LDT: Lightweight Dirty Tracking of Memory Pages for x86 Systems

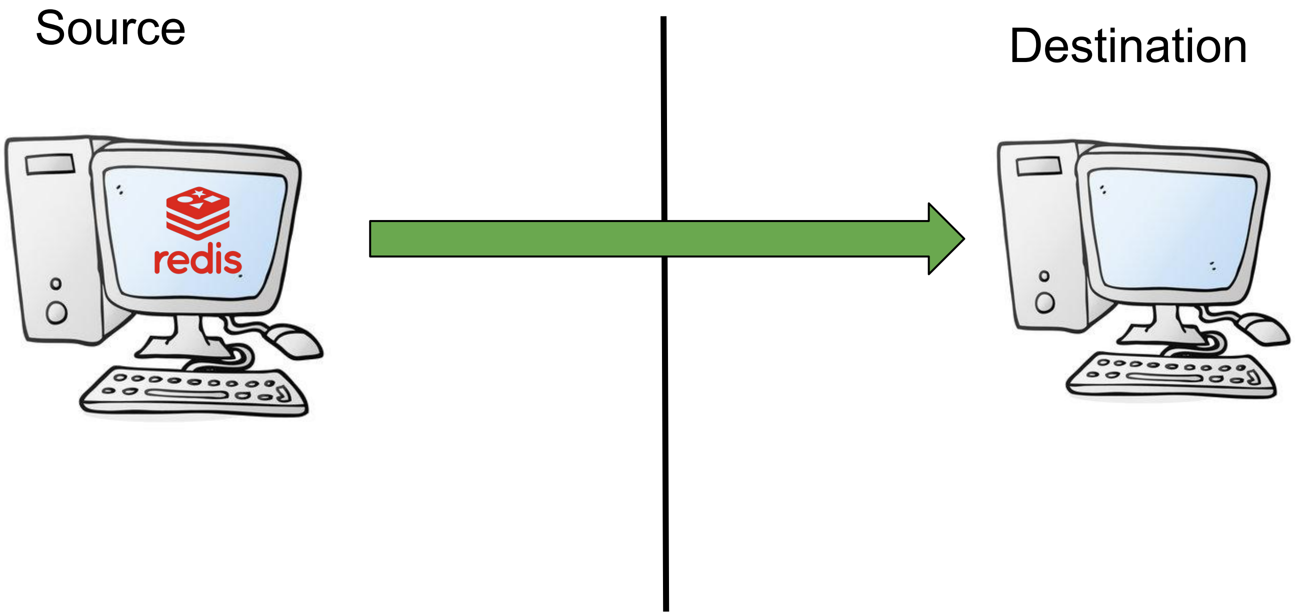
IRS 2023

Rohit Singh
IIT Kanpur

Arun KP
IIT Kanpur

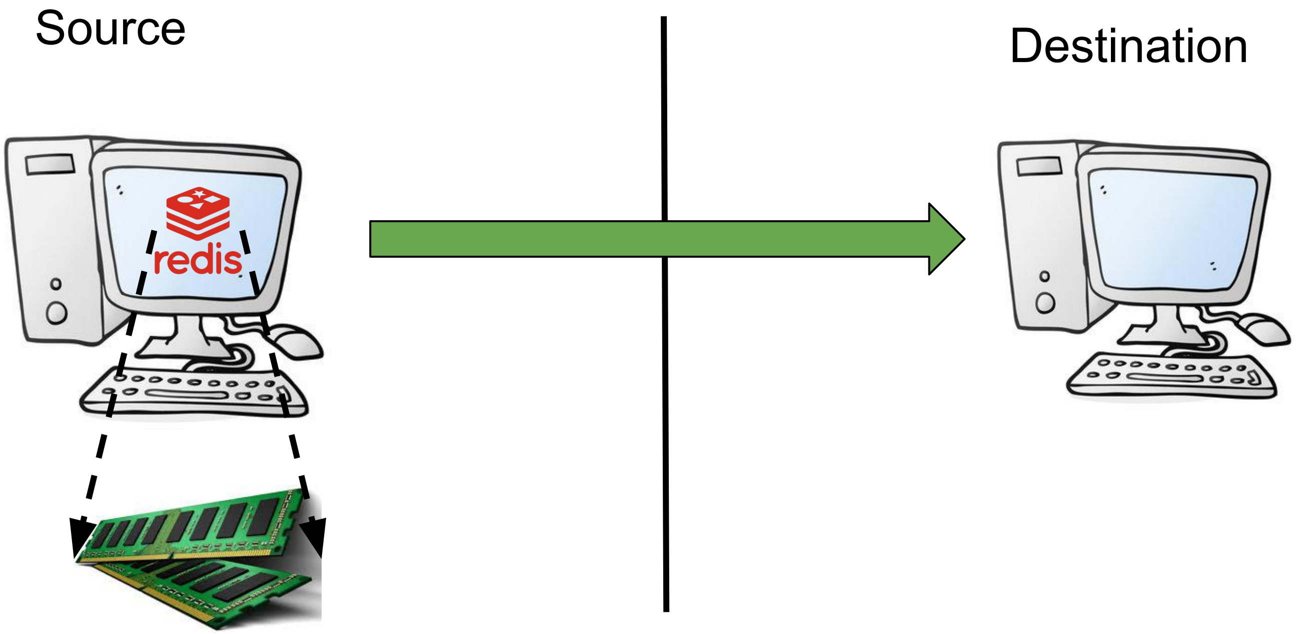
Debadatta Mishra
IIT Kanpur

Process Migration



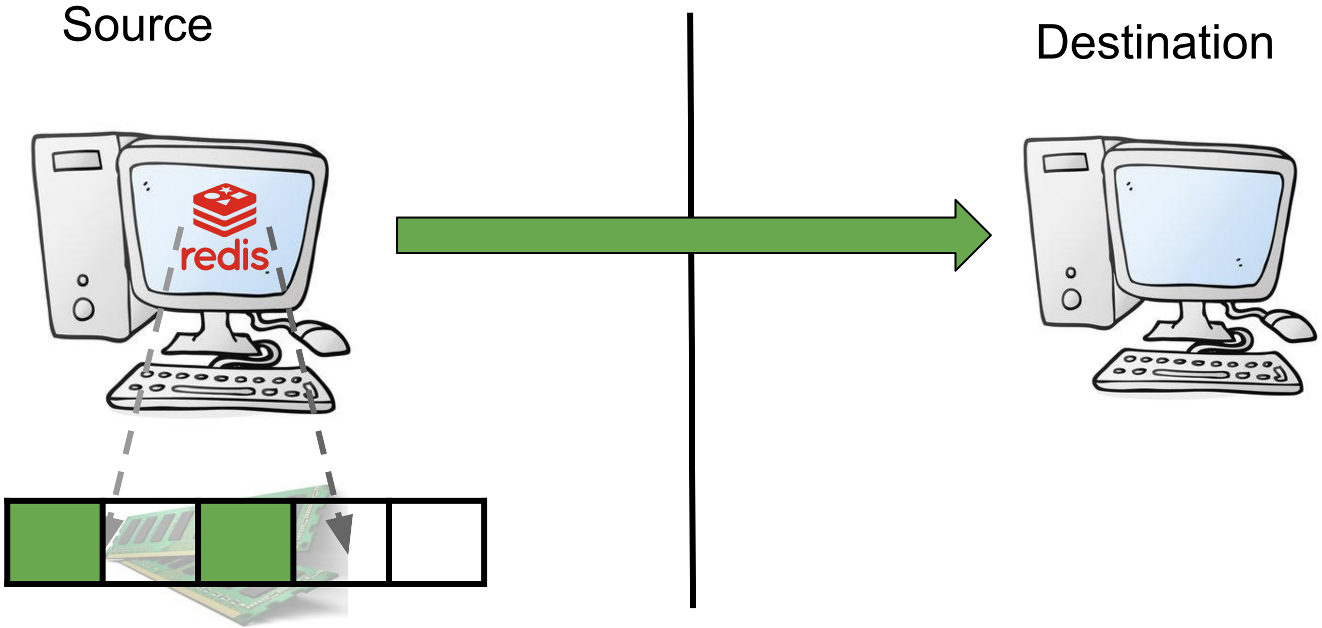
Ref: google images

Process Migration



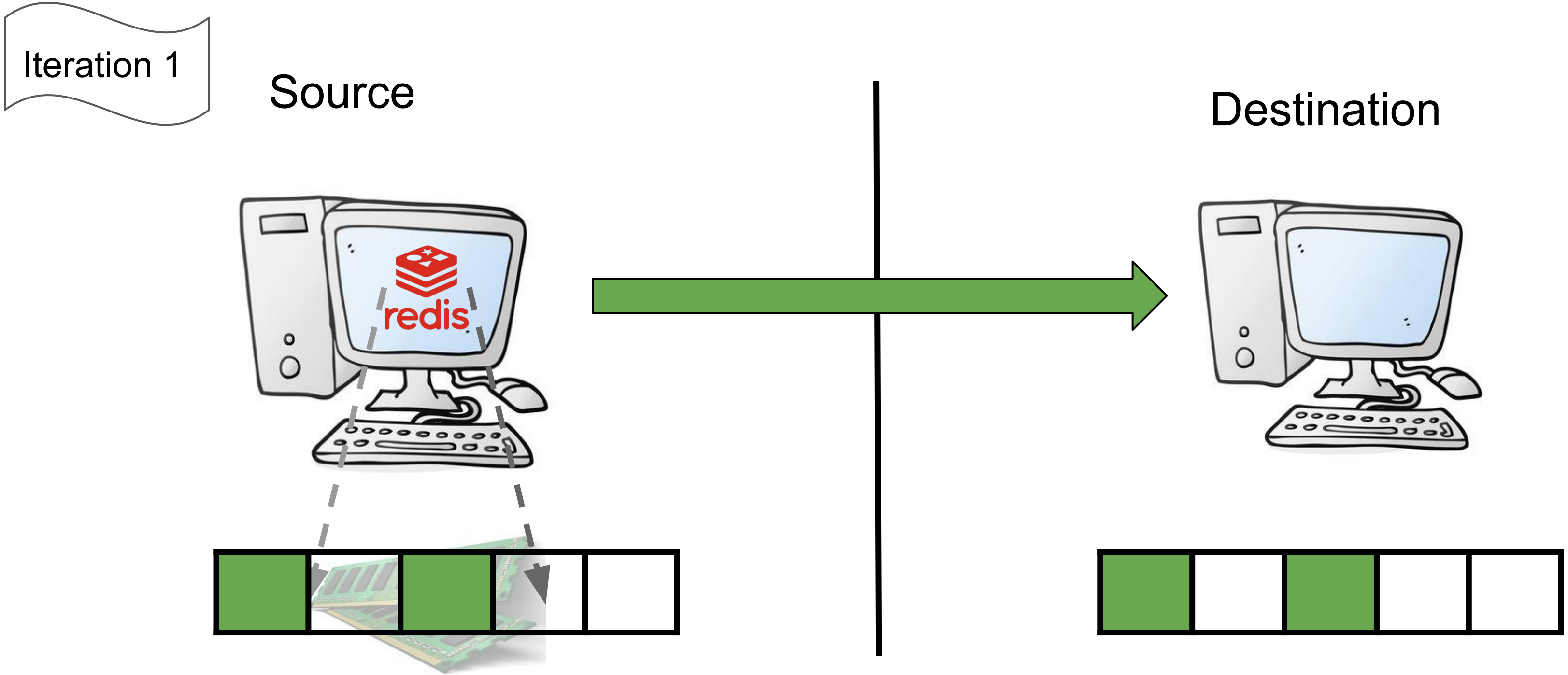
Ref: google images

Process Migration



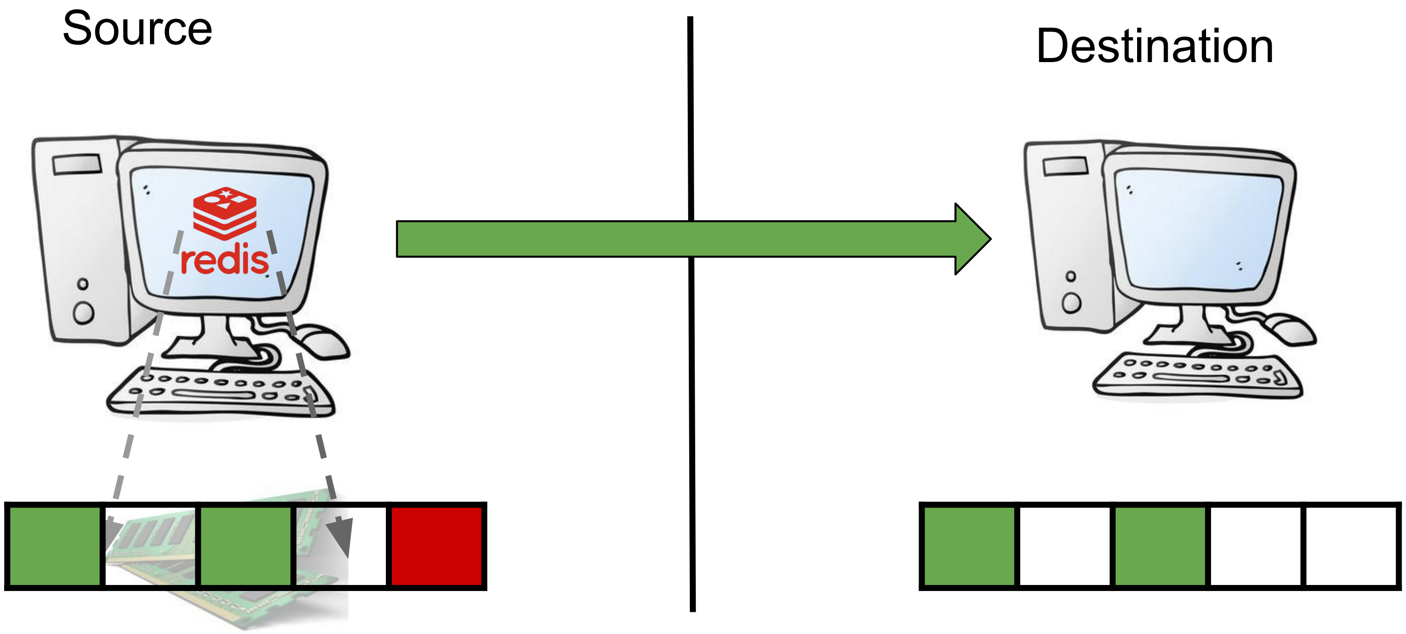
Ref: google images

Process Migration



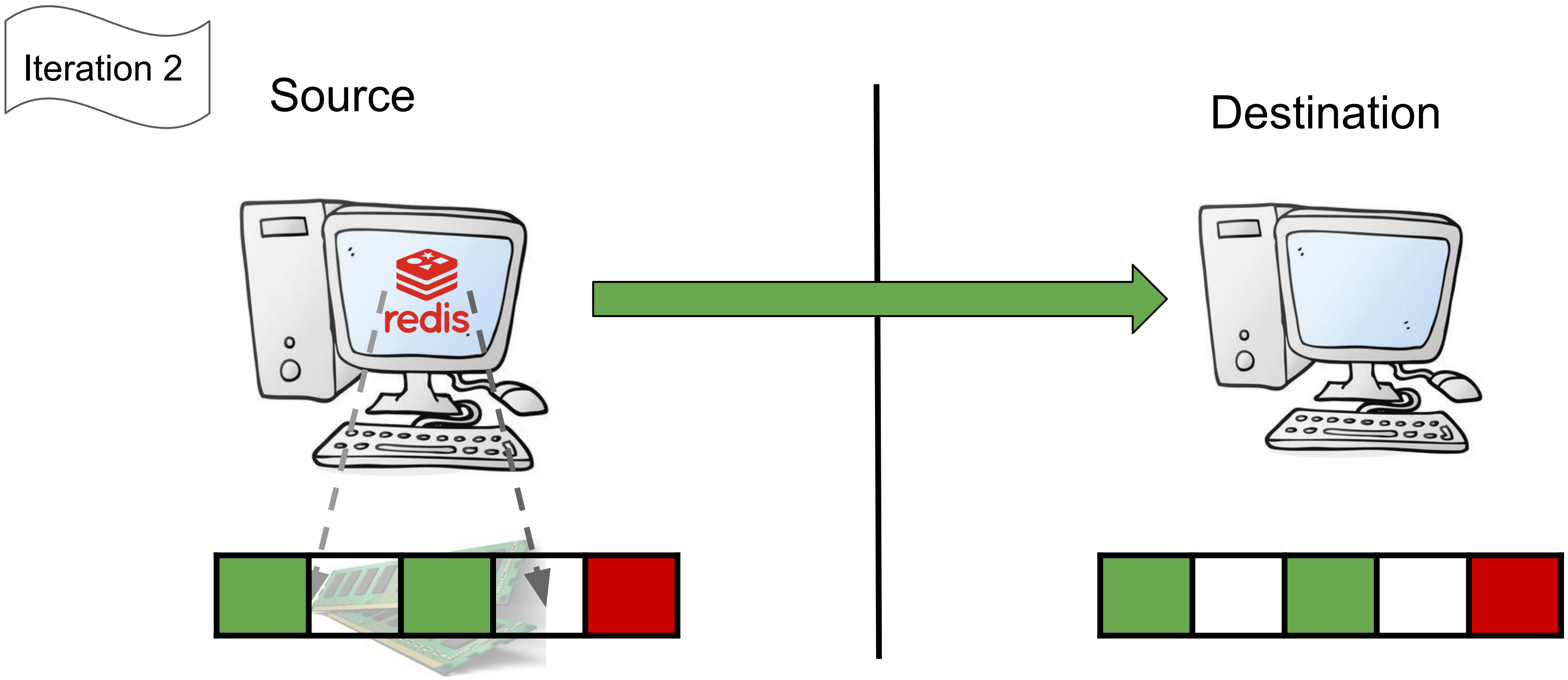
Ref: google images

Process Migration



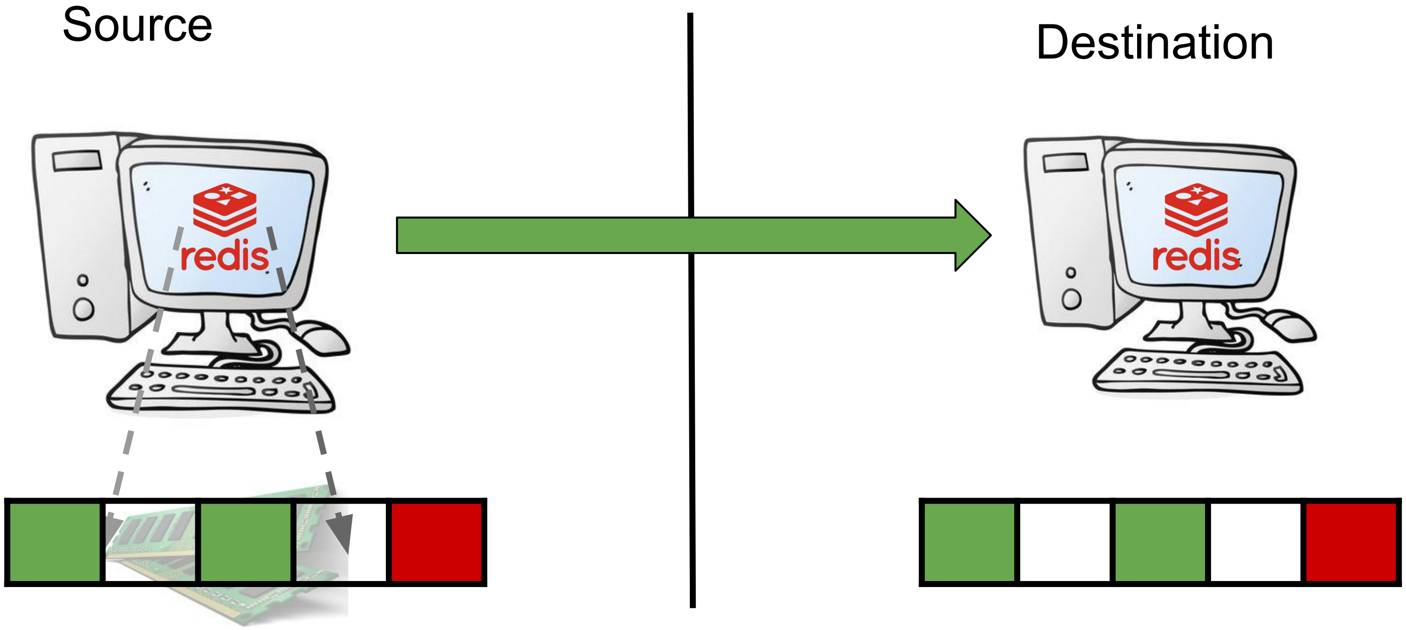
Ref: google images

Process Migration



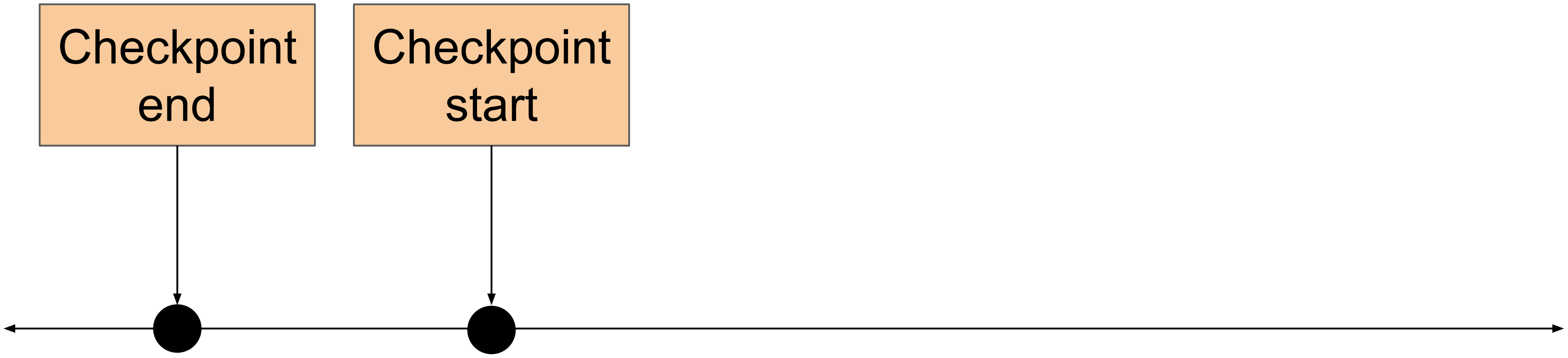
Ref: google images

Process Migration



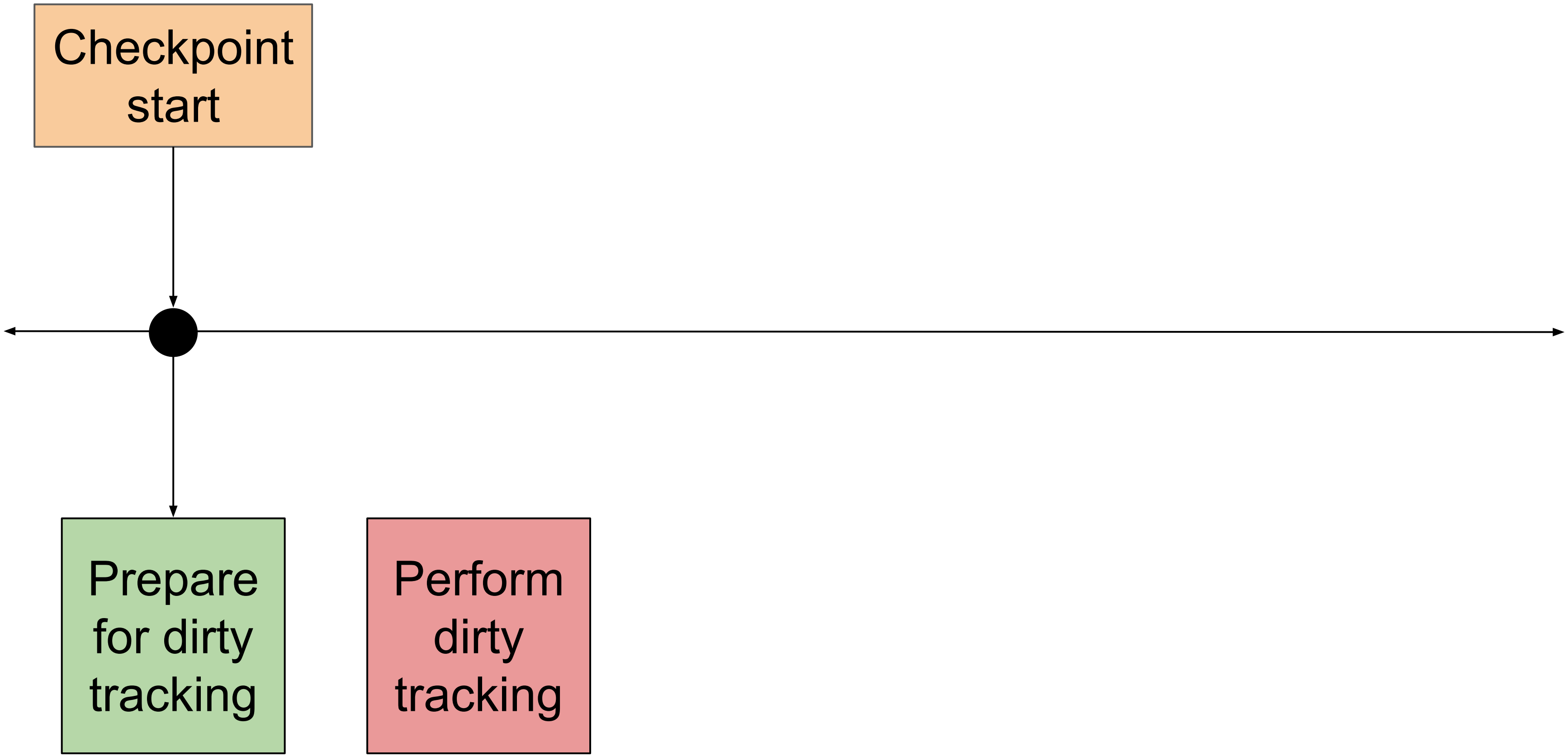
Ref: google images

Incremental Checkpointing

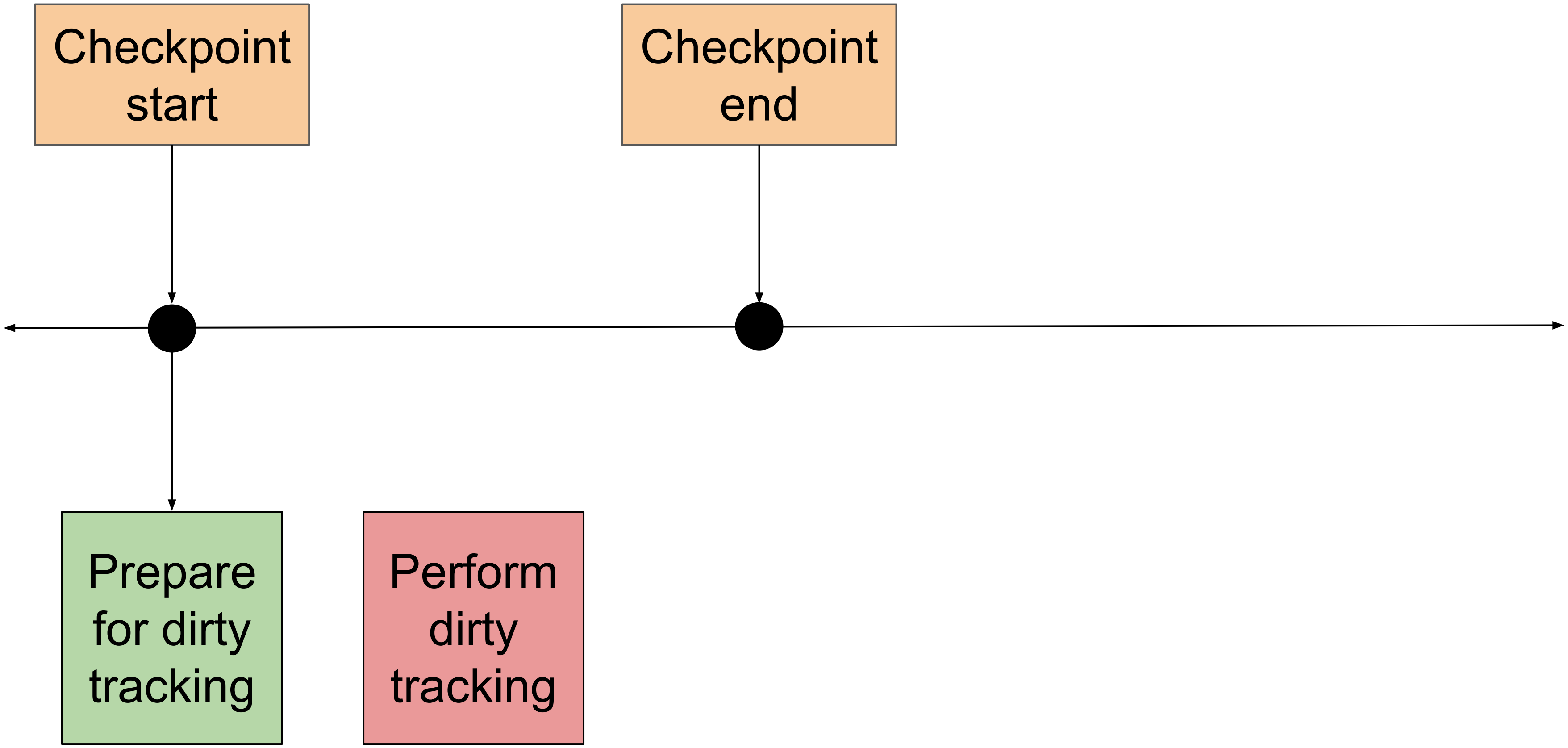


Incremental Checkpointing

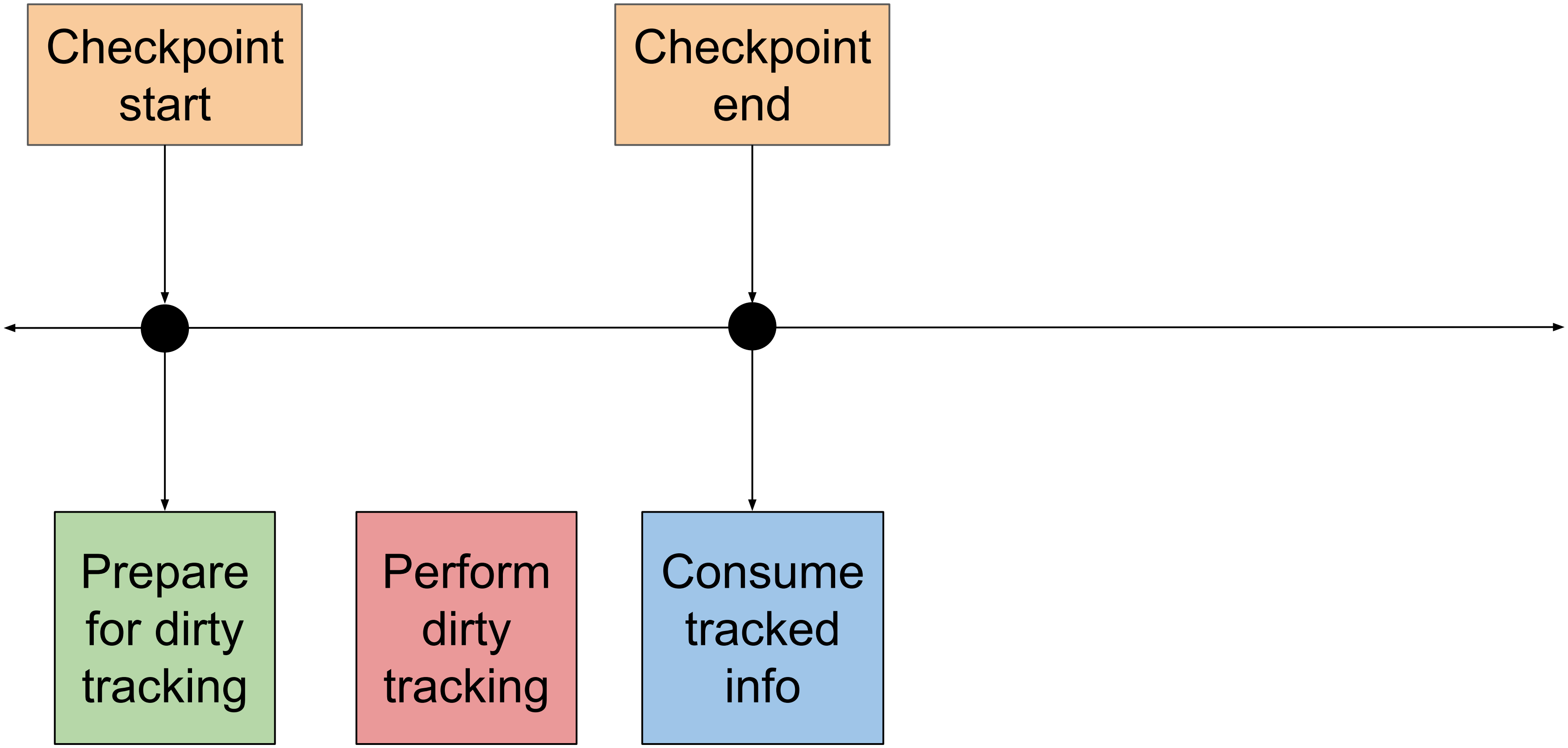
Incremental Checkpointing



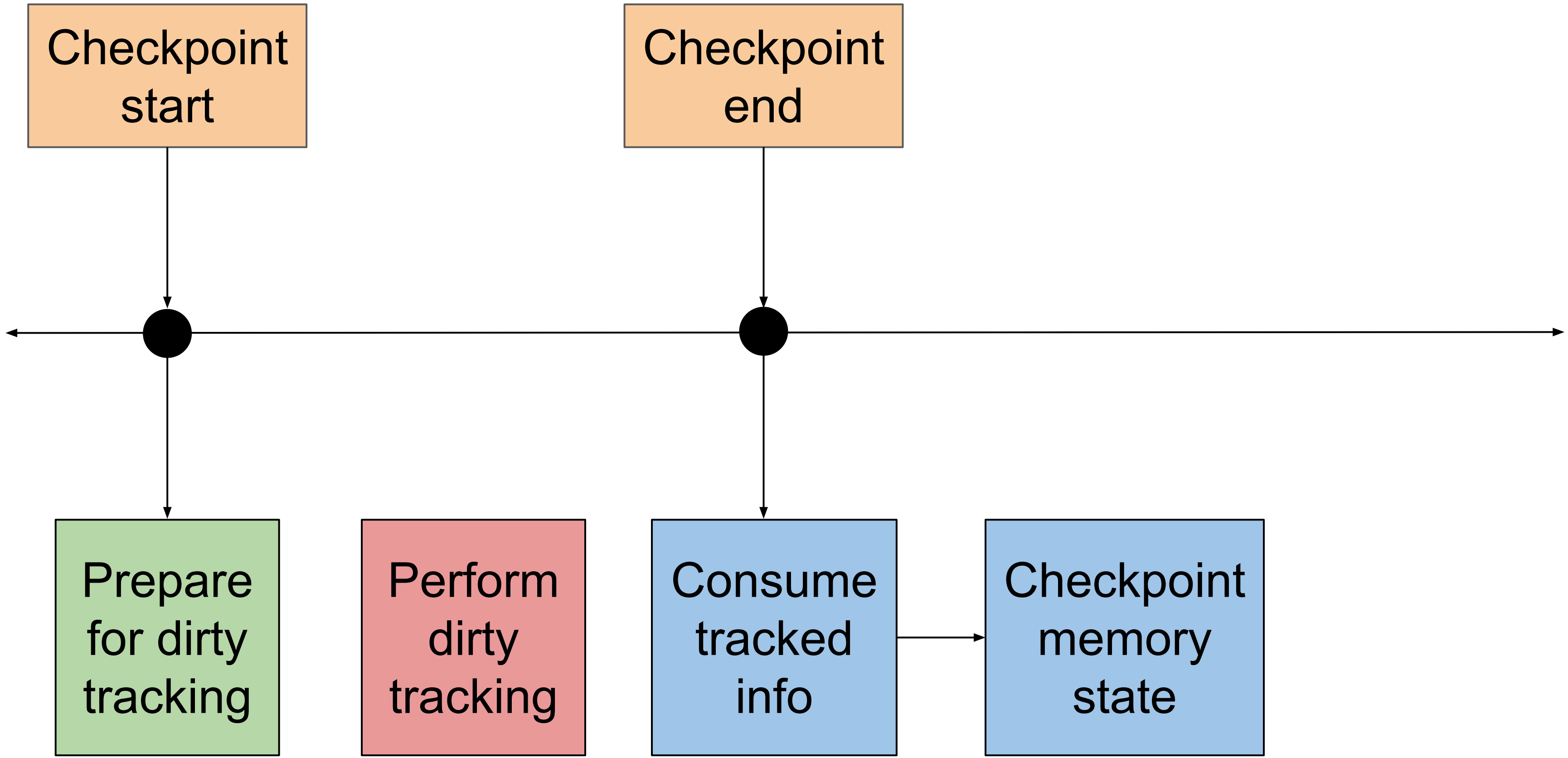
Incremental Checkpointing



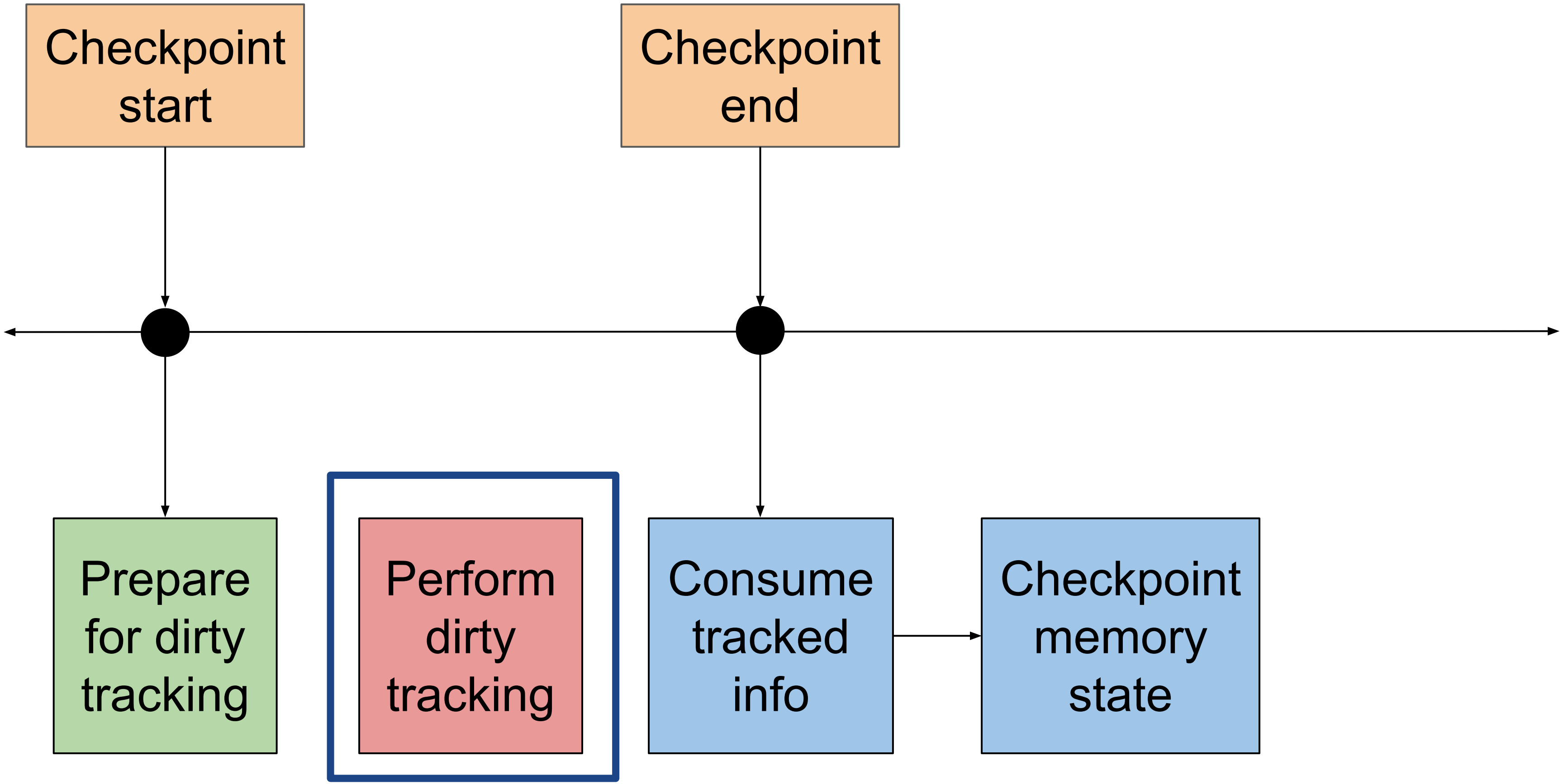
Incremental Checkpointing



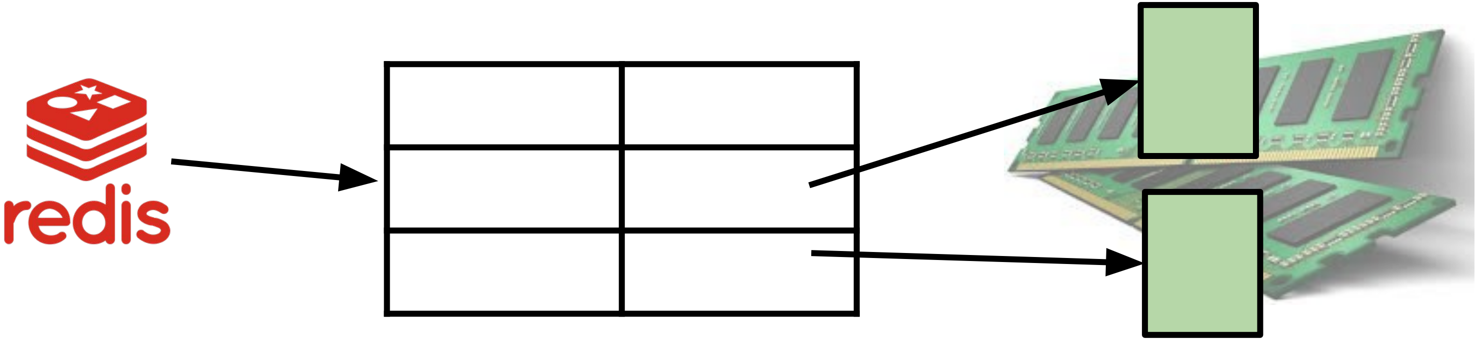
Incremental Checkpointing



Incremental Checkpointing



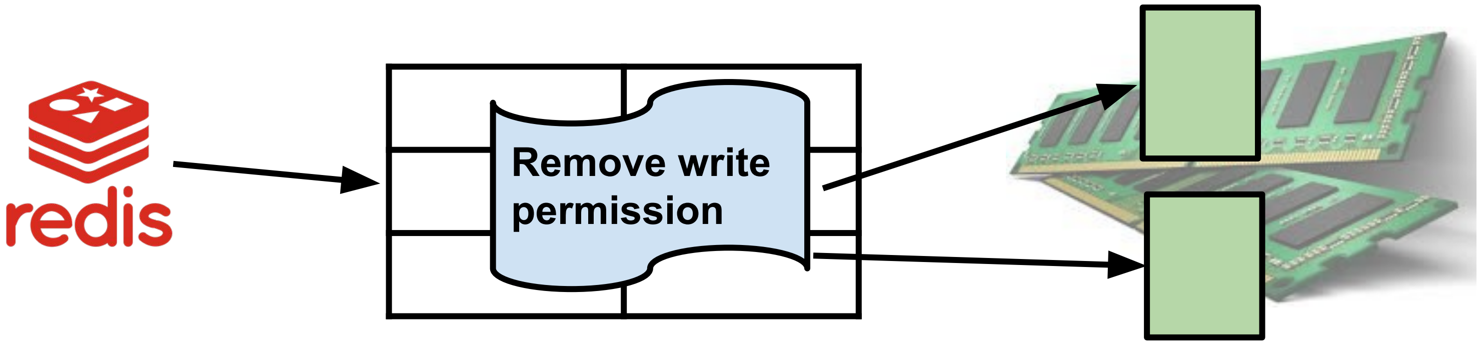
State of the Art: Dirty Tracking with write faults



Translation Table

Ref: google images

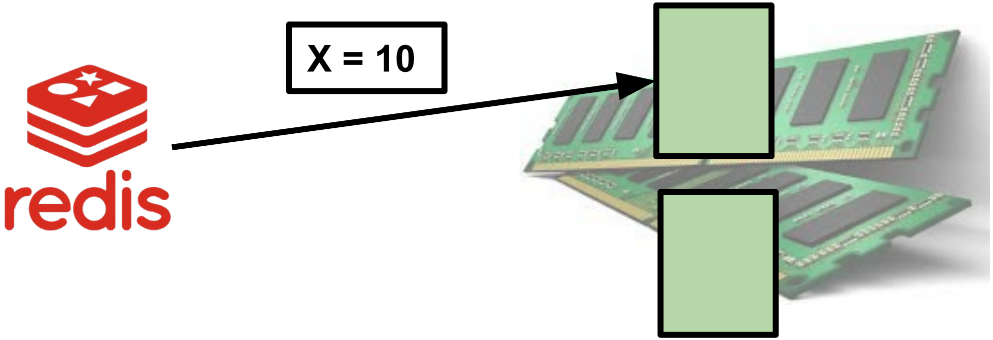
State of the Art: Dirty Tracking with write faults



Translation Table

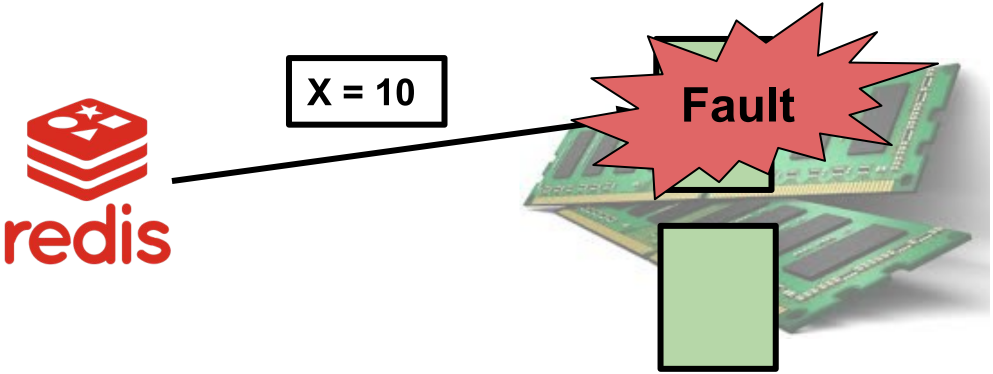
Ref: google images

State of the Art: Dirty Tracking with write faults



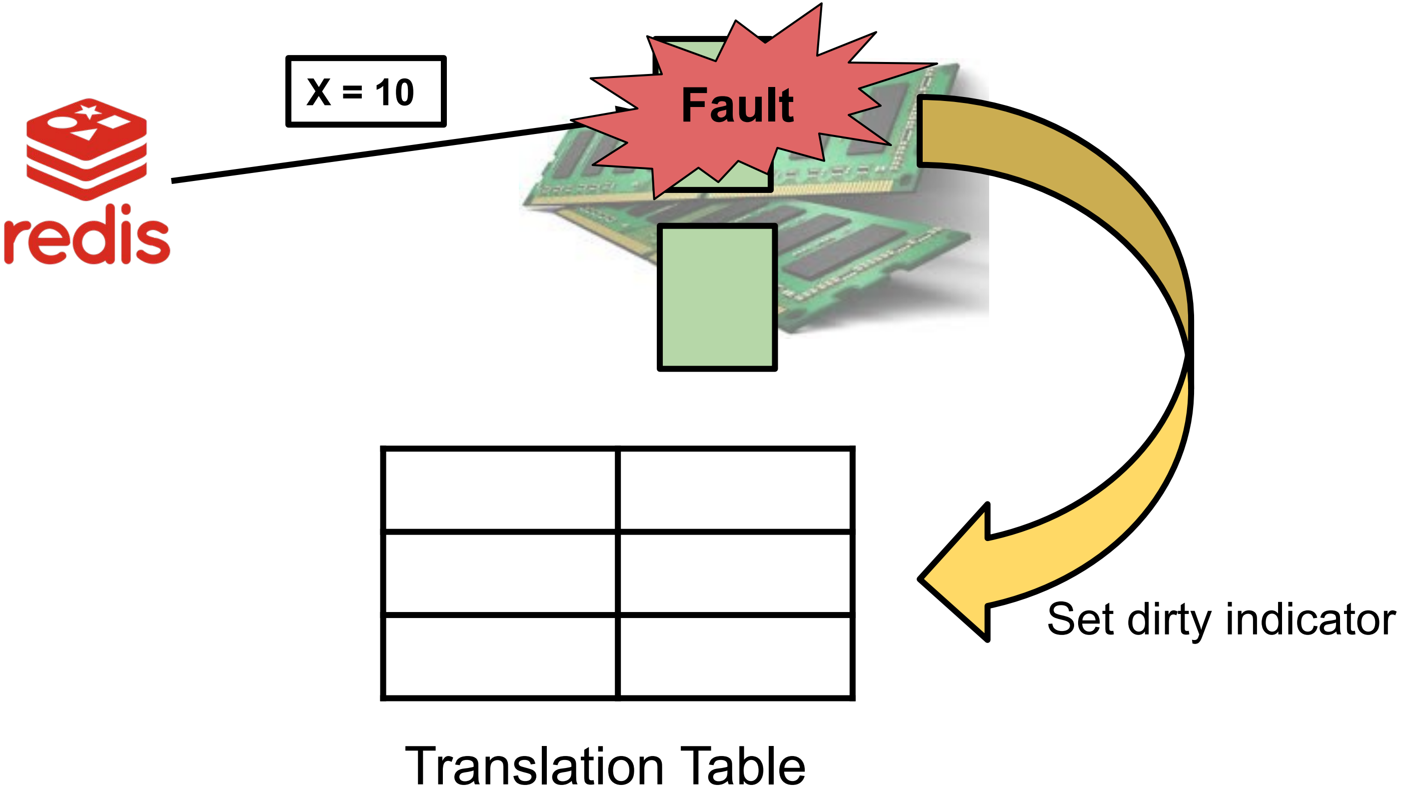
Ref: google images

State of the Art: Dirty Tracking with write faults



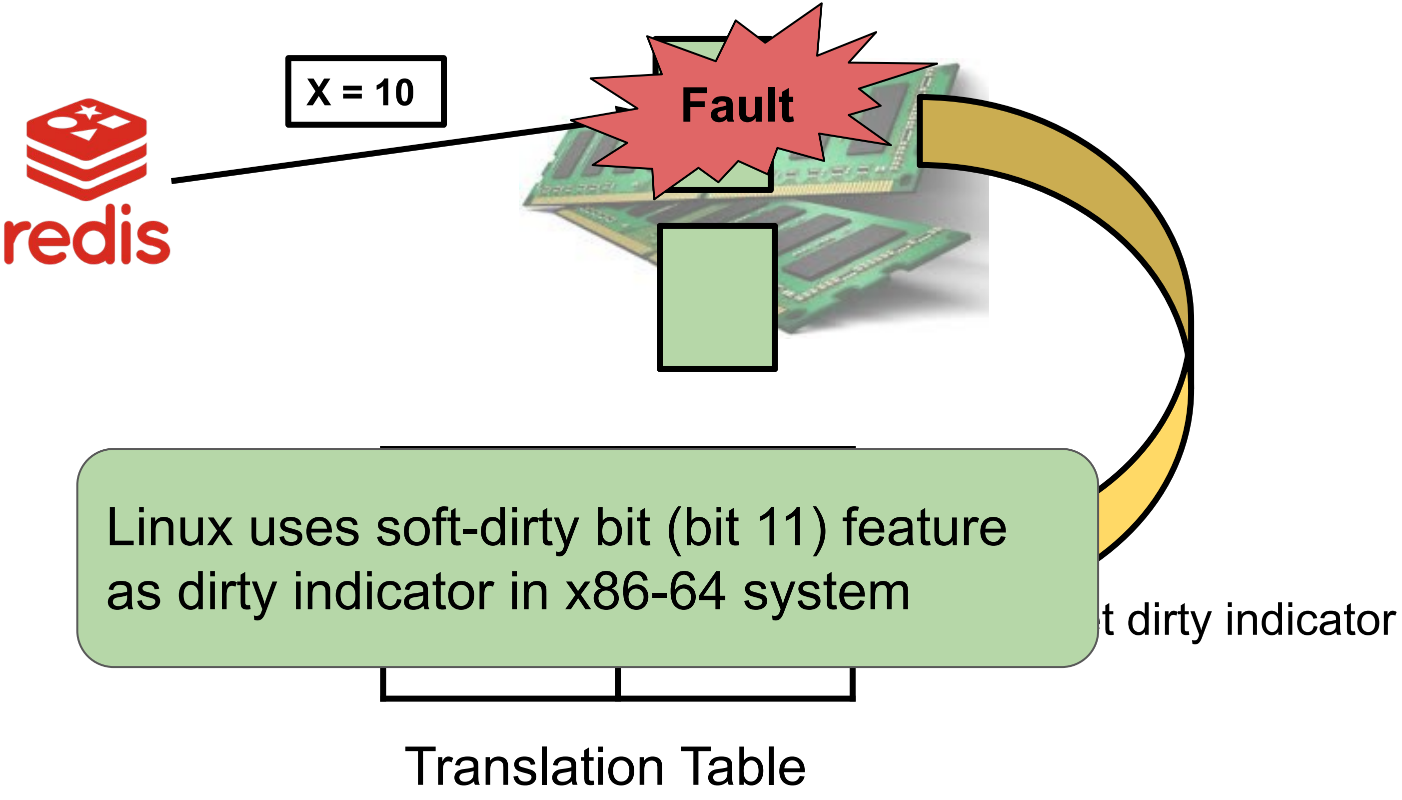
Ref: google images

State of the Art: Dirty Tracking with write faults



Ref: google images

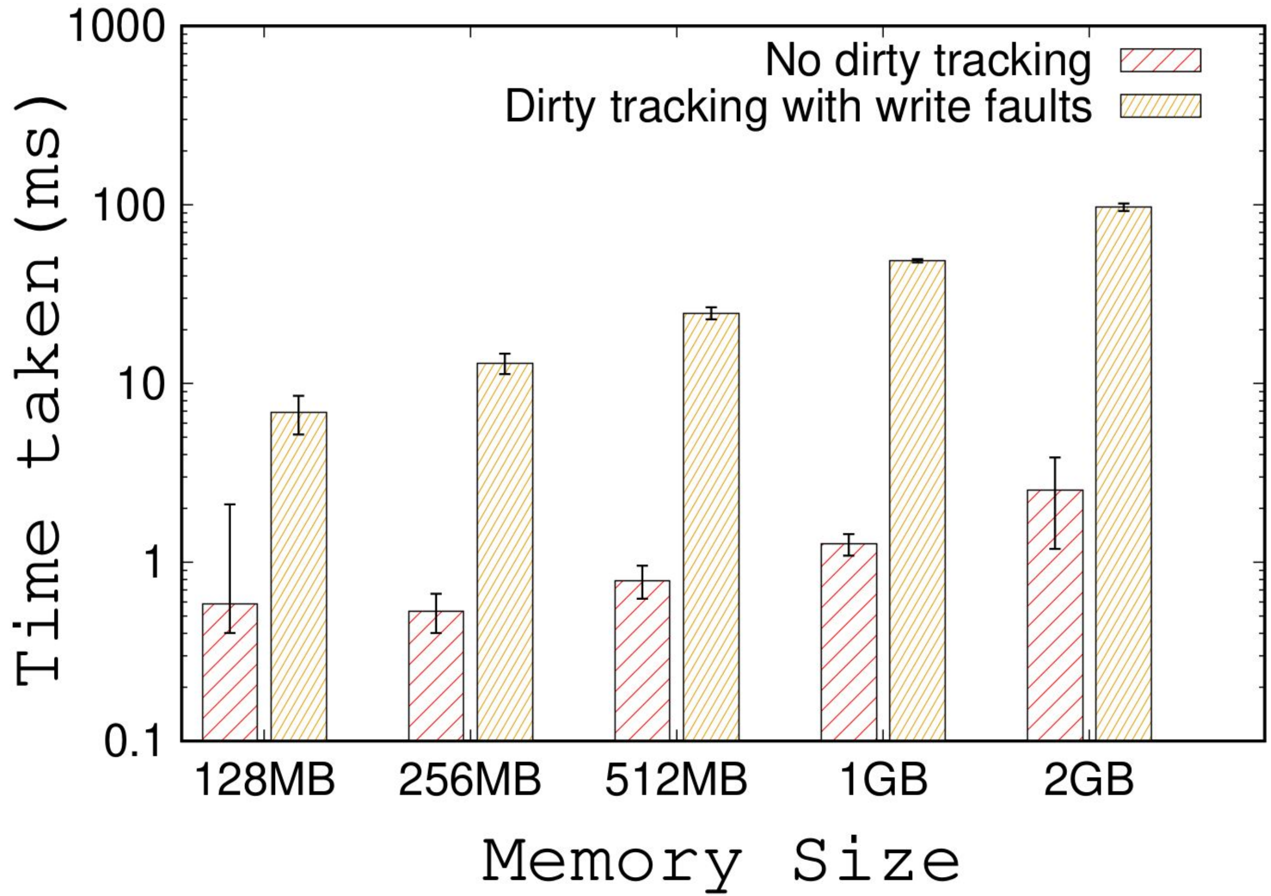
State of the Art: Dirty Tracking with write faults



Linux uses soft-dirty bit (bit 11) feature as dirty indicator in x86-64 system

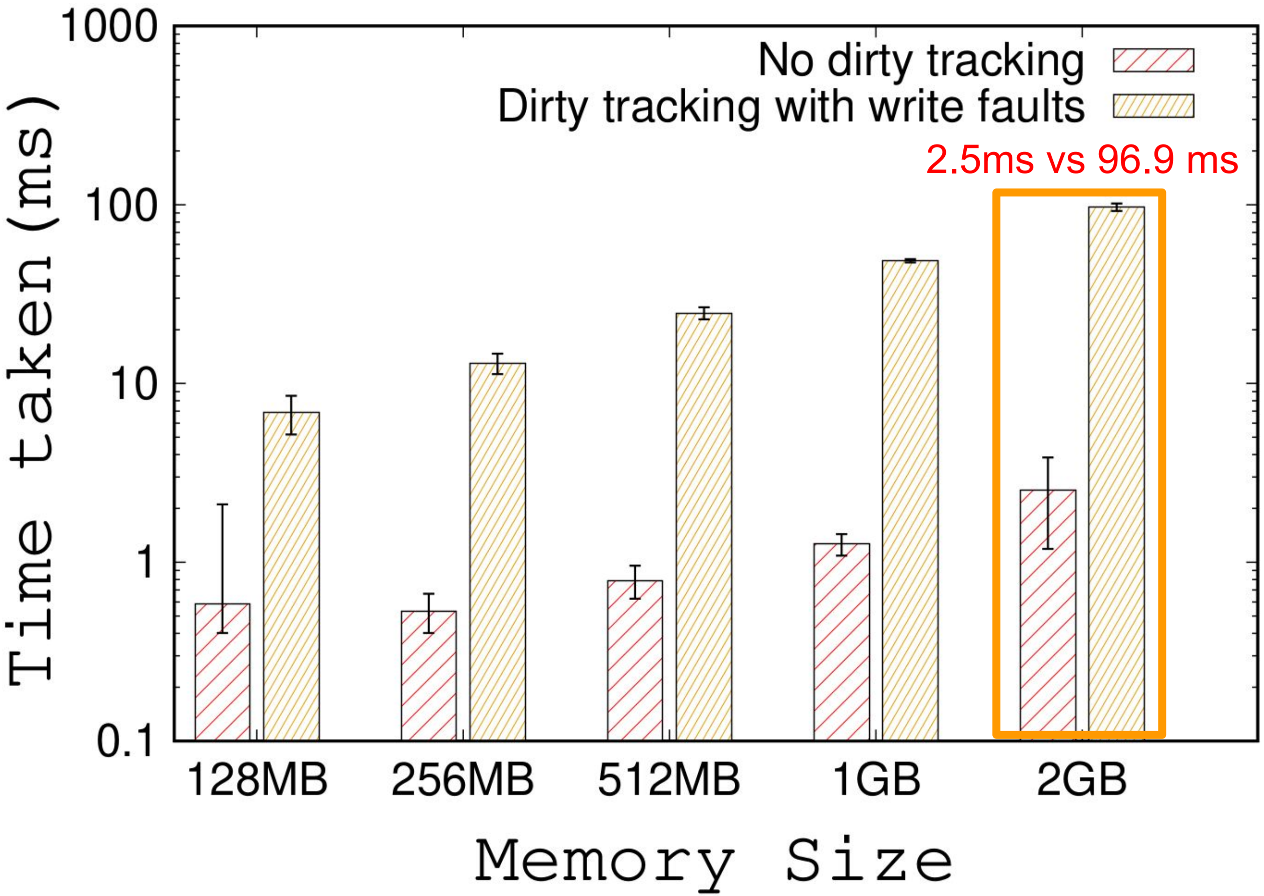
Ref: google images

Dirty Tracking with write faults overhead



- Workload with different working set sizes where 1 byte of each page is written

Dirty Tracking with write faults overhead

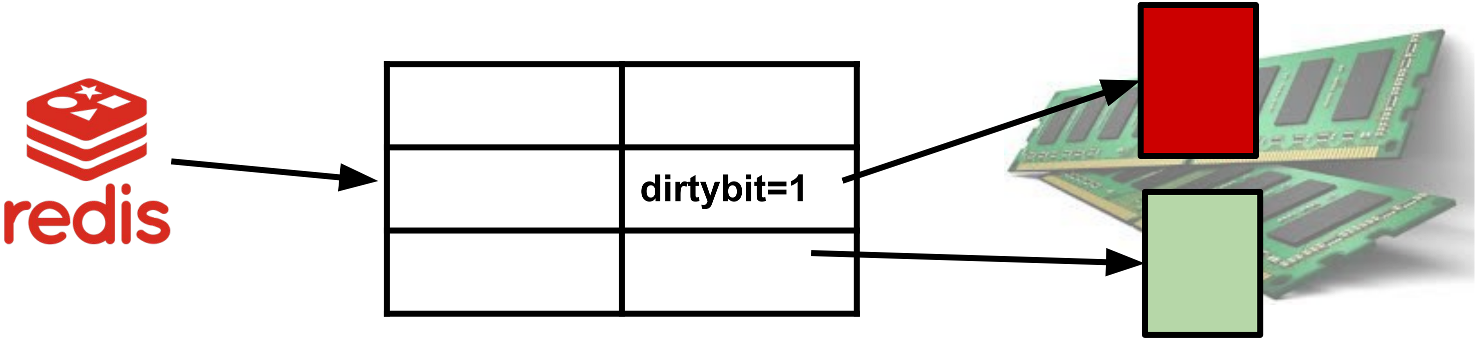


- Workload with different working set sizes where 1 byte of each page is written

- Time taken to write for each working set size is many times more in case of dirty tracking with write faults

Can we use an alternative approach with less overhead for dirty tracking?

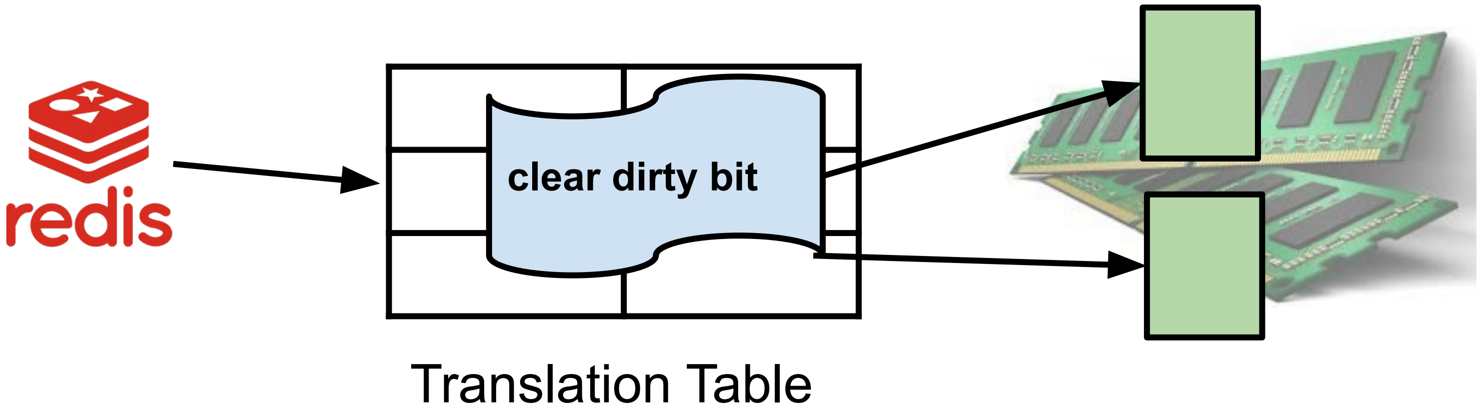
Alternative: Dirty Tracking with Dirty Bit (x86)



Translation Table

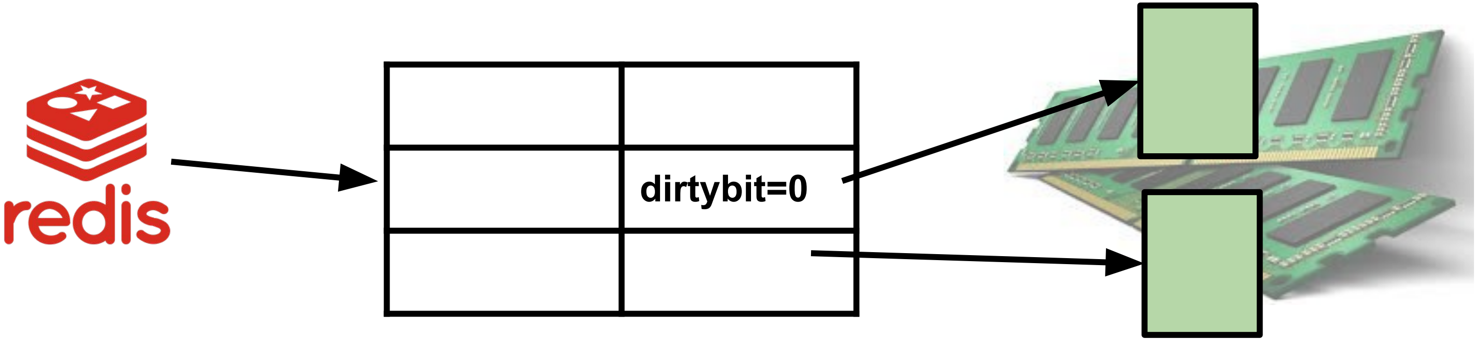
Dirty bit is set by hardware if a memory page is modified.

Alternative: Dirty Tracking with Dirty Bit (x86)



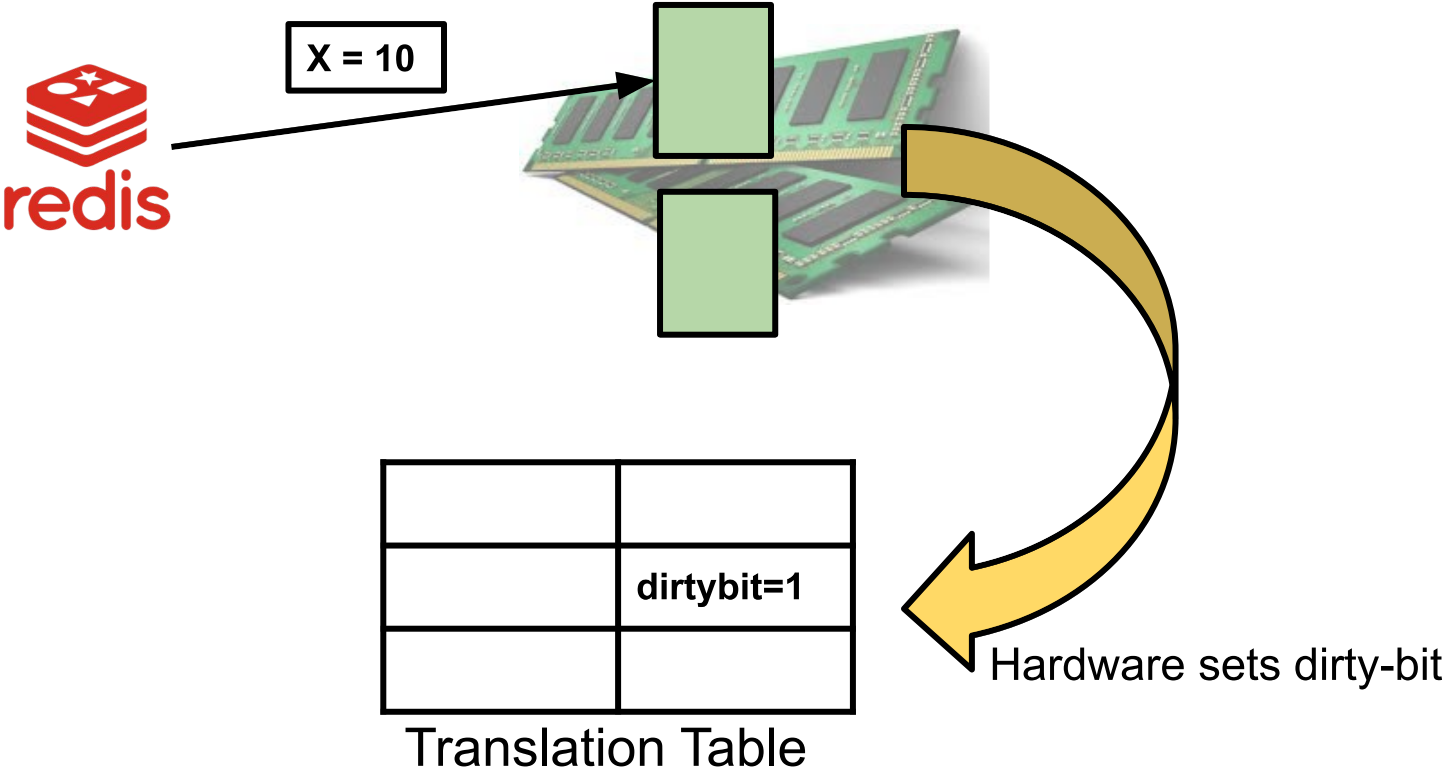
At dirty tracking start, clear dirty bit in translation table.

Alternative: Dirty Tracking with Dirty Bit (x86)

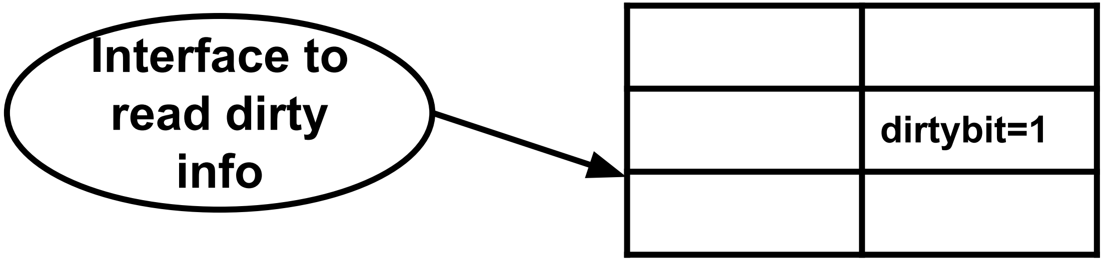


Translation Table

Alternative: Dirty Tracking with Dirty Bit (x86)



Alternative: Dirty Tracking with Dirty Bit (x86)

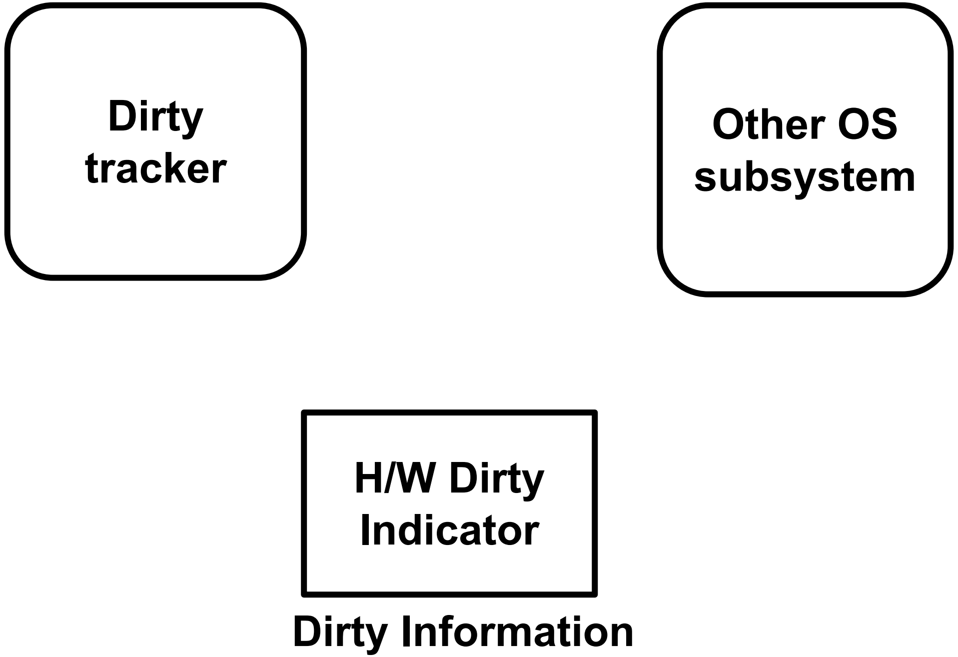


Translation Table

At dirty tracking end, inspect dirty bit in translation table to check page modification.

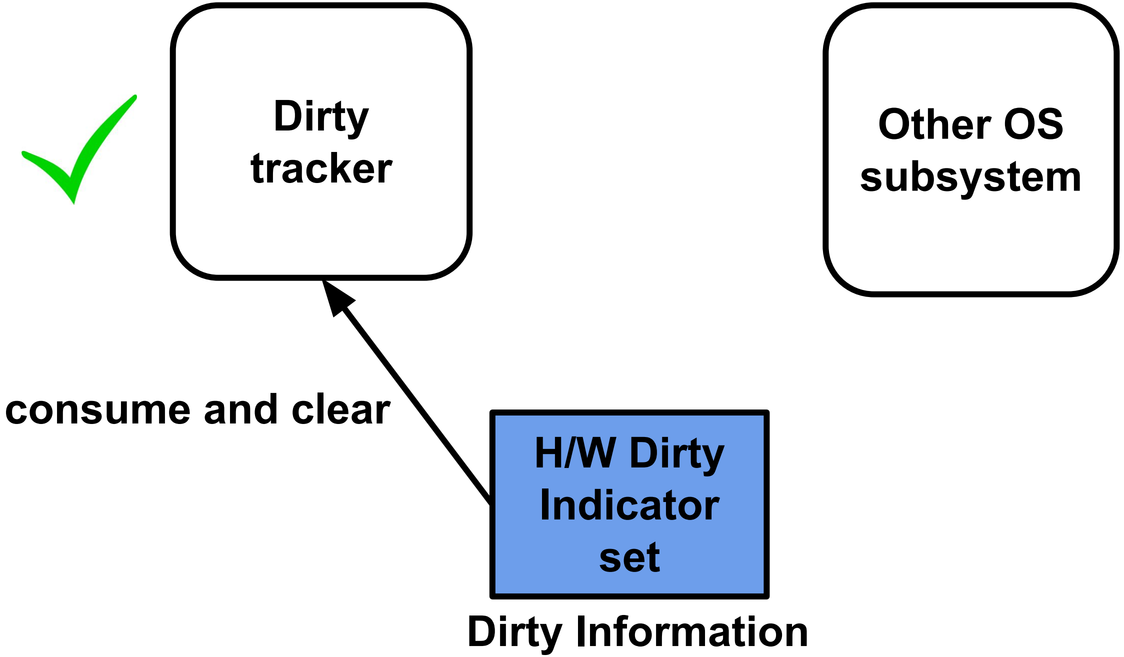
What are the challenges?

Challenge: Interaction with OS subsystems



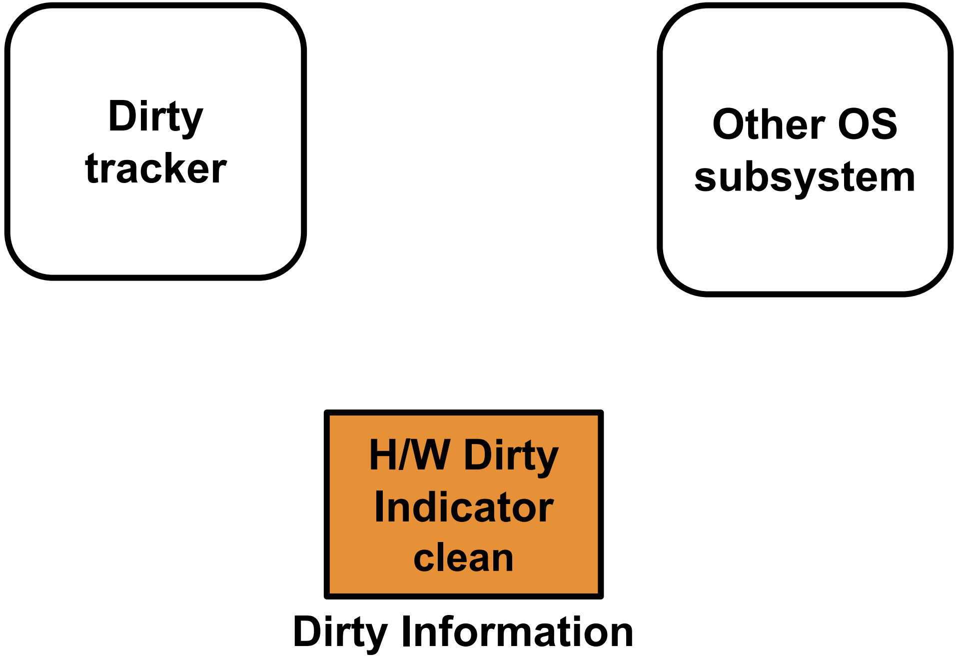
Challenge: Interaction with OS subsystems

Scenario : 1



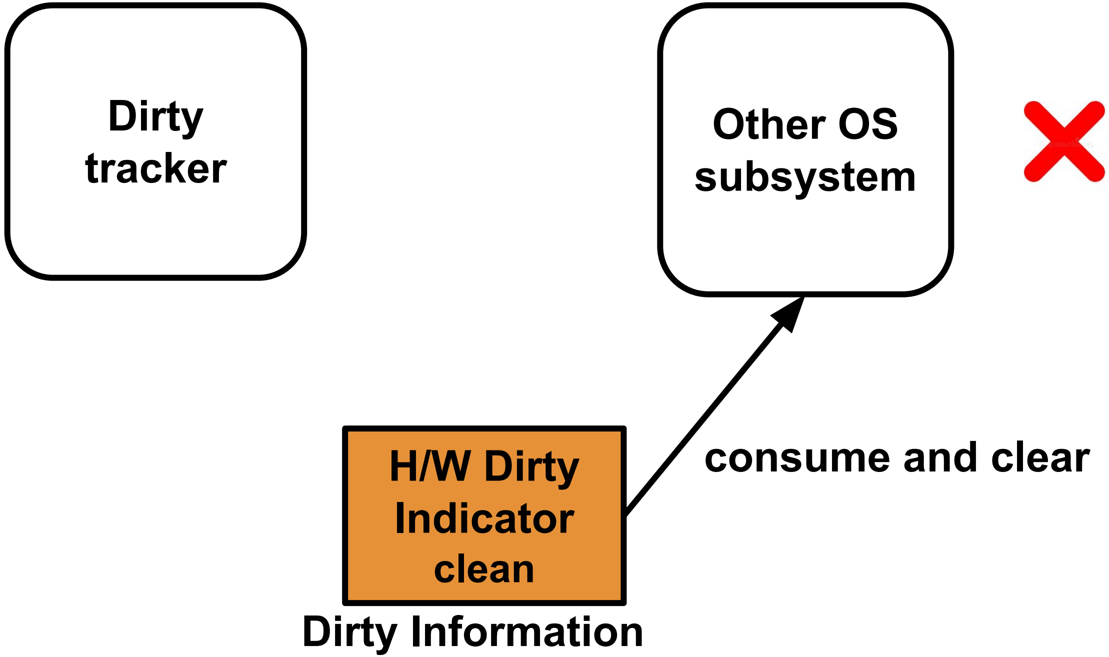
Challenge: Interaction with OS subsystems

Scenario : 1



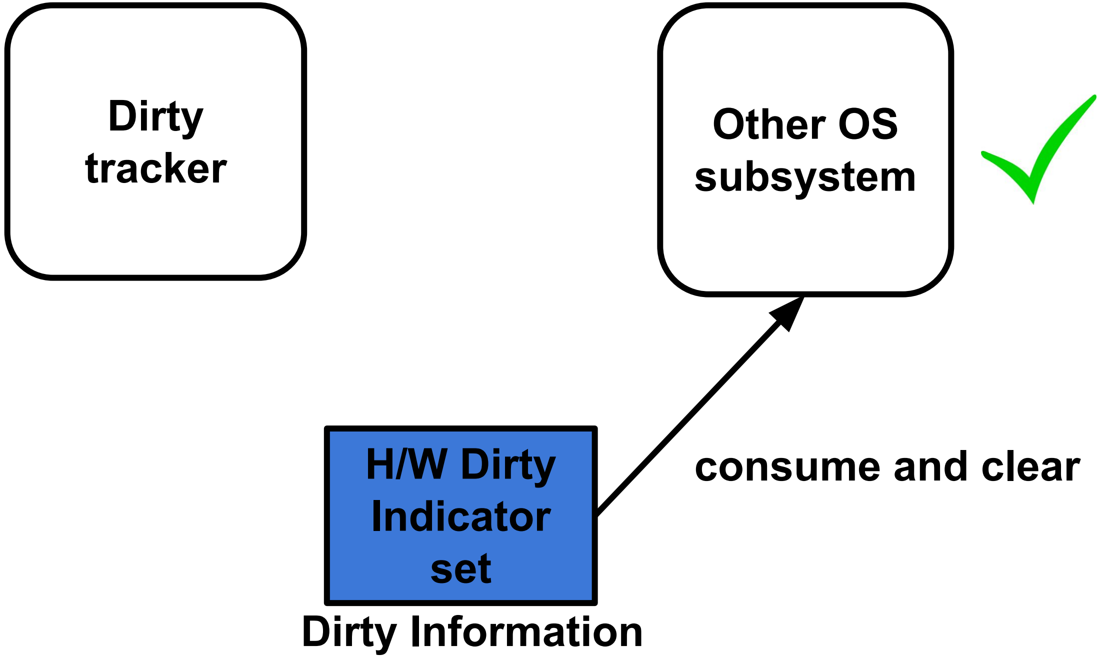
Challenge: Interaction with OS subsystems

Scenario : 1



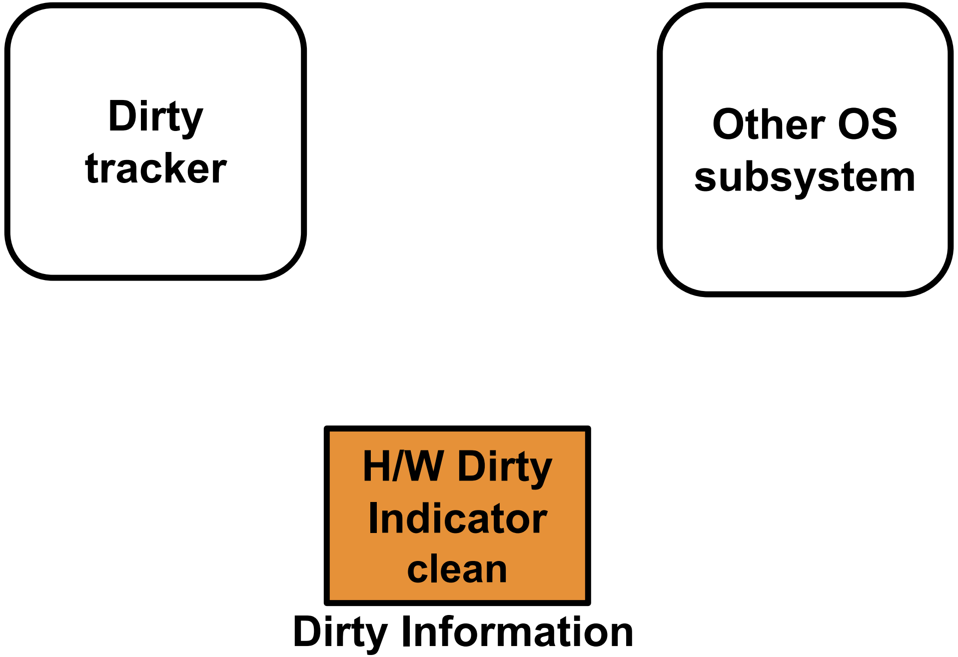
Challenge: Interaction with OS subsystems

Scenario : 2



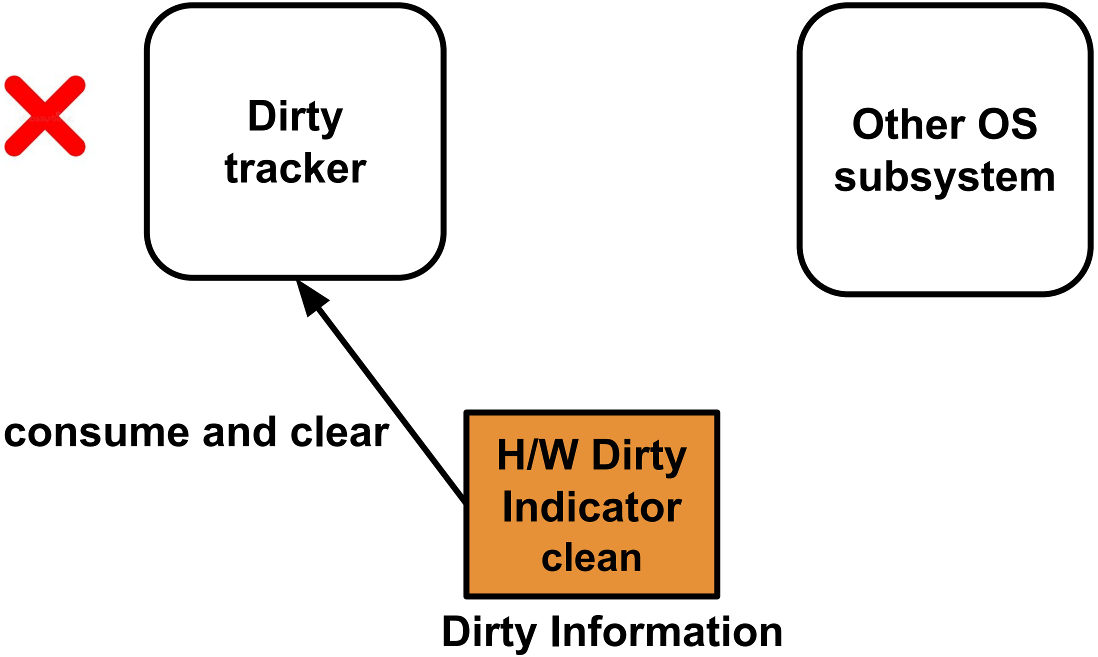
Challenge: Interaction with OS subsystems

Scenario : 2



Challenge: Interaction with OS subsystems

Scenario : 2



How to overcome the challenges?

LDT: Idea to overcome the challenge

- Translation table entry has unused bits for the software to use.

LDT: Idea to overcome the challenge

- Translation table entry has unused bits for the software to use.
- x86-64 has ~10 unused bits in translation table entry.

LDT: Idea to overcome the challenge

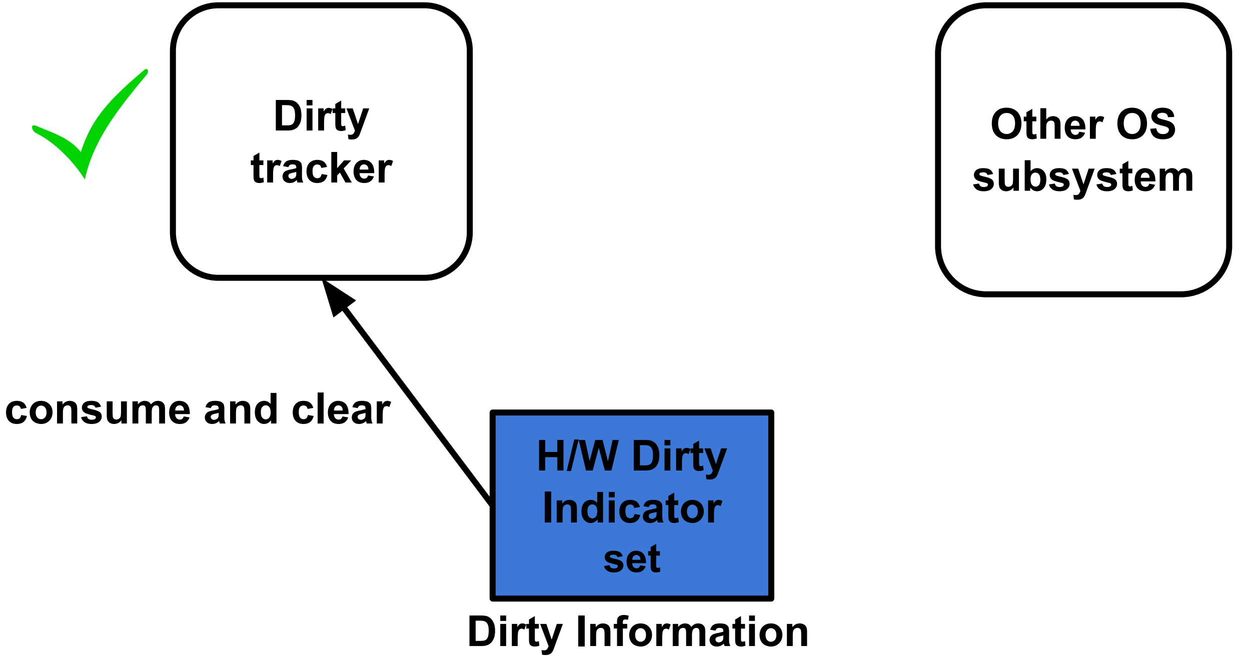
- Translation table entry has unused bits for the software to use.
- x86-64 has ~10 unused bits in translation table entry.
- Use 2 unused bits in x86-64 to coordinate with other OS subsystem.
  - Use 1st unused bit to maintain backup of **dirty bit** information for other OS subsystems in case LDT consume and clears.

LDT: Idea to overcome the challenge

- Translation table entry has unused bits for the software to use.
- x86-64 has ~10 unused bits in translation table entry.
- Use 2 unused bits in x86-64 to coordinate with other OS subsystem.
  - Use 1st unused bit to maintain backup of **dirty bit** information for other OS subsystems in case LDT consume and clears.
  - Use 2nd unused bit to maintain backup of **dirty bit** information for LDT in case other subsystem consume and clears.

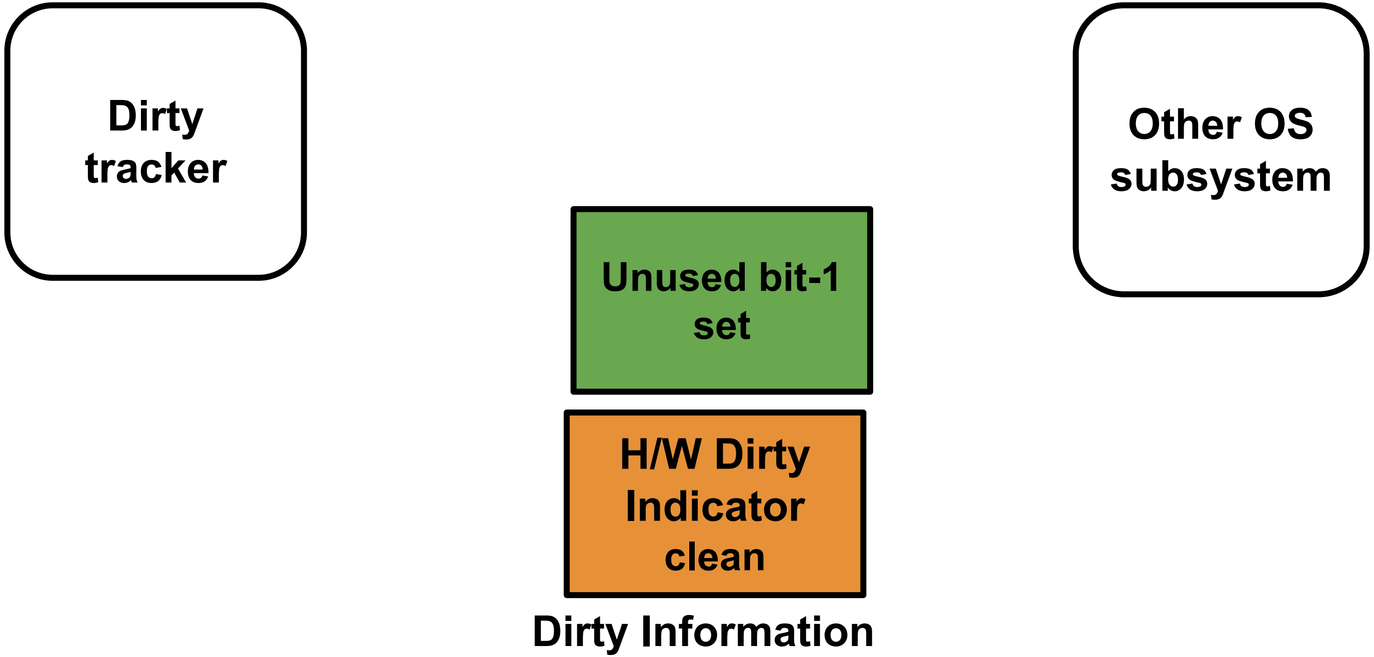
LDT: Handle interaction with OS subsystems

Scenario : 1

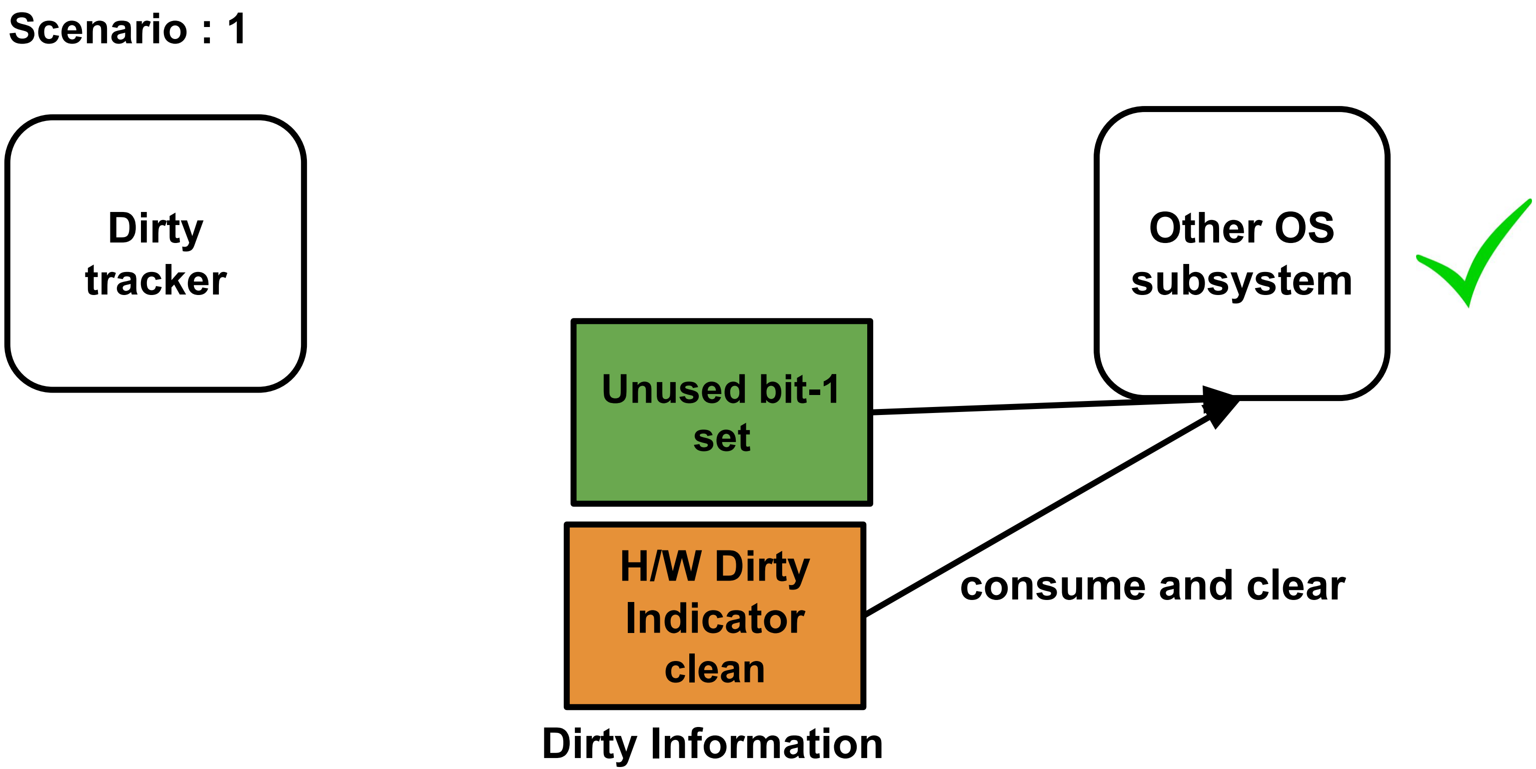


LDT: Handle interaction with OS subsystems

Scenario : 1

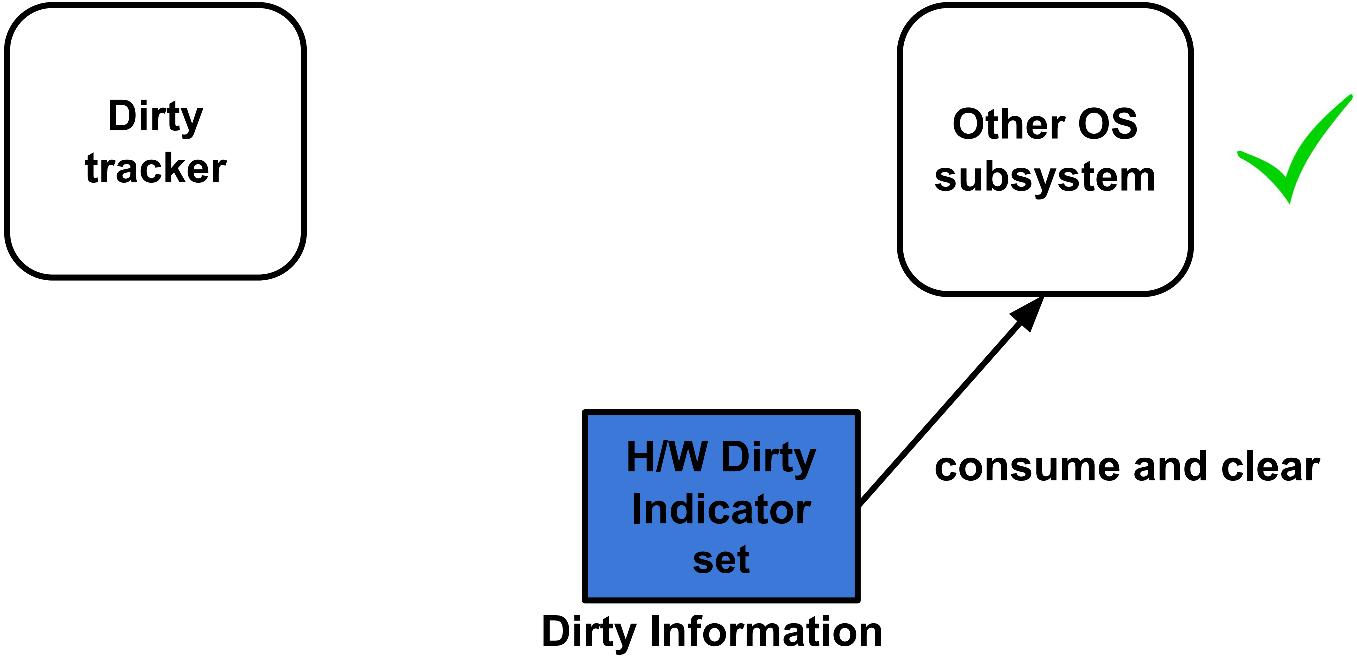


LDT: Handle interaction with OS subsystems



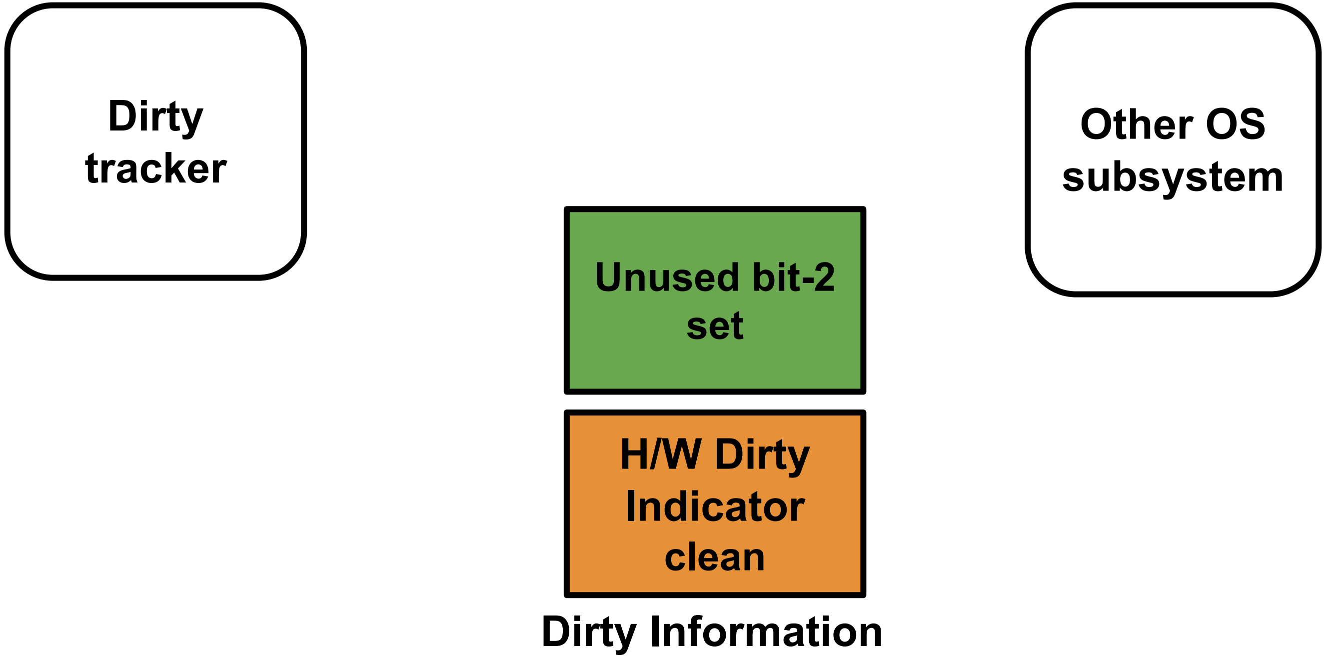
LDT: Handle interaction with OS subsystems

Scenario : 2



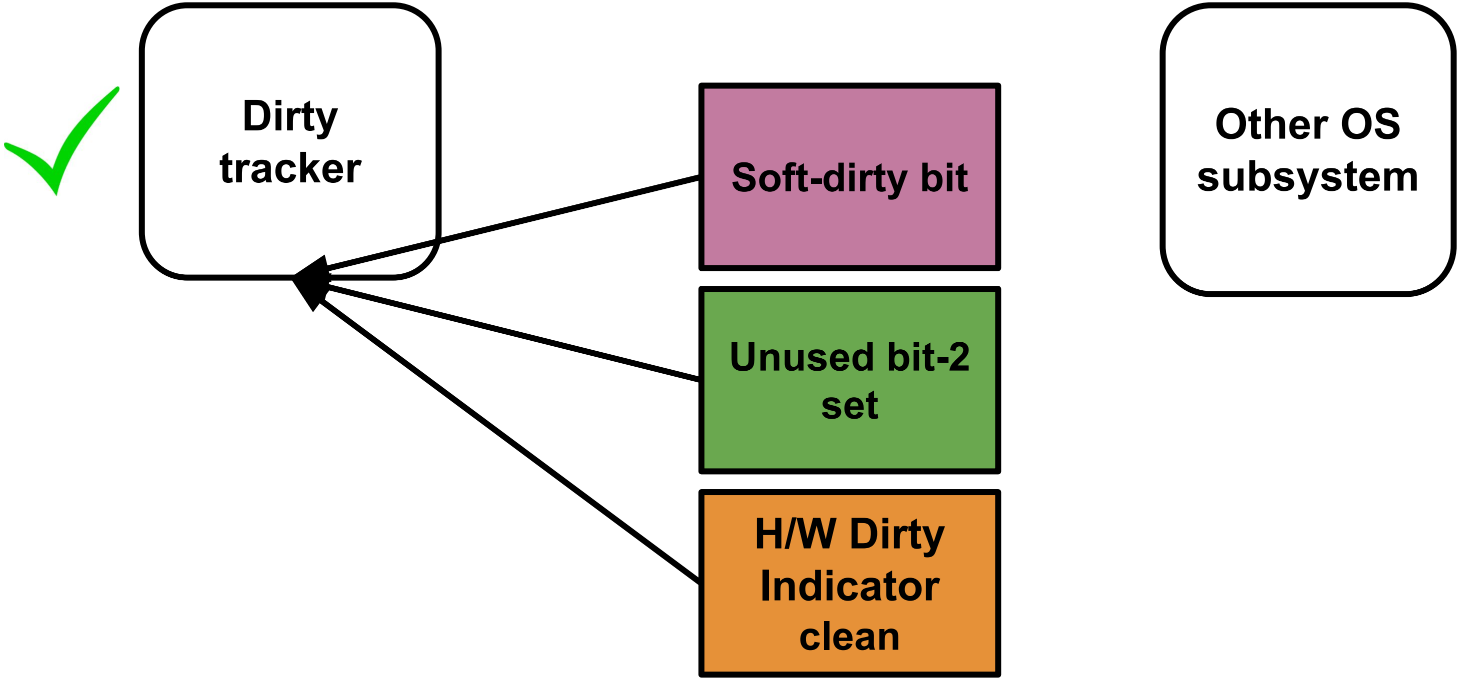
LDT: Handle interaction with OS subsystems

Scenario : 2



LDT: Handle interaction with OS subsystems

Scenario : 2



Dirty Information

LDT: Interface to read dirty track information

- Existing dirty tracking interface passes whole translation table entries to userspace.

- LDT dirty tracking interface passes only modified virtual address information to userspace.

We implemented LDT in linux kernel version 5.5.10

LDT Results

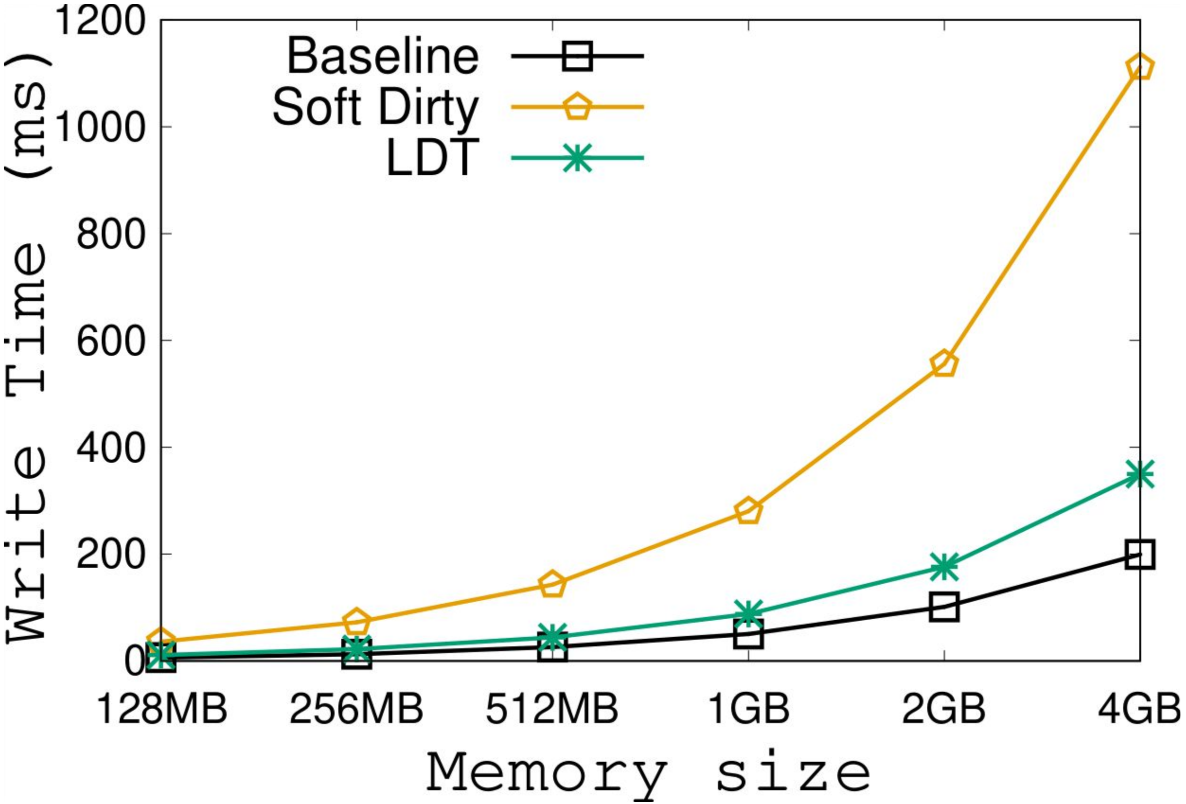
LDT: Correctness checking

- Using micro-benchmarks compared page dirty information given by LDT interface with soft-dirty interface.
- Performed overnight tests with Redis to confirm that LDT is not introducing any kernel issues (assert failures, crashes etc).
- Extreme memory pressure scenarios created using Redis to introduce swapping.
- Performed iterative migration of a container hosting Redis. Docker container is restored correctly and starts serving requests normally after restore.

Evaluation: System Specifications

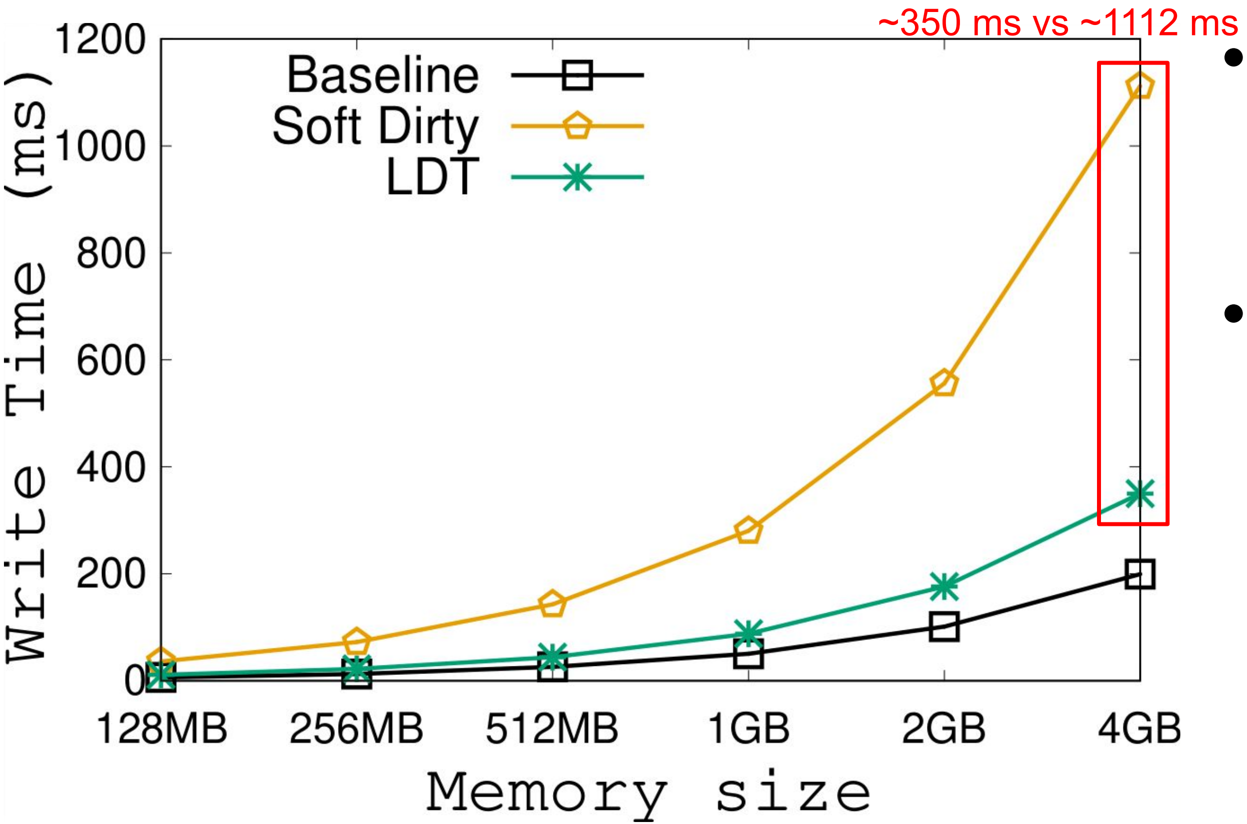
CPU	Intel i7-4770 CPU @ 3.40GHz
L1-D/I	32 KB (8 way)
L2	256 KB (8 way)
L3	8 MB (16 way)
DRAM	16 GB
Distribution	Ubuntu 18.04.3 LTS
Linux Kernel	5.5.10

Evaluation: Write only scenario



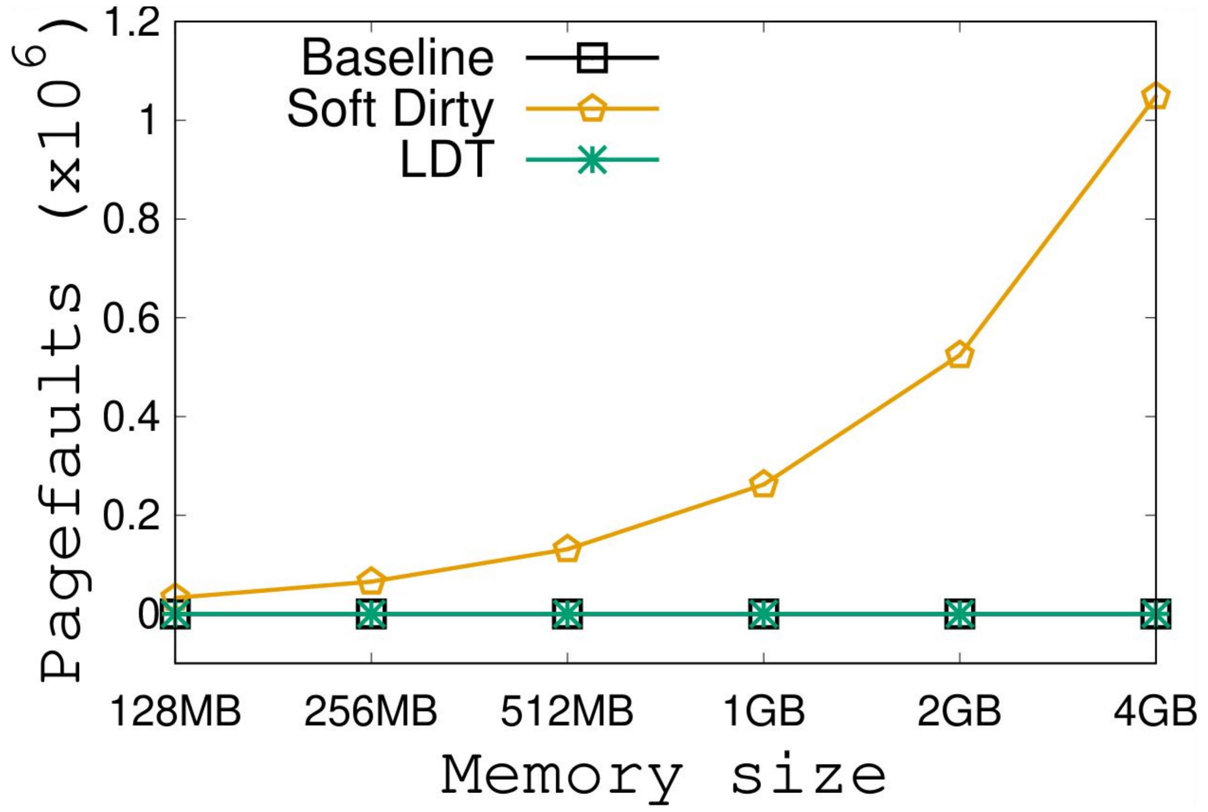
- Workload with different working set sizes where 4096 bytes of each page are written

Evaluation: Write only scenario



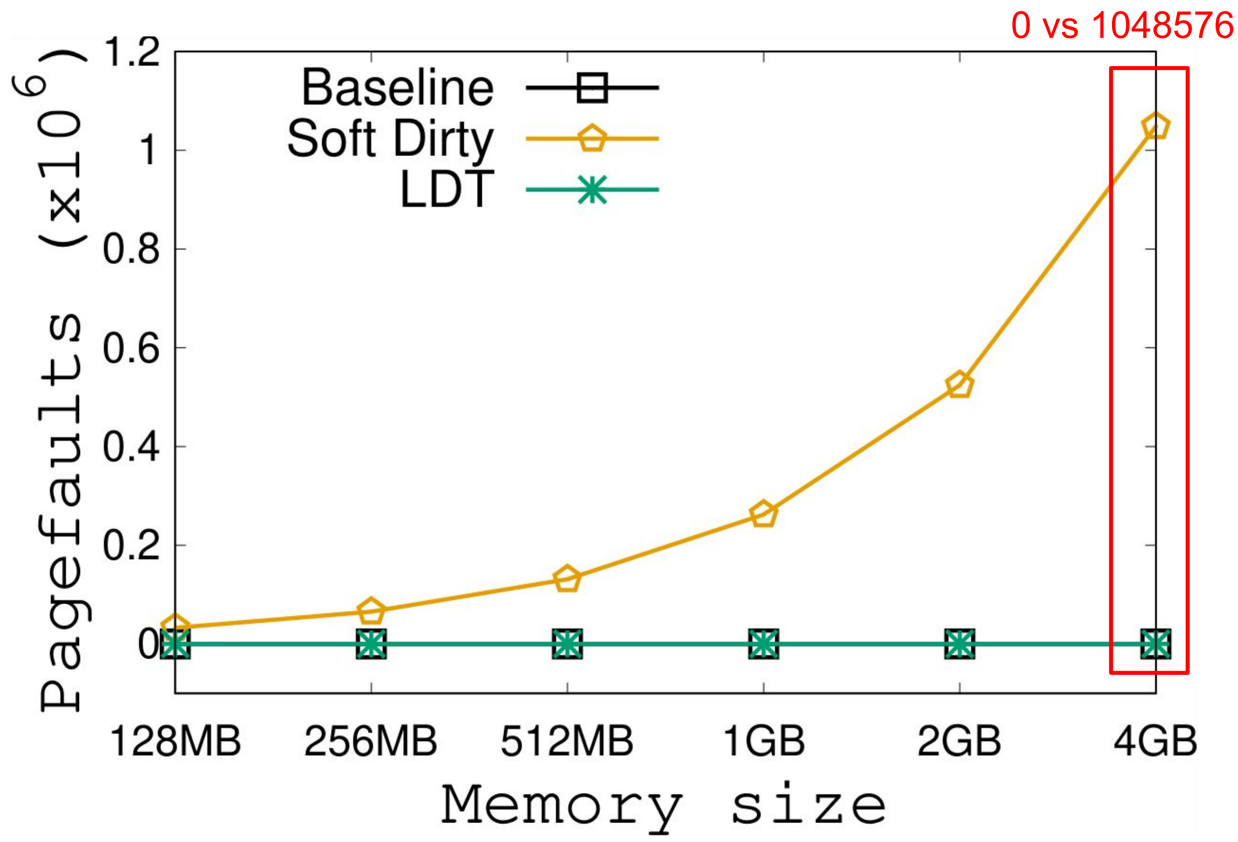
- Workload with different working set sizes where 4096 bytes of each page are written
- Soft Dirty approach takes largest amount of time to complete the write operation

Evaluation: Write only scenario



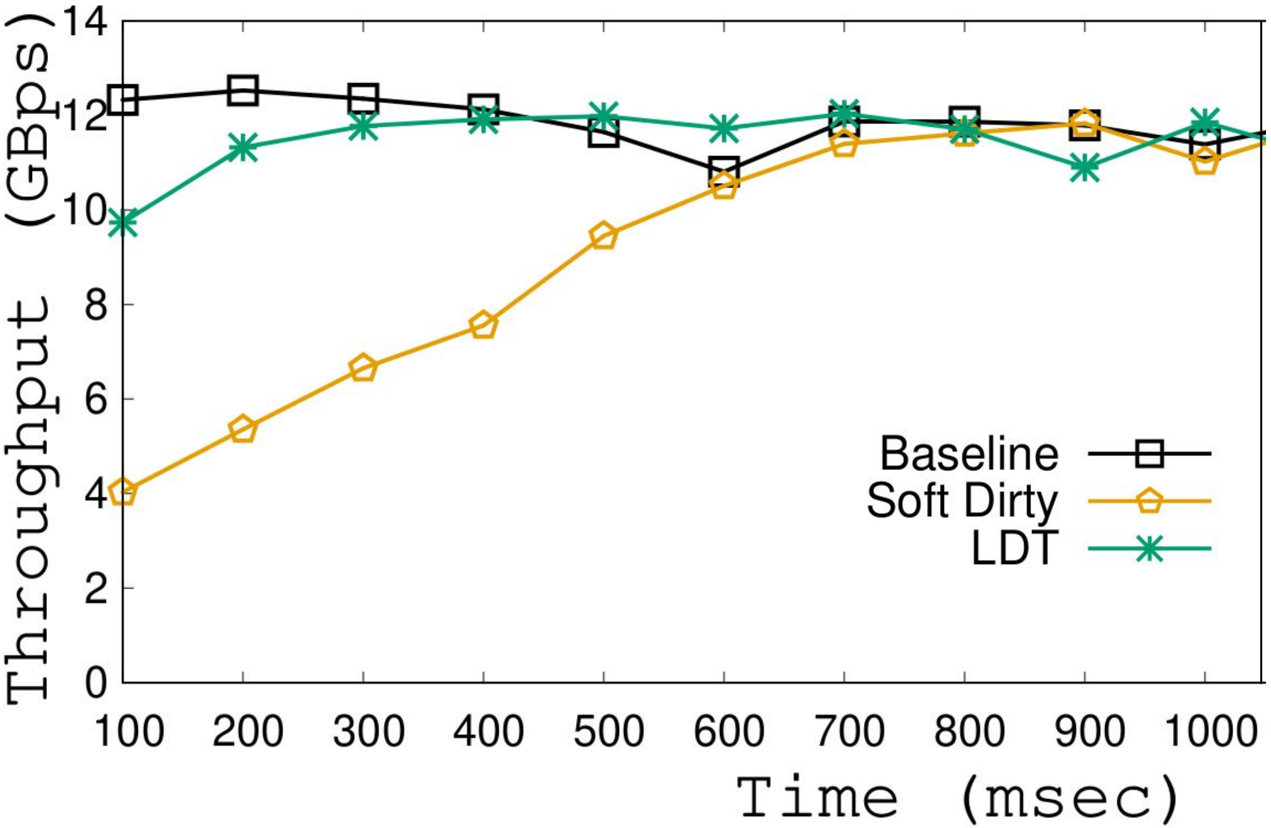
- Workload with different working set sizes where 4096 bytes of each page are written
- Soft Dirty approach takes largest amount of time to complete the write operation
- Bad performance of soft dirty approach can be attributed to page faults

Evaluation: Write only scenario



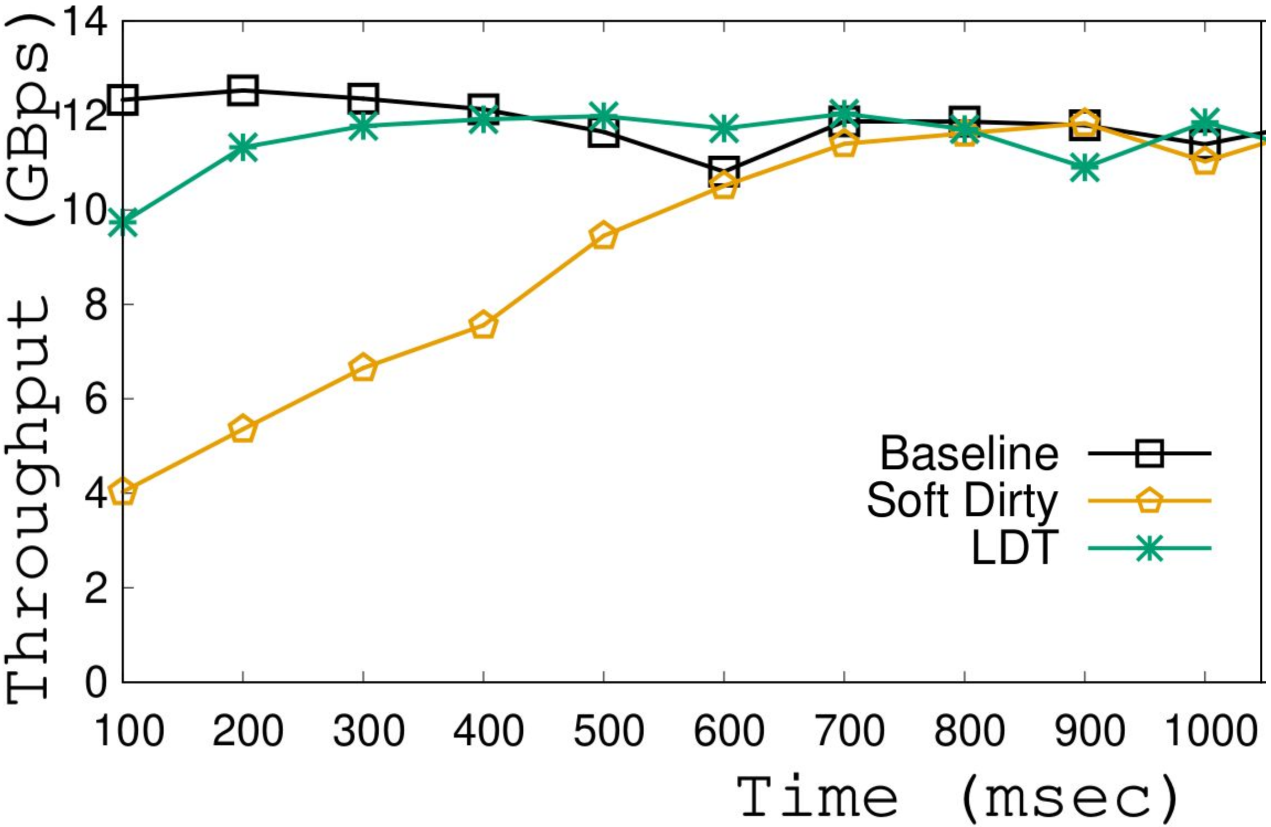
- Workload with different working set sizes where 4096 bytes of each page are written
- Soft Dirty approach takes largest amount of time to complete the write operation
- Bad performance of soft dirty approach can be attributed to page faults

Evaluation: Throughput under read-write intensity



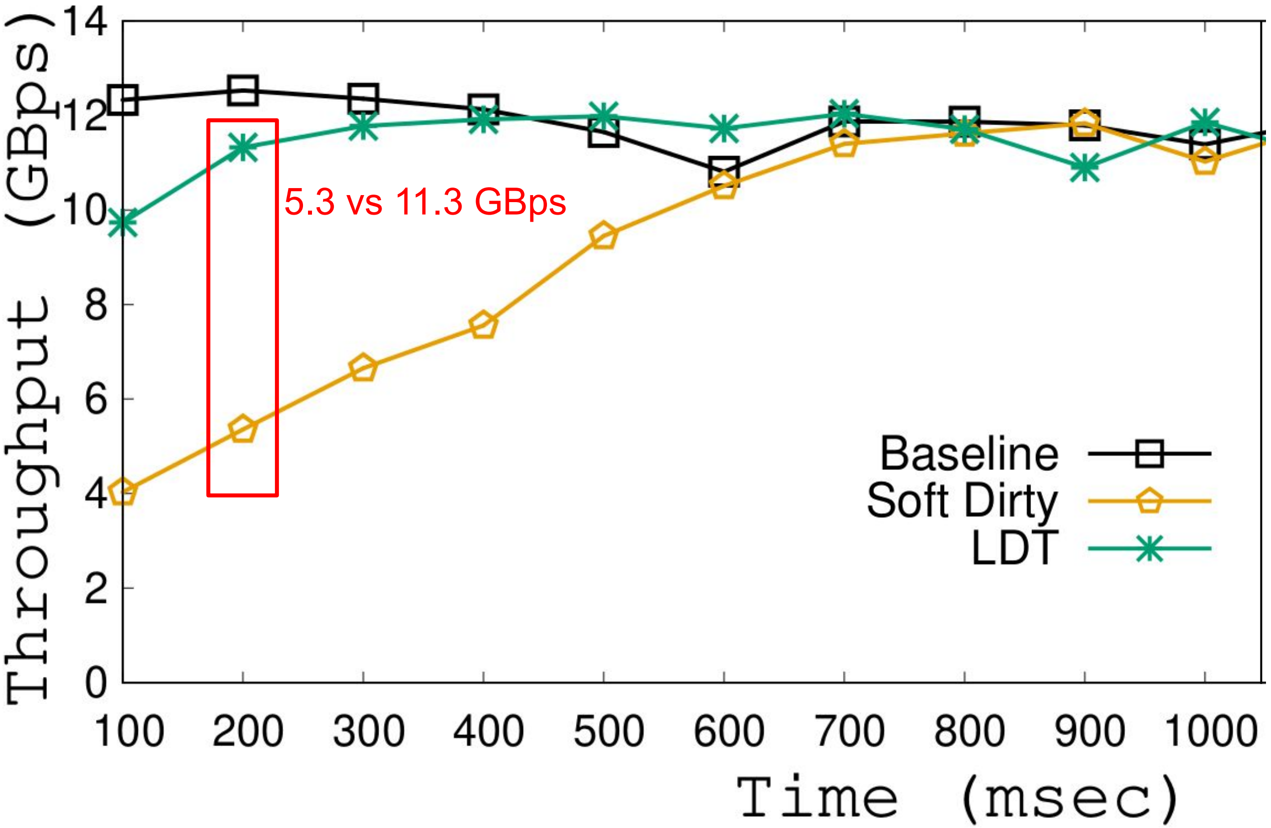
- Under 25% read, 75% write workload, throughput every 100ms

Evaluation: Throughput under read-write intensity



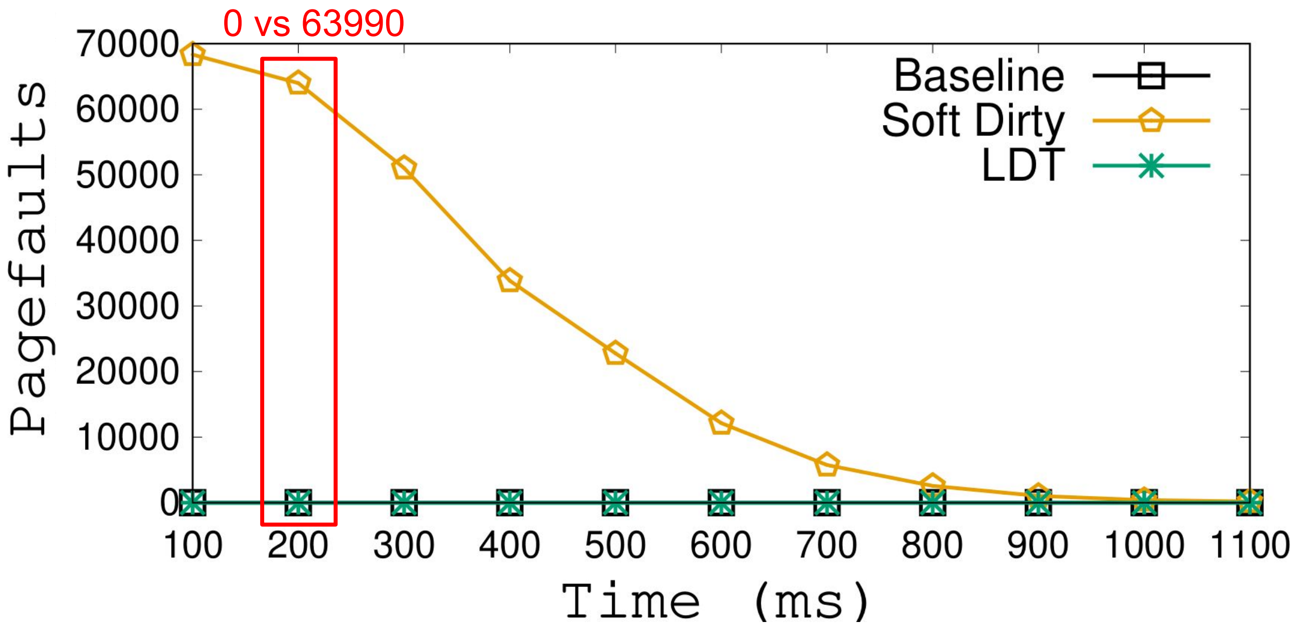
- Under 25% read, 75% write workload, throughput every 100ms
- During every read/write operation, 4096 bytes are consumed/written

Evaluation: Throughput under read-write intensity



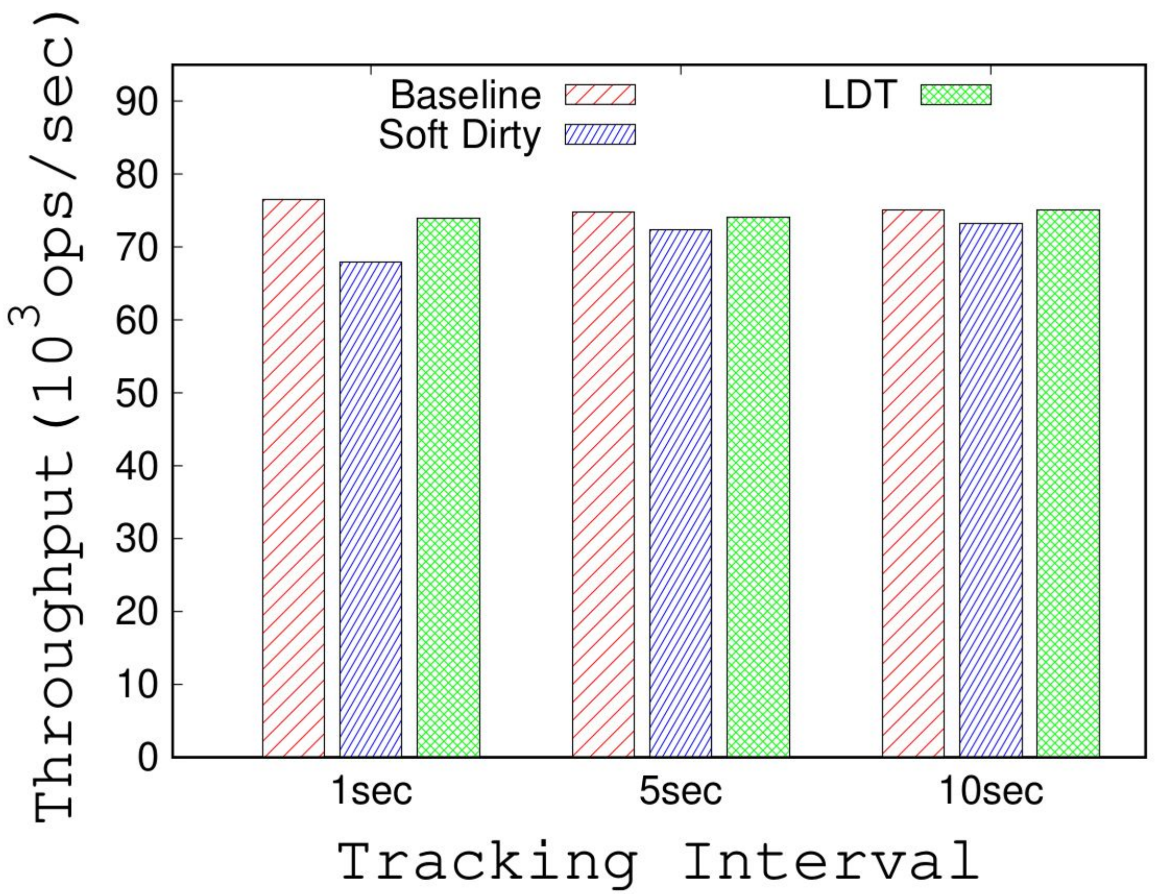
- Under 25% read, 75% write workload, throughput every 100ms
- During every read/write operation, 4096 bytes are consumed/written
- LDT and Baseline throughput is around ~3x more than soft dirty during initial stage of experiment due to page faults

Evaluation: Throughput under read-write intensity



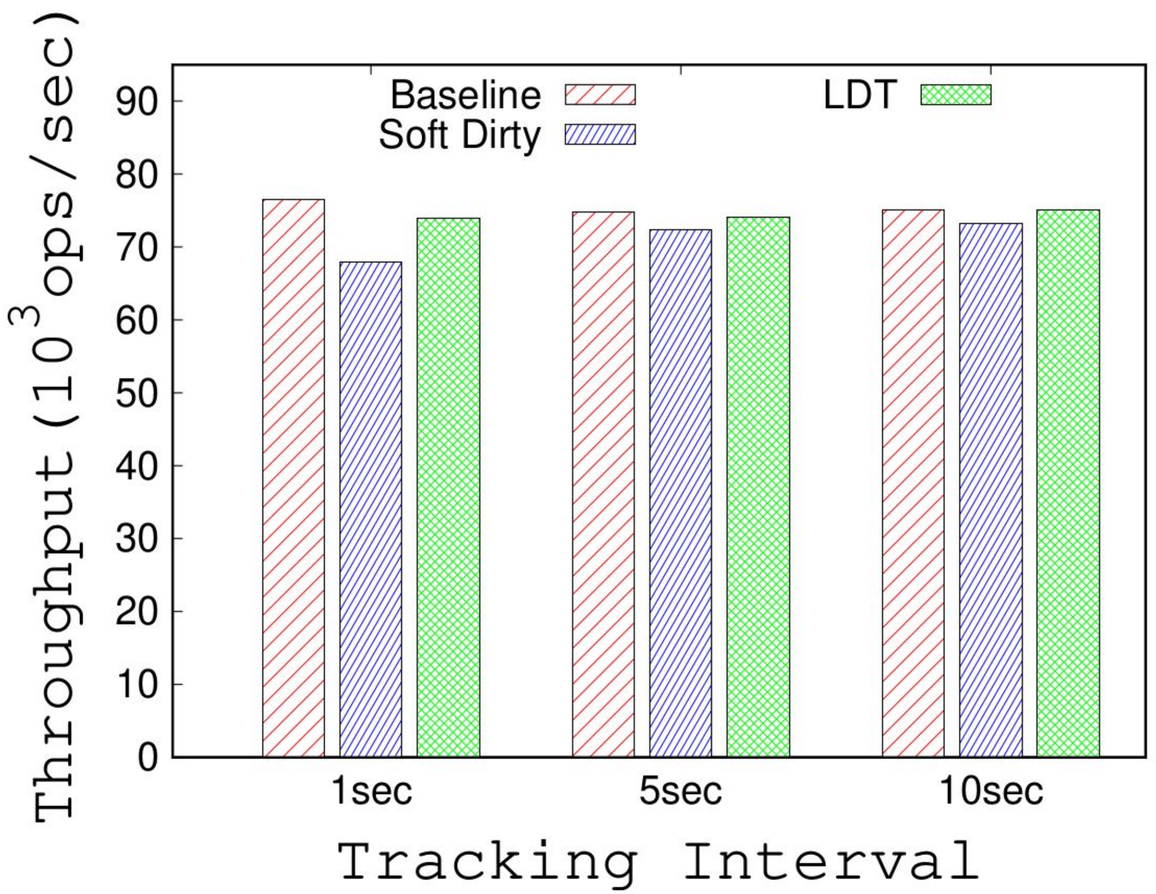
- Under 25% read, 75% write workload, throughput every 100ms
- During every read/write operation, 4096 bytes are consumed/written
- LDT and Baseline throughput is around ~3x more than soft dirty during initial stage of experiment due to page faults

Evaluation: Dirty tracking with Redis benchmark



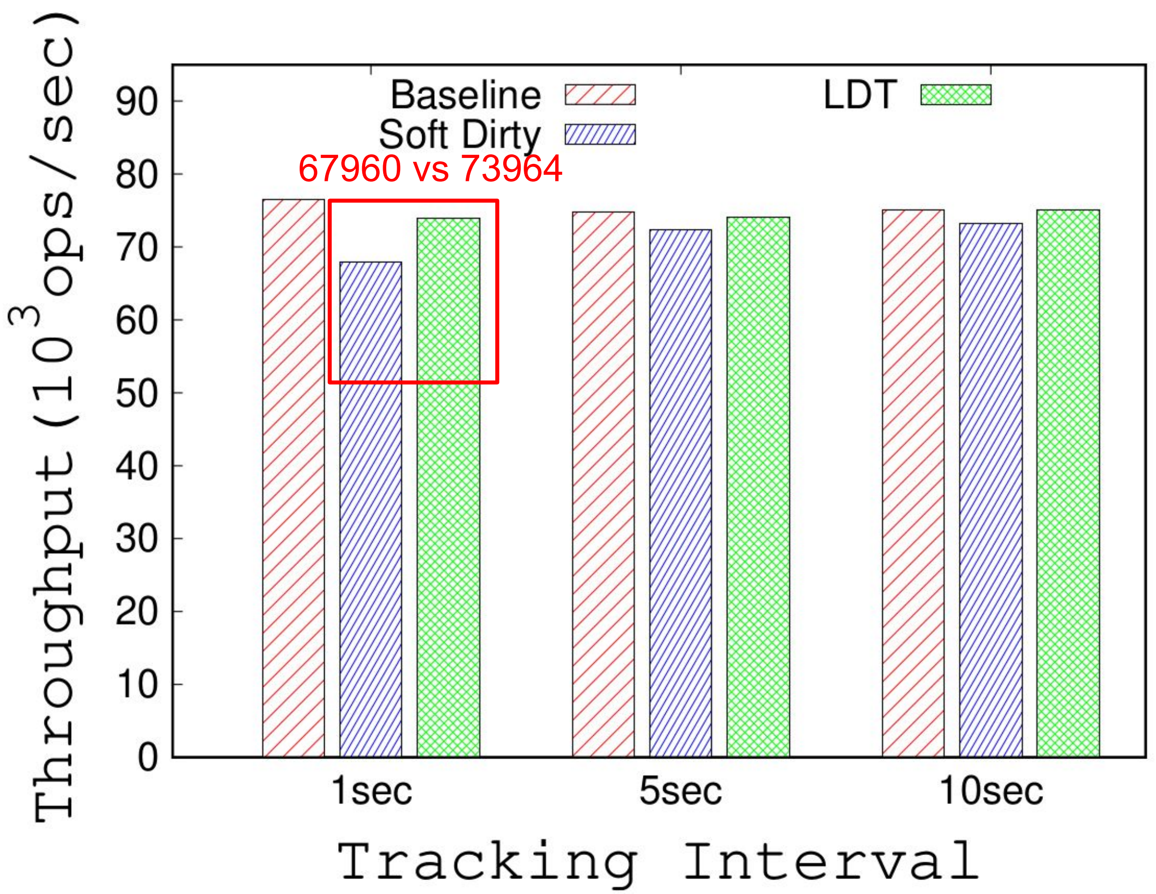
- We ran redis server and then we ran YCSB to perform read/write operations on this redis server

Evaluation: Dirty tracking with Redis benchmark



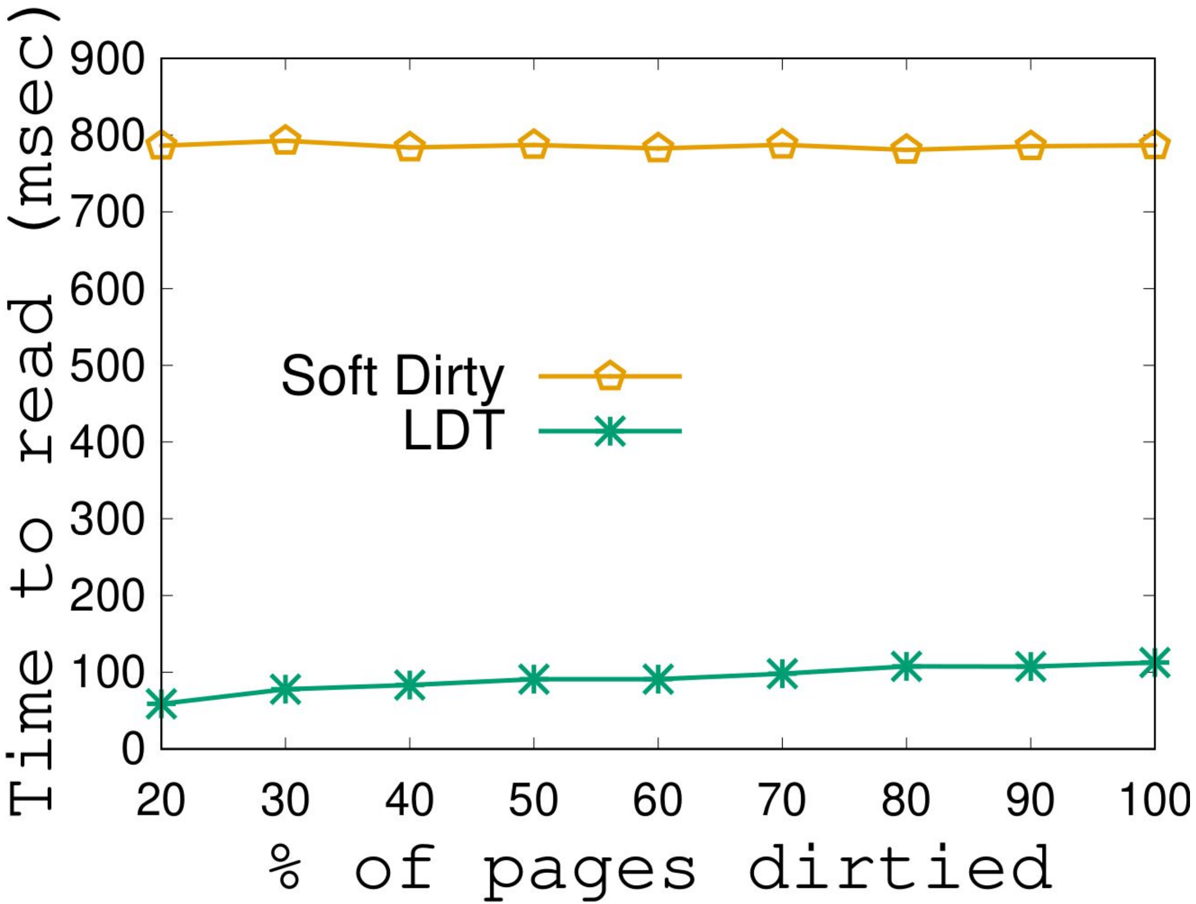
- We ran redis server and then we ran YCSB to perform read/write operations on this redis server
- Dirty tracking occurs every ‘x‘ seconds (1,5,10 seconds)

Evaluation: Dirty tracking with Redis benchmark



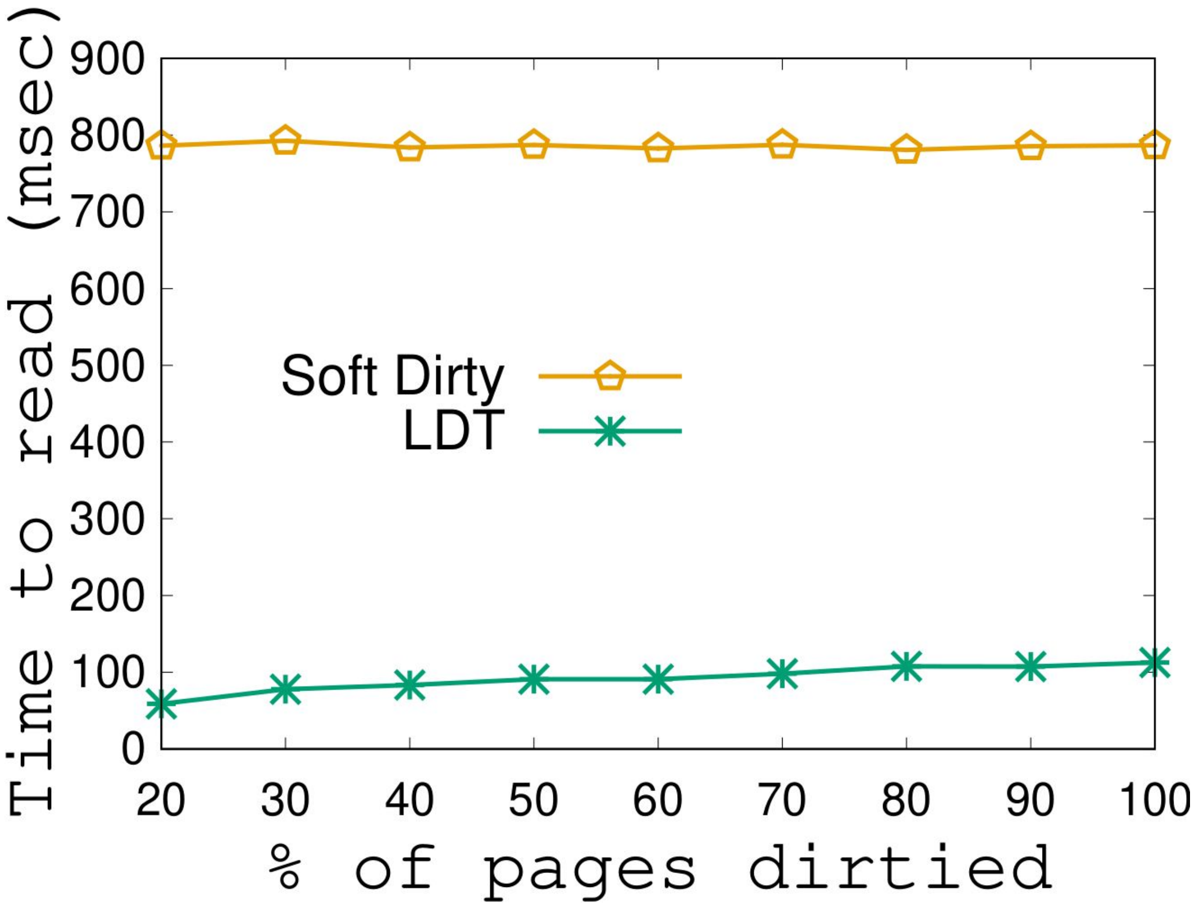
- We ran redis server and then we ran YCSB to perform read/write operations on this redis server
- Dirty tracking occurs every ‘x‘ seconds (1,5,10 seconds)
- Baseline gives best throughput, LDT throughput is close to baseline. Soft dirty incurs worse throughput

Evaluation: Time to read dirtied page information



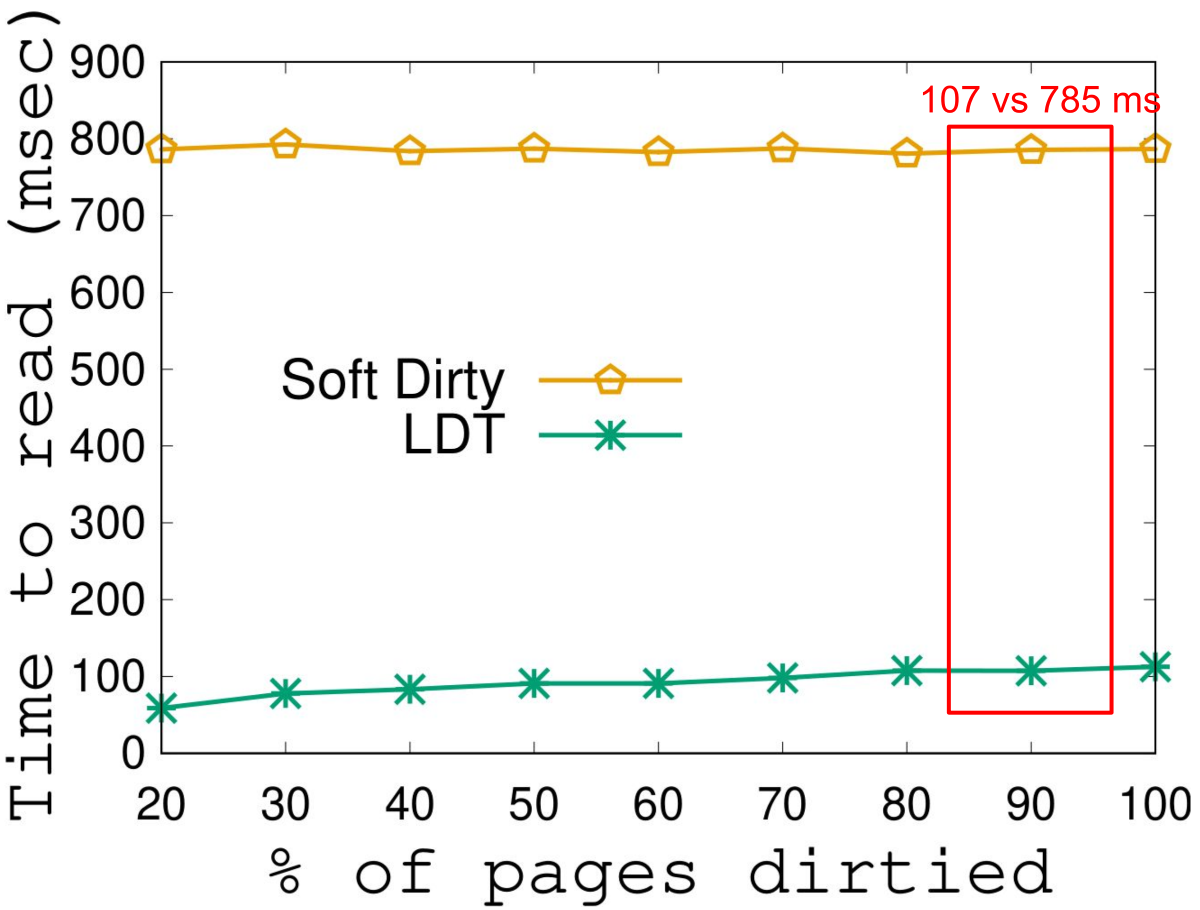
- In this experiment, we write to x% (10%, 20%, .. 100%) of a 1GB mmapped region

Evaluation: Time to read dirtied page information



- In this experiment, we write to x% (10%, 20%, .. 100%) of a 1GB mmapped region
- After that, soft dirty interface/LDT interface is called to read the info about dirtied pages

Evaluation: Time to read dirtied page information



- In this experiment, we write to x% (10%, 20%, .. 100%) of a 1GB mmapped region
- After that, soft dirty interface/LDT interface is called to read the info about dirtied pages
- Soft dirty takes more time because it reports dirty status for entire address space

Conclusion

- LDT enables efficient process migration through lightweight memory dirty tracking.

Conclusion

- LDT enables efficient process migration through lightweight memory dirty tracking.

- LDT provided ~8% throughput improvement over state of the art dirty tracking for Redis.

Conclusion

- LDT enables efficient process migration through lightweight memory dirty tracking.

- LDT provided ~8% throughput improvement over state of the art dirty tracking for Redis.

- LDT showed ~2.4x improvement over state of the art dirty tracking for a workload with 75% writes.

Questions?