# Advanced Computer Systems

*Theory Assignment 1*

University of Copenhagen

Julian Sonne Westh Wulff <bkx655@alumni.ku.dk>

Toke Emil Heldbo Reines <bkx591@alumni.ku.dk>

December 9, 2021

## 1 Techniques for Performance

### 1.1 The difference between dallying and batching.

The difference between dallying and batching is that the former purposely delays requests and the later groups multiple requests to run them at once. The memory layer abstraction in computers can be seen as an example of dallying as writes to the disk are delayed by saving them in memory until they are either overwritten or finally saved to the disk. When we write to the disk we may also use batching to improve performance if the write requests are for the same disk block.

### 1.2 Concurrency and latency relationship

Concurrency can reduce latency by parallelizing one or more stages of a request. This method of optimizing a system often leads to a positive effect if the introduced overheads relative size is not to big. Besides the overhead of parallelizing a stage there is also a risk of introducing race conditions and deadlocks into the system, thus it takes some carefull implementation. In some cases concurrency can actually have an negative effect on latency, an example of this in some DBMS systems that uses locking to interleave request. When the number active transactions (e.g. the amount of parallelism) increases the risk of thrashing increase in which case the effect of increasing concurrency has a negative effect and throughput and latency go down.

### 1.3 Fast path optimization

Caching is one example of the fast path optimization technique. As we with caching split the pipeline into to paths, the slow path for reading data that is not cached and the fast path for reading data that is cached. Using the fast path optimization technique has its trade-offs as increasing the speed of one path often reduces the speed of the other path. Furthermore introducing two paths instead of one also increases the complexity of the system.

## 2 Fundamental Abstraction

### 2.1 Name mapping scheme

As name mapping scheme for the proposed memory abstraction we would choose a static dictionary using a two-level hashing technique[1], where the single address spaces is hashed to the location of one the $K$ machines, and the second level of the hashing algorithm hashes down to an entry in a page map of the specific machine. This name mapping scheme would be very effective as looking up in the dictionary could be performed in $O(1)$ time. Furthermore, the top level of the name scheme would be controlled from on central component such that clients would immediately send the `READ/WRITE` request to the correct machine. This design

[1]This data structure was introduced to us by Mikkel Thorup in the RAD course and is described in Rajeev Motwani and Prabhakar Raghavan, *Randomized Algorithms*. 1995. Sectoin 8.5

wouldn't be very tolerable towards adding or removing machines as that would require to recalculate the individual hashing algorithms and remap all of the memory. Though the time to construct is expected at to run in $O(n)$ time, where n is the number of keys, which in the case is our single address space.

## 2.2 READ/WRITE

In our proposed abstraction the pseudocode for the `READ` operation is shown in Figure 1. First the read function gets the machine address through the naming scheme described in subsection 2.1 that hashes $n$ down to one of $K$ values (eg. if $K = 4$, then $n$ is hashed down into one of the values in the range $\{0, 1, 2, 3\}$). This hashed value then represent en entry in a machine address table from which the specific machine address is returned. Using this address the read function then sends a read request to the specific machine for the given $n$. When a machine receives a read request it then runs the second-level hashing algorithm to find the entry in that machines page table and returns data.

```
READ(n):
    machineAddress <- getMachineAddress(n)
    value <- readFromMachine(n, machineAddress)
    return value
```

Figure 1: Pseudocode for the `READ` operation of our memory abstraction

The pseudocode for the `WRITE` operation is shown in Figure 2 and follows the same basic structure and name translation technique as the `READ` operation described above. The only difference here being that that value being written is also send to machine in the request. When the machine commits or aborts the write operation the client then receives a response which tells the client if anything went wrong or if it was written succesfully.

```
WRITE(n, value):
    machineAddress <- getMachineAddress(n)
    response <- writeToMachine(n, value, machineAddress)
    return response
```

Figure 2: Pseudocode for the `READ` operation of our memory abstraction

## 2.3 Atomicity

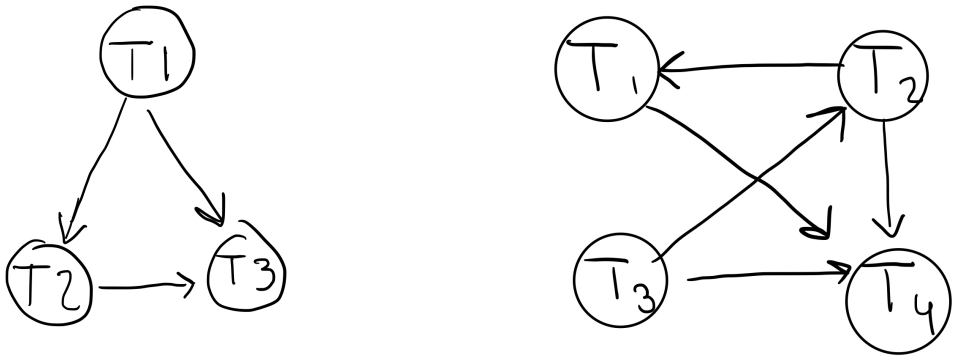
To ensure that clients does not read any dirty data or data that has not committed the `READ` and `WRITE` operations of our memory abstraction should have the before-or-after abstraction. To achieve this one could implement some sort of a concurrency control. This could be done either with a Locking model such strict 2PL, in which case deadlock prevention or deadlock detection would also have to be implemented. Another approach would be an optimistic concurrency control model such as the Kung-Robinson Model.

## 2.4 Dynamic naming

As mentioned in in subsection 2.1 the naming scheme uses a static dictionary which in it nature is not suited for dynamic changes like machines entering and leaving the system. If a new machine was to enter the system or machine was to leave the system it would require the system to generate the dictionary anew. Though this is possible to do in $O(n)$ time there a no guarantees that the new dictionaries hashing functions would still hash to the same entries in machine address table and the individual page tables. Thus these tables would have to updated according to the new hash functions. To do this in a fault tolerating way all reads and writes from clients would have to be postponed until the new dictionary with the accompanying machine address table and pages tables was updated acordingly.

## 3 Serializability & Locking

By looking at the precedence graphs in Figure 3a and Figure 3b we see that both schedule 1 and schedule 2 is conflict serializable, as none of the graphs has any cycles within them.




(a) Schedule 1 precedence graph

(b) Schedule 2 precedence graph

Figure 3: Precedence graphs of the two schedules

It is possible to make schedule 1 not conflict serializable by changing just one a single action type. If we look at the first action `R(X)` in `T2` in schedule 1 we see that this action is not conflicting with any other action, but if we change the first action `R(X)` in `T1` these would be conflicting which would give us the precedence graph shown Figure 4. As this graph contains a cycle this would make schedule not conflict serializable

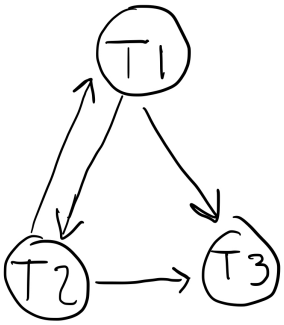



Figure 4: Schedule 1 precedence graph

As "strict 2PL ensures that the precedence graph for any schedule that it allows is acyclic" [1] and the precedence graphs in Figure 3 is acyclic the two schedules could have been generated by a scheduler using strict 2PL. Thus we can inject the lock operations in accordance with the stric 2PL rules in the schedules as shown in Table 1.

```
T1:              S(X), R(X)  X(Z), W(Z)              X(Y), W(Y)  C
T2:  S(X), R(X)                                                          X(Y), W(Y)  S(Y), R(Y)  C
T3:                                      S(Z), R(Z)                X(X), W(X)  C
```

(a) Schedule 1

```
T1:  S(X), R(X)                                                                  X(Y), W(Y), C
T2:             S(X), R(X) S(Y), R(Y)                                  X(Z), W(Z), C
T3:                                    S(X), R(X) S(Z), R(Z)                                        X(U), W(U), C
T4:                                                        X(X), R(X), W(X), C
```

(b) Schedule 2

Table 1: Schedule 1 and 2 with injected lock operations in accordance with strict 2PL rules

# 4 Optimistic Concurrency Control

## Scenario 1

In this scenario T3 is not allowed to commit. First we check that T1 allows T3 to commit, since T1 completes before T3 starts *test 1* is successful and we make a test between T2 and T3. As T2 completes before T3 begins with its write phase we will use *test 2* where we check that the intersection between the write-set of T2 and the read-set of T3 is empty to allow committing. In this case we see that $T2_{WS}\{4,5\} \cap T3_{RS}\{3,4,6\} = \{4\} \neq \varnothing$, and therefore T3 is not committed

## Scenario 2

In this scenario T3 is not allowed to commit. First we check that T1 allows T3 to commit, since T1 completes before T3 begins with its write phase, we use *test 2* and see that $\{3\} \cap \{5,6,7\} = \varnothing$. Thus we go on and check whether T2 also allows T3 to commit. Here we use *test 3* as T2 completes read phase before T3 does. Thus we check that the intersection between the write-set of T2 and the read-set of T3 is empty and that the write-set of T2 and the write-set of T3 is empty. Thus we have $T2_{WS}\{3,8\} \cap T3_{RS}\{4,5,6,7\} = \varnothing$ and $T2_{W}S\{3,8\} \cap T3_{W}S\{3\} = \{3\} \neq \varnothing$. As the last one fails T3 is not allowed to commit.

## Scenario 3

In this scenario we follow the same structure as in Scenario 2, as T1 completes before T3 begins with its write phase and T2 completes read phase before T3 does. Here we see that $T1_{WS}\{4\} \cap T2_{RS}\{6,7,8\} = \varnothing$ and go with test 3 between T2 and T3. Here we see that $T2_{WS}\{6\} \cap T3_{RS}\{2,3,5,7,8\} = \varnothing$ and that $T2_{WS}\{6\} \cap T3_{WS}\{7,8\} = \varnothing$ and thus T3 is allowed to commit.

# References


[1] R. Ramakrishnan and J. Gehrke. *Database Management Systems. Third Edition.*, chapter 16–17, pages 519–544, and 549–575. McGraw-Hill, 2003. ISBN 978-3-030-47146-0.