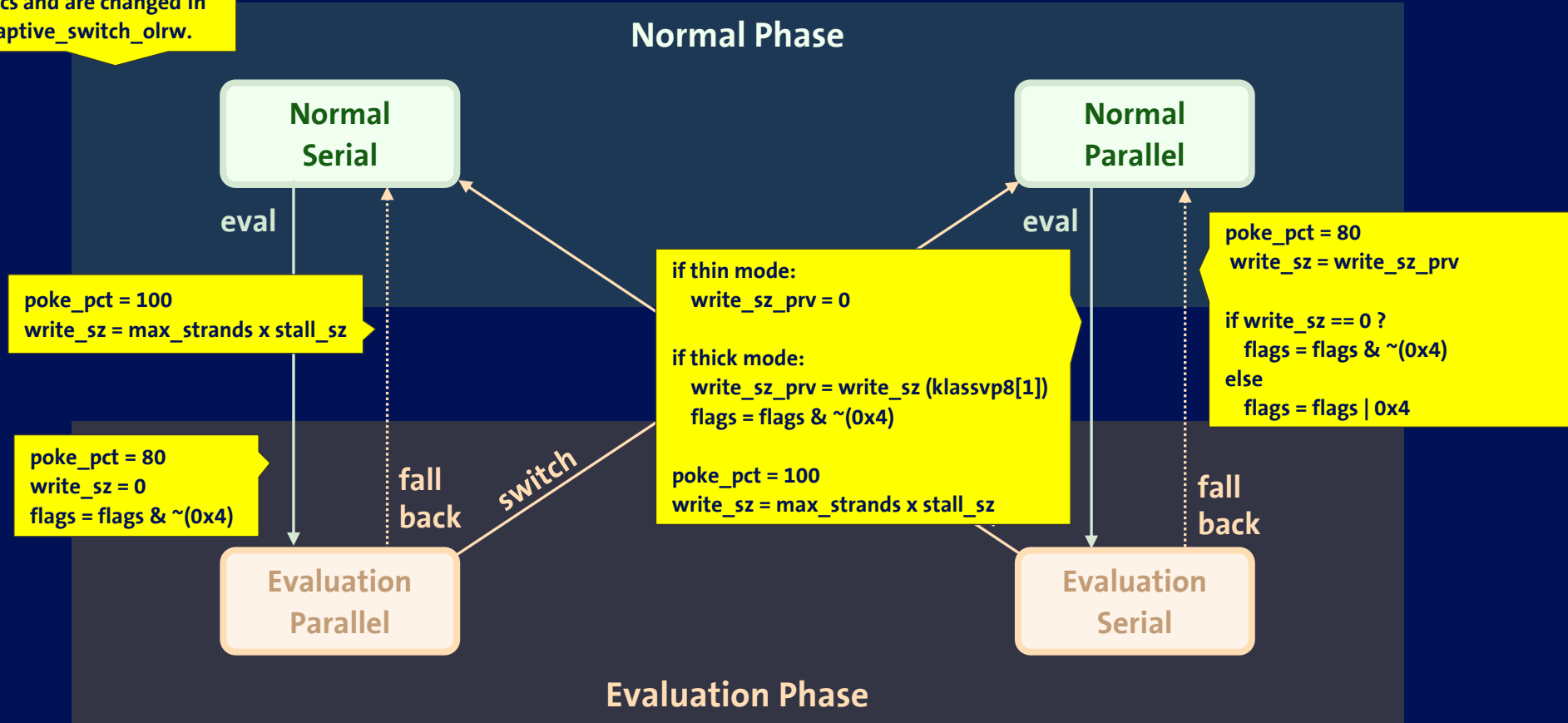
Appendix B: Phase State Transitions (Detailed)



Pipelined Log Writes – Normal & Evaluation Phase State Transitions (Detailed)

Note:

The OLRW metrics and are changed in kcrfw_slave_adaptive_switch_olrw.



Notes:

- Falling back from evaluation/serial to normal/parallel restores thick mode (and its previous `write_sz`) if thick mode was active before the transition
- During the evaluation/parallel phase, thick mode is switched off.
- In serial mode `poke_pct` is 100; in parallel mode, `poke_pct` is 80.
- In serial mode, `write_sz` is set to: `max_strands x stall_sz` (single active strand) and `max_strands x stall_sz / actv_strands` (with multiple active strands)
- In parallel mode, `write_sz` is recalculated dynamically at runtime, but the write threshold differs between thin and thick mode!



Pipelined Log Writes – Normal & Evaluation Phase State Transitions (Detailed)

Initial State	Target State	Write Size	Write Size Prev	Poke Pct	Write Threshold
Normal/Serial	Evaluation/Parallel (Eval.)	0	n/a	80	1
Evaluation/Parallel	Normal/Serial (Fallback)	$\text{max_strands} * \text{stall_sz}$	n/a	100	1 strand : $\text{write_sz} * \text{poke_pct}/100$ >1 strands: $\text{write_sz} * \text{poke_pct}/100 / \text{actv_strands}$
Evaluation/Parallel	Normal/Parallel (Switch)	(0)	(n/a)	(80)	(1)
Normal/Parallel	Evaluation/Serial (Eval.)	$\text{max_strands} * \text{stall_sz}$	thin: 0 thick: write_sz	100	1 strand : $\text{write_sz} * \text{poke_pct}/100$ >1 strands: $\text{write_sz} * \text{poke_pct}/100 / \text{actv_strands}$
Evaluation/Serial	Normal/Parallel (Fallback)	write_sz_prv	n/a	80	Thin : 1 Thick: $\text{write_sz} * \text{poke_pct}/100 / \text{actv_strands}$
Evaluation/Serial	Normal/Serial (Switch)	$(\text{max_strands} * \text{stall_sz})$	(n/a)	(100)	(1 strand : $\text{write_sz} * \text{poke_pct}/100$) (>1 strands: $\text{write_sz} * \text{poke_pct}/100 / \text{actv_strands}$)

Notes:

- `kcrfw_slave_adaptive_switch_olrw` changes the write size, the previous write size, and the poke pct when a transition from normal to evaluation phase occurs. However, in parallel mode, the `write_sz` is dynamically recomputed at runtime, but it only impacts the write threshold when thick mode is active (that is, in thin mode, the write threshold is fixed to 1)
- When a transition from normal/serial to evaluation/parallel occurs, parallel mode is set to thin mode; when a transition from evaluation/serial to normal/parallel occurs, thick mode and the previously used write size are restored when thick mode was active before the transition
- The write threshold is always computed in `kcrfw_redo_gen_ext`



Appendix C: Log Buffer Write Thresholds



Oracle Documentation – Redo Log Buffer "1/3 Full" Rule

Configuring the Redo Log Buffer

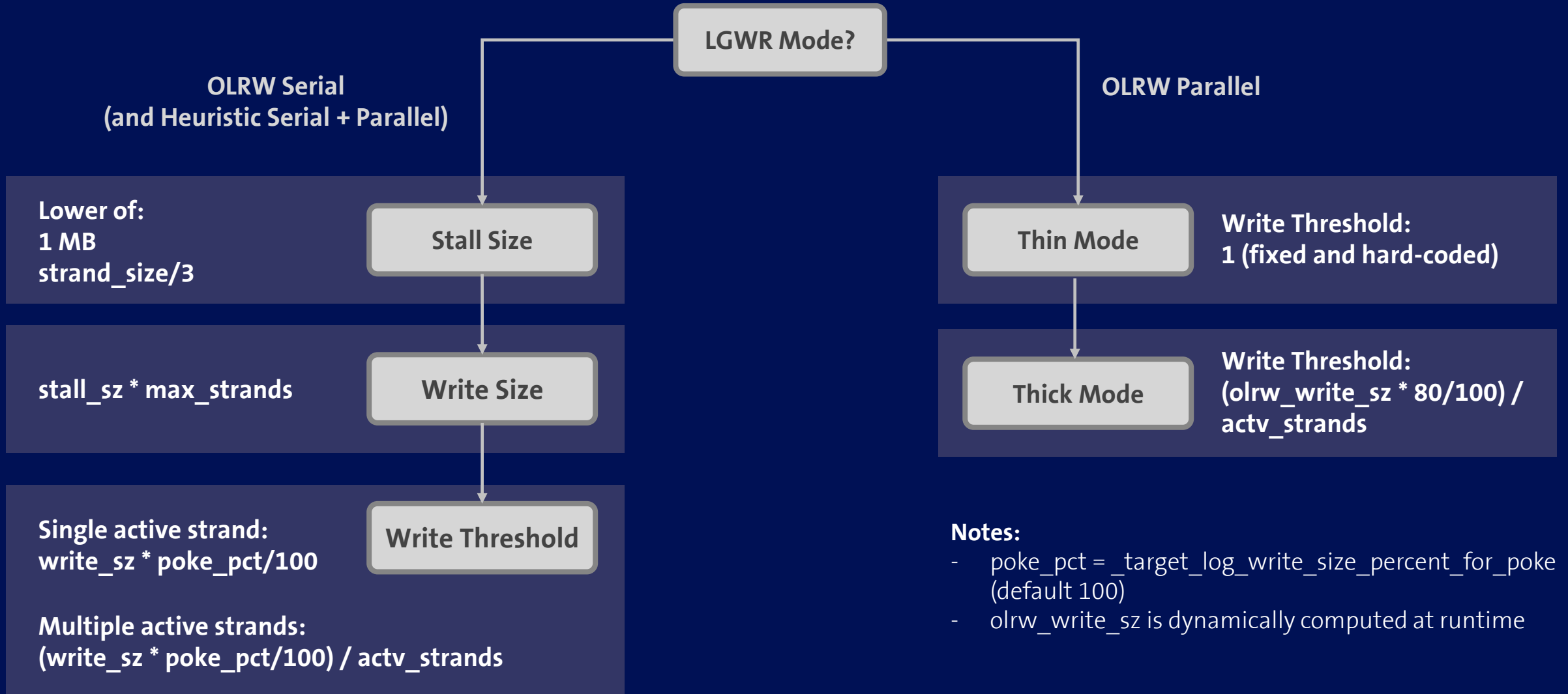
Server processes making changes to data blocks in the buffer cache generate redo data into the log buffer. The log writer process (LGWR) begins writing to copy entries from the redo log buffer to the online redo log if any of the following conditions are true:

- The redo log buffer becomes at least one-third full
- LGWR is posted by a server process performing a `COMMIT` or `ROLLBACK`
- A database writer process (DBWR) posts LGWR to do so





Redo Log Buffer Write Threshold





Pipelined Log Writes – Redo Log Buffer Write Threshold Examples

Single Mode / Single Active Strand



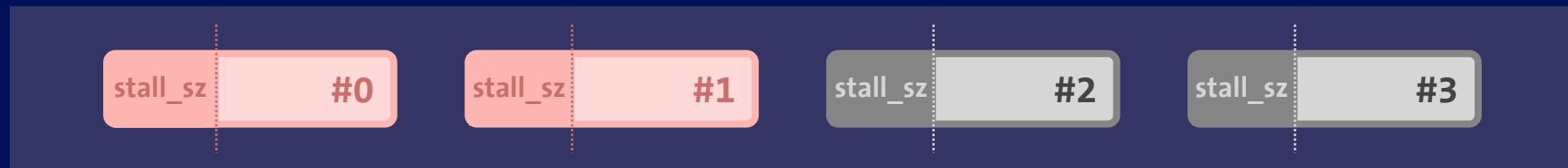
Write Size:

$$\text{stall_sz} * \text{max_strands} = 1 \text{ MB} * 4 = 4 \text{ MB}$$

Write Threshold:

$$\text{write_sz} * \text{poke_pct}/100 = 4 \text{ MB} * 100/100 = 4 \text{ MB}$$

Single Mode / Multiple Active Strands



Write Size:

$$\text{stall_sz} * \text{max_strands} = 1 \text{ MB} * 4 = 4 \text{ MB}$$

Write Threshold:

$$\text{write_sz} * \text{poke_pct}/100 / \text{actv_strands} = 4 \text{ MB} * 100/100 / 2 = 2 \text{ MB}$$



Pipelined Log Writes – Redo Log Buffer "1/3 Full" Rule

1. LGWR in serial mode
2. All strands active
3. Strand size < 1 MB

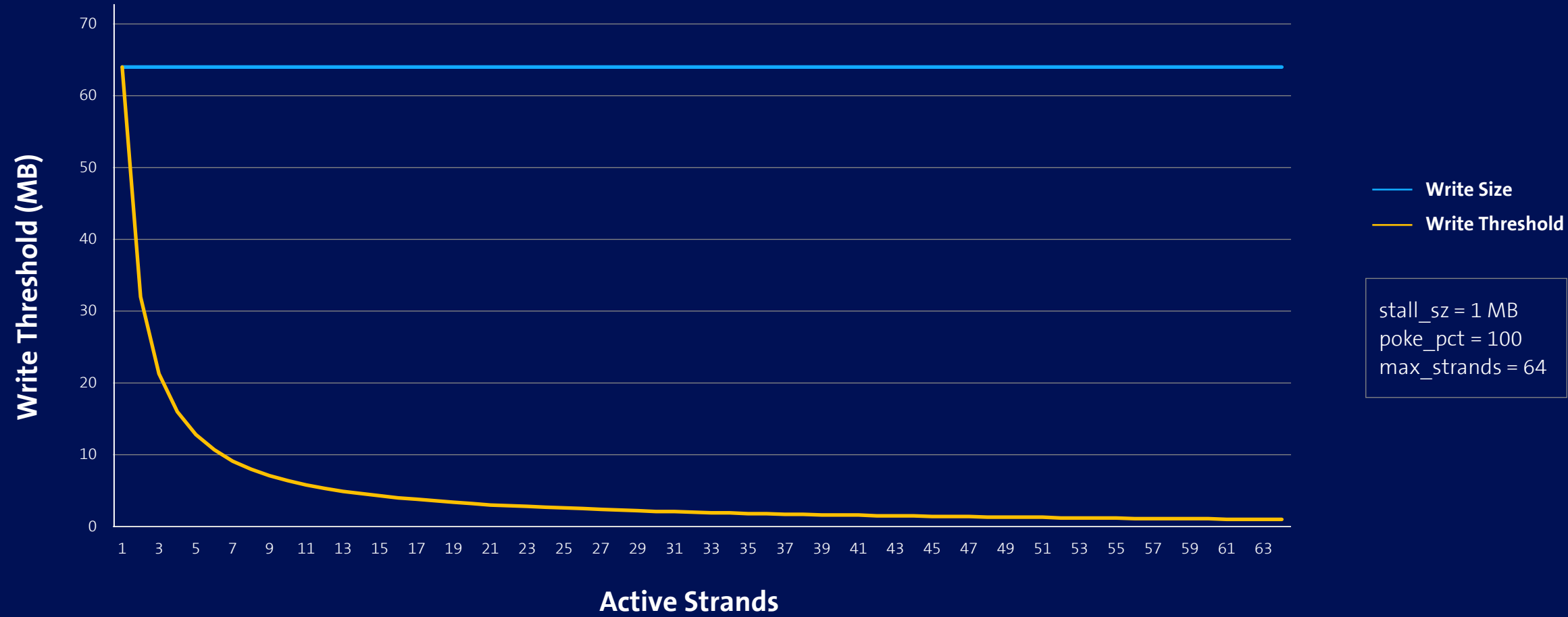
The log buffer "1/3 full" rule mentioned in the docs only applies when all these conditions are met!





Redo Log Buffer Write Threshold – OLRW Serial Mode (64 Strands)

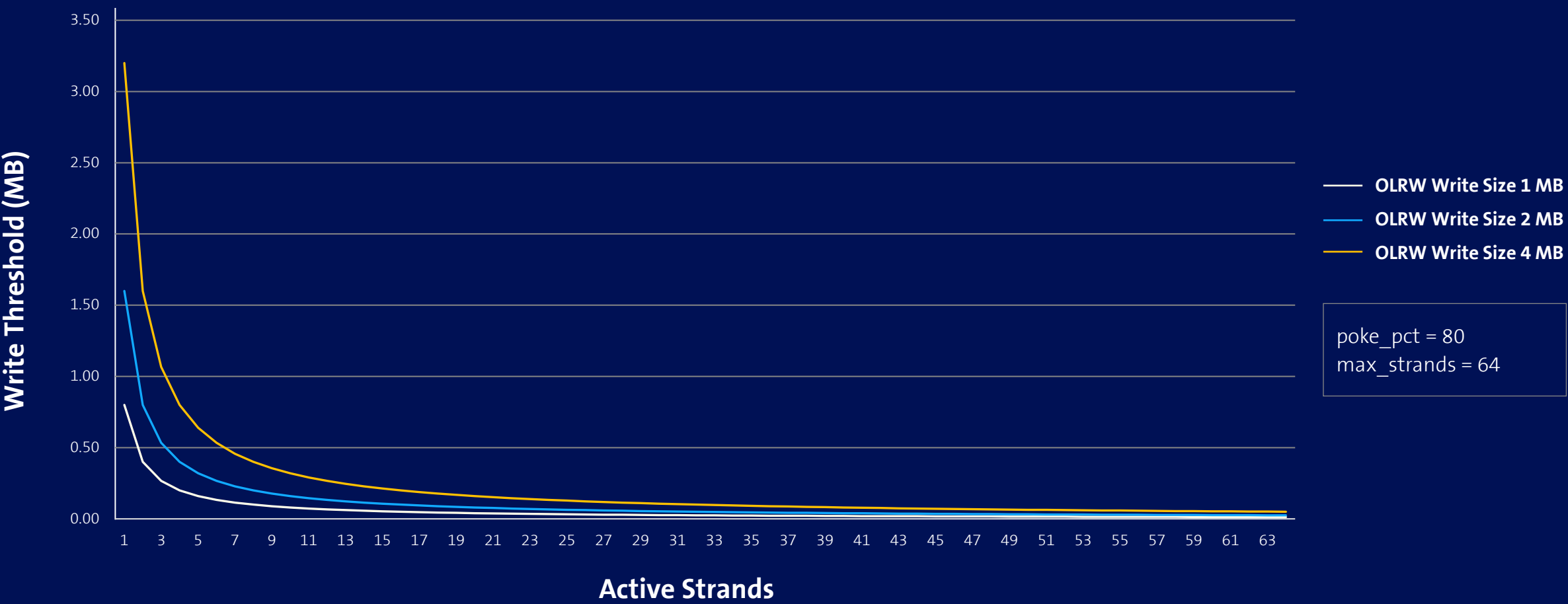
Write Threshold vs Active Strands





Redo Log Buffer Write Threshold – OLRW Parallel Thick Mode (64 Strands)

Write Threshold vs Active Strands





Log Buffer Write Threshold – Computation

LGWR

```
/* At instance startup */
kcrfw_olrw_init_params()
    poke_pct = _target_log_write_size_percent_for_poke; /* default = 100 */

    stall_sz = least(1048576/redo_block_sz, log_buffer/kcrf_max_strands_/redo_block_sz/3);
    write_sz = kcrf_max_strands_ * stall_sz;

/* At lgwr mode switch */
kcrfw_slave_adaptive_switch_olrw()
    poke_pct = lgwr_serial_mode ? poke_pct : 80; /* value of 80 is hard-coded */

/* Before every log write */
kcrfw_gather_lwn()

    if single strand active: wr_thresh = write_sz * poke_pct/100;

    if multiple strands active: wr_thresh = (write_sz * poke_pct/100) / actv_strands;

    kcrfa->start_wr_thresh_kcrfa_cln = wr_thresh;
```

FG

```
/* Redo change vector creation */
kcrfw_redo_gen_ext()
    kcrfa->start_wr_thresh_kcrfa_cln -= alloc_redo_buffers;

    if kcrfa->start_wr_thresh_kcrfa_cln < 1 ?
        ksbasend() /* post lgwr */
```



Appendix D: LGWR NUMA Awareness



LG Workers – NUMA Awareness

```
$ ps -aef | grep -i $ORACLE_SID |  
grep 'lg[0-9]\{2\}' |  
awk '{print $2}' |  
xargs -I{} grep -i current_node /proc/{}/sched |  
sort | uniq -c
```

```
32 current_node=0, numa_group_id=0
```

```
32 current_node=1, numa_group_id=1
```

Oracle evenly distributes LG worker processes across different NUMA nodes





Exadata Socket Binding Service – exadata-dbproc-bind.service

```
# systemctl status exadata-dbproc-bind.service
```

```
- exadata-dbproc-bind.service - Oracle Exadata DB process socket-binding service
  Loaded: loaded (/etc/systemd/system/exadata-dbproc-bind.service; enabled; vendor preset: disabled)
  Active: active (running) since Tue 2025-10-28 09:13:56 CET; 5 days ago
Main PID: 24018 (exadata-dbproc-)
  Tasks: 1 (limit: 319999)
Memory: 10.5M (limit: 500.0M)
  CGroup: /system.slice/exadata-dbproc-bind.service
          └─24018 /bin/python3 /etc/exadata/systemd/exadata-dbproc-bind
```

```
Nov 01 11:22:30 exadb01 exadata-dbproc-bind[24018]: ..... ++++++
Nov 01 11:22:30 exadb01 exadata-dbproc-bind[24018]: new process of interest: ora_lgwr_mysid[95803:95803] (bound to node 1)
                                                cpuset: 96-98,104-106,112-114,120-122,128-130,136-138,144-146,152-[...]
Nov 01 11:22:30 exadb01 exadata-dbproc-bind[24018]: ..... ++++++
```



Appendix E: ALWE and OLRW Sysstat Counters



Pipelined Log Writes – ALWE Sysstat Counters

redo writes adaptive all

All redo writes in serial and parallel mode (tracked by LGWR).

redo writes adaptive worker

Redo writes in parallel mode (tracked by LGWR).

Notes:

- The counters are only incremented during the normal phase; that is, evaluative writes are not tracked.
- To calculate the number of serial writes: redo writes adaptive all - redo writes adaptive worker



Pipelined Log Writes – OLRW Sysstat Counters (1/2)

adaptive overlapped writes thin mode (all)

Total number of redo writes in thin mode (tracked by LGWR).

adaptive overlapped writes thin mode (size)

Number of redo write in thin mode that reach the target write size (tracked by LGWR).

adaptive overlapped writes thick mode (all)

Total number of redo writes in thick mode (tracked by LGWR).

adaptive overlapped writes thick mode (size)

Number of redo writes in thick mod that reach the target write size (tracked by LGWR).

Notes:

- The "size" counters are only incremented when the write size target threshold is reached.
- To calculate the number of redo writes that reached the timeout threshold:
adaptive overlapped writes thin mode (all) - adaptive overlapped writes thin mode (size)



Pipelined Log Writes – OLRW Sysstat Counters (2/2)

adaptive overlapped writes thin mode (all blks written)

Total number of redo blocks written in thin mode (tracked by the LG workers).

adaptive overlapped writes thick mode (all blks written)

Total number of redo blocks written in thick mode (tracked by the LG workers).



Appendix F: Tracing Scripts



Pipelined Log Writes – ALWE Observability (bpftrace)

```
# ./kcrfw_slave_adaptive_check_phase.bt $ORACLE_SID

155107 ora_lgwr_tctest -> kcrfw_slave_adaptive_check_phase
155107 ora_lgwr_tctest      fast_sync = 0
155107 ora_lgwr_tctest      kcrf_slave_info = 3
155107 ora_lgwr_tctest      lgwr_mode = 0
155107 ora_lgwr_tctest      lgwr_stdby_mode = 0
155107 ora_lgwr_tctest      max_outstanding_log_writes = 8
...
155107 ora_lgwr_tctest      ALWE: Normal phase using serial (848 writes, 848 o/l writes, 0 anomalies) finished:
155107 ora_lgwr_tctest      normal=5566, lfs factor=1, adj. normal=5566
155107 ora_lgwr_tctest      ALWE: Normal phase: tot_blocks=1681, tot_time=302008, avg size=1
155107 ora_lgwr_tctest      ALWE: evaluation of parallel (16 blocks) started
...
155107 ora_lgwr_tctest      alwe stats
155107 ora_lgwr_tctest      blocks_per_sec      5605      5566      4294967257
155107 ora_lgwr_tctest      normal_writes      848      848      0
155107 ora_lgwr_tctest      eval_writes      0      -8      -8
155107 ora_lgwr_tctest      blocks      0      16      16
155107 ora_lgwr_tctest      blocks_fraction      0      16      16
155107 ora_lgwr_tctest      anomalies      0      0      0
155107 ora_lgwr_tctest      ol_writes      848      0      -848
155107 ora_lgwr_tctest      tot_time      302008      0      -302008
155107 ora_lgwr_tctest      tot_blocks      1681      0      -1681
155107 ora_lgwr_tctest      ow_blocks      0      0      0
...
155107 ora_lgwr_tctest <- kcrfw_slave_adaptive_check_phase
```



Pipelined Log Writes – ALWE/OLRW Observability (bpftrace)

```
# ./kcrfw_defer_write-thresholds.bt $ORACLE_SID
```

TIME	PHASE	LGWR_MODE	OLRW_MODE	ARBITER	HI_BSY_GRP	WRITE_SZ	W_DIFF	W_TGT?	TIMEOUT	T_DIFF	POKE_PCT	MAX_STRANDS	ACT_STRANDS	WR_THRESH
2025-12-06 11:16:57.700114	normal	parallel	thin	0	-1	0	0	no	0	0	80	16	1	1
2025-12-06 11:16:58.700334	normal	parallel	thin	0	-1	0	0	no	0	0	80	16	1	1
2025-12-06 11:16:58.700620	normal	parallel	thin	0	-1	0	0	no	0	0	80	16	1	1
2025-12-06 11:16:58.701267	normal	parallel	thin	0	-1	0	0	no	0	0	80	16	1	1
2025-12-06 11:17:02.028300	normal	parallel	thin	1	7	256	0	no	748	748	80	16	1	1
2025-12-06 11:17:02.030461	normal	parallel	thin	2	7	256	0	no	748	0	80	16	1	1
2025-12-06 11:17:02.033810	normal	parallel	thick	0	7	1382	1382	no	1500	0	80	16	1	1105
2025-12-06 11:17:03.035067	normal	parallel	thick	0	7	1446	64	no	1500	0	80	16	1	1156
2025-12-06 11:17:03.038611	normal	parallel	thick	0	7	1510	64	no	1500	0	80	16	1	1208
2025-12-06 11:17:03.042461	normal	parallel	thick	0	7	1574	64	no	1500	0	80	16	1	1259
2025-12-06 11:17:04.043188	normal	parallel	thick	0	7	1638	64	no	1500	0	80	16	1	1310
2025-12-06 11:17:04.046709	normal	parallel	thick	0	7	1702	64	no	1500	0	80	16	1	1361



Pipelined Log Writes – OLRW Simulate hi_busy_grp_num (gdb)

```
# gdb -q -ex 'set $p_hi_busy_grp_num=7' -x ./olrw-sim-hi_busy_grp_num.gdb -p <lgwr_pid>

alwe->flags      = 0x6
olrw->flags      = 0x11b
lgwr_mode       = 1
max_log_wri_par = 8
olrw trace disabled, enabling it
olrw->flags = 0x11b

Breakpoint 1 at 0x13848e62
Breakpoint 2 at 0xf928c52
Breakpoint 3 at 0xf928d4d

Breakpoint 1, 0x0000000013848e62 in kcrfw_defer_write ()
-> kcrfw_defer_write

Breakpoint 2, 0x00000000f928c52 in kcrfw_slave_queue_hi_busy_group ()
-> kcrfw_slave_queue_hi_busy_goup

Breakpoint 3, 0x00000000f928d4d in kcrfw_slave_queue_hi_busy_group ()
    Changing hi_busy_grp_num to: 7
    Changed hi_busy_grp_num to : 7
<- kcrfw_slave_queue_hi_busy_goup
```



Log Buffer Write Threshold – Serial LGWR / Single Strand Active

```
# gdb -q -x ./kcrfw_redo_gen_ext-poke.gdb -p 34534

-> kcrfw_redo_gen_ext: 0x2, 0x7ffe99f19f38, 0x7ffe99f19ef8
max_strands = 16, actv_strands = 1, write_sz = 32768, wr_thresh_calc = 32768

      strand: #0          start      end      diff
      tot_bufs_kcrfa      12288     12288      0
      memory_bufs_available_kcrfa_cln      7      7      0
      start_wr_thresh_kcrfa_cln      20489     20489      0
<- kcrfw_redo_gen_ext: 1

-> kcrfw_redo_gen_ext: 0x2, 0x7ffe99f19f38, 0x7ffe99f19ef8
...
<- kcohrh: redo_sz = 7652, redo_blocks = 14

-> kslgetl: laddr=0x4168cf8e8, wait=0, why=0, where=3893
acquired RAL: 0x4168cf8e8, strand: #0
<- kslgetl: 1

-> ksbasend: tgt_paddr=0x60021628, msg_idx=191, sz=0x20
dest = LGWR [Redo writer post action] caller_ip = 0x1383f21d

-> kskthbwt: ev#=166
start wait: log buffer space
-> kskthewt: ev#=166
end wait: log buffer space

      strand: #0          start      end      diff
      tot_bufs_kcrfa      12288     12288      0
      memory_bufs_available_kcrfa_cln      12286     12270     -16
      start_wr_thresh_kcrfa_cln      32768     32752     -16
<- kcrfw_redo_gen_ext: 1
```

The "start write threshold" is larger than the total number of redo buffers in strand #0.

The strand is almost full (only 7 buffers left), but the write threshold is still not exceeded!

The next allocation needs more than 7 buffers, but the strand is full ...

... so the session signals lgwr to start writing ...

... and then starts a "log buffer space" wait.

Lgwr completes writing, frees the buffers and resets the write threshold.



Log Buffer Write Threshold – Serial LGWR / All Strands Active

```
# gdb -q -ex 'set $min_actv_strands=16' -x ./kcrfw_redo_gen_ext-poke.gdb -p 34534
```

Changing the number of active strands to: 16

```
-> kcrfw_redo_gen_ext: 0x2, 0x7ffe7f9aefb8, 0x7ffe7f9aef78
max_strands = 16, actv_strands = 16, write_sz = 32768, wr_thresh_calc = 2048
```

strand: #5	start	end	diff
tot_bufs_kcrfa	12288	12288	0
memory_bufs_available_kcrfa_cln	10250	10247	-3
start_wr_thresh_kcrfa_cln	12	9	-3

```
<- kcrfw_redo_gen_ext: 1
```

```
-> kcrfw_redo_gen_ext: 0x2, 0x7ffe7f9aefb8, 0x7ffe7f9aef78
```

...

```
<- kcohrh: redo_sz = 6520, redo_blocks = 12
```

```
-> kslgetl: laddr=0x4168cfc08, wait=0, why=5, where=3893
```

```
acquired RAL: 0x4168cfc08, strand: #5
```

```
<- kslgetl: 1
```

```
-> ksbasend: tgt_paddr=0x60021628, msg_idx=191, sz=0x20
```

```
dest = LGWR [Redo writer post action] caller_ip = 0x1383e69a
```

strand: #5	start	end	diff
tot_bufs_kcrfa	12288	12288	0
memory_bufs_available_kcrfa_cln	10247	10234	-13
start_wr_thresh_kcrfa_cln	9	-4	-13

```
<- kcrfw_redo_gen_ext: 1
```

With 16 active strands, the "start write threshold" is 1 MB (2048 redo blocks of 512 bytes).

Calculation:

$32768 * 100/100 / 16 = 2048$

The write threshold drops to 9 buffers ...

... but the next allocation needs 12 buffers ...

... so the session "stalls" and signals lgwr to start writing.

While lgwr is writing, the session continues with further allocations, and the write threshold goes negative.



Log Buffer Write Threshold – Parallel LGWR / Thin Mode / Single Strand Active

```
# gdb -q -x ./kcrfw_redo_gen_ext-poke.gdb -p 34534

-> kcrfw_redo_gen_ext: 0x2, 0x7ffd7a86c4c8, 0x7ffd7a86c488
max_strands = 16, actv_strands = 1, write_sz = 0, wr_thresh_calc = 0
...
-> ksbasend: tgt_paddr=0x60021628, msg_idx=191, sz=0x20
dest = LGWR [Redo writer post action] caller_ip = 0x1383e69a

strand: #0
tot_bufs_kcrfa      start      end      diff
memory_bufs_available_kcrfa_cln 12286  12283  -3
start_wr_thresh_kcrfa_cln      1      1      0
<- kcrfw_redo_gen_ext: 1

-> kcrfw_redo_gen_ext: 0x2, 0x7ffd7a86cf68, 0x7ffd7a86cf28
max_strands = 16, actv_strands = 1, wr_thresh_calc = 0
...
-> ksbasend: tgt_paddr=0x60021628, msg_idx=191, sz=0x20
dest = LGWR [Redo writer post action] caller_ip = 0x1383e69a

strand: #0
tot_bufs_kcrfa      start      end      diff
memory_bufs_available_kcrfa_cln 12286  12285  -1
start_wr_thresh_kcrfa_cln      1      1      0
<- kcrfw_redo_gen_ext: 1
```

The calculated "start write threshold" is 0, and Oracle internally adjusts this to 1.

Due to the threshold of 1, the session signals lgwr to start writing after every redo buffer allocation.

The pattern repeats for every write in thin mode.



Log Buffer Write Threshold – Parallel LGWR / Thick Mode / Single Strand Active

```
# gdb -q -x ./kcrfw_redo_gen_ext-poke.gdb -p 34534
-> kcrfw_redo_gen_ext: 0x1, 0x7ffe077aaab8, 0x7ffe077aaa78
    max_strands = 16, actv_strands = 1, write_sz = 4582, wr_thresh_calc = 3665

<- kcohrh: redo_sz = 132, redo_blocks = 0

strand: #0
  tot_bufs_kcrfa      start      end      diff
  tot_bufs_kcrfa      12288     12288      0
  memory_bufs_available_kcrfa_cln  12286     12285     -1
  start_wr_thresh_kcrfa_cln        3665     3664     -1

<- kcrfw_redo_gen_ext: 1
```

The calculated "start write threshold" is 3665

Calculation:

$4582 * 80/100 = 3665$

The write threshold is higher than the number of allocated redo blocks, so the session does not post lgwr to start writing.



Log Buffer Write Threshold – Parallel LGWR / Thick Mode / Multiple Strands Active

```
# gdb -q -x ./kcrfw_redo_gen_ext-poke.gdb -p 34534

-> kcrfw_redo_gen_ext: 0x2, 0x7ffd7a86c4c8, 0x7ffd7a86c488
    max_strands = 16, actv_strands = 10, write_sz = 8192, wr_thresh_calc = 655

<- kcobrh: redo_sz = 1408, redo_blocks = 2

    strand: #0
    tot_bufs_kcrfa      12288      12288      0
    memory_bufs_available_kcrfa_cln 12286      12283     -3
    start_wr_thresh_kcrfa_cln      655      652     -3

<- kcrfw_redo_gen_ext: 1
```

The calculated "start write threshold" is 655.

Calculation:
 $8192 * 80 / 100 / 10 = 655$

The write threshold is higher than the number of allocated redo blocks, so the session does not post lgwr to start writing.



Appendix G: LGWR Trace Event 10468



Trace Event 10468

Event Description

```
10468, 00000, "log writer debug module"  
// *Document: NO  
// *Cause:  
// *Action: Set this event to the appropriate level for log writer debugging.
```

Enable and Disable Trace

```
-- Select target process  
SQL> oradebug setorapname lgwr  
  
-- Enable Level 1 trace  
SQL> oradebug event '10468 trace name context forever, level &&level';  
  
-- Disable trace  
SQL> oradebug event '10468 trace name context off';
```

You can also enable and disable event 10468 system-wide using 'ALTER SYSTEM ...'

Trace Levels

```
32 : ALFS diagnostics  
1024 : ALWE diagnostics  
2048 : ALWE verbose diagnostics  
32768 : OLRW diagnostics  
65536 : OLRW verbose diagnostics  
524288 : kcrfw slave details
```

ALWE and OLRW trace details are logged to the alert log!

Note: You may need to enable event 10468 in different processes and at different levels!



Appendix H: struct alwe



Overlapped Redo Writes – struct alwe (19.26)

In Oracle 19c, the pointer to the the alwe structure is stored in SGA variable klassvp7_

```
struct alwe {
    uint32_t eval_intvl_us;          /* klassvp7[0]      */
    uint32_t blocks_per_sec;        /* klassvp7[1]      */
    uint64_t epoch_ts_prv;          /* klassvp7[2]      */
    uint64_t normal_writes;         /* klassvp7[4]      */
    uint32_t seqno;                  /* klassvp7[6]      */
    uint32_t eval_fraction;         /* klassvp7[7]      */
    int64_t eval_writes;             /* klassvp7[8]      */
    uint64_t blocks;                 /* klassvp7[10]     */
    int64_t blocks_fraction;        /* klassvp7[12]     */
    uint64_t pic_start;              /* klassvp7[14]     */
    uint64_t ow_blocks;              /* klassvp7[16]     */
    uint64_t tot_blocks;             /* klassvp7[18]     */
    uint64_t tot_time;               /* klassvp7[20]     */
    uint64_t epoch_ts;               /* klassvp7[22]     */
    uint32_t unknown1;               /* klassvp7[24]     */
    uint32_t unknown2;               /* klassvp7[25]     */
    uint32_t unknown3;               /* klassvp7[26]     */
    uint32_t unknown4;               /* klassvp7[27]     */
    uint32_t lfs_factor;             /* klassvp7[28]     */
    int32_t anomalies;               /* klassvp7[29]     */
    int32_t ol_writes;               /* klassvp7[30]     */
    int32_t flags;                   /* klassvp7[31]     */
};
```



Appendix I: struct olrw



Overlapped Redo Writes – struct olrw (19.26)

```
struct olrw
    uint32_t flags; /* klassvp8[0] */
    uint32_t write_sz; /* klassvp8[1] */
    uint32_t stall_sz; /* klassvp8[2] */
    uint32_t write_sz_prv; /* klassvp8[3] */
    uint32_t poke_pct; /* klassvp8[4] */
    uint32_t tgt_log_write_sz; /* klassvp8[5] */
    uint32_t tgt_log_write_tmo; /* klassvp8[6] */
    uint32_t thin_p1; /* klassvp8[7] */
    uint32_t wait_zone_start; /* klassvp8[8] */
    uint32_t thin_p2; /* klassvp8[9] */
    uint32_t wait_min_sz; /* klassvp8[10] */
    uint32_t thin_p3; /* klassvp8[11] */
    uint32_t wait_max_sz; /* klassvp8[12] */
    uint32_t thin_p4; /* klassvp8[13] */
    uint32_t wait_min_tmo_usec; /* klassvp8[14] */
    uint32_t thin_p5; /* klassvp8[15] */
    uint32_t wait_max_tmo_usec; /* klassvp8[16] */
    uint32_t thick_p1; /* klassvp8[17] */
    uint32_t fallback_zone_end; /* klassvp8[18] */
    uint32_t thick_p2; /* klassvp8[19] */
    uint32_t overflow_zone_start; /* klassvp8[20] */
    uint32_t thick_p3; /* klassvp8[21] */
    uint32_t thick_min_sz; /* klassvp8[22] */
    uint32_t thick_p4; /* klassvp8[23] */
    uint32_t thick_max_sz; /* klassvp8[24] */
    uint32_t thick_tmo_usec; /* klassvp8[25] */
    uint32_t arbiter; /* klassvp8[26] */
    uint32_t opt_numgrps; /* klassvp8[27] */
    uint32_t opt_grp_sz; /* klassvp8[28] */
};
```

In Oracle 19c, the pointer to the the olrw structure is stored in SGA variable klassvp8_