The PCL Theorem. Transactions cannot be Parallel, Consistent and Live.

Victor Bushkov
EPFL, IC, LPD
victor.bushkov@epfl.ch

Dmytro Dziuma
FORTH-ICS
ddziuma@ics.forth.gr

Panagiota Fatourou*
University of Crete &
FORTH-ICS
faturu@csd.uoc.gr

Rachid Guerraoui
EPFL, IC, LPD
rachid.guerraoui@epfl.ch

ABSTRACT

We show that it is impossible to design a transactional memory system which ensures **parallelism**, i.e. transactions do not need to synchronize unless they access the same application objects, while ensuring very little **consistency**, i.e. a consistency condition, called *weak adaptive consistency*, introduced here and which is weaker than snapshot isolation, processor consistency, and any other consistency condition stronger than them (such as opacity, serializability, causal serializability, etc.), and very little **liveness**, i.e. that transactions eventually commit if they run solo.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords

transactional memory; disjoint-access-parallelism; snapshot isolation; processor consistency; weak adaptive consistency; obstruction-freedom; lower bounds; universal constructions

1. INTRODUCTION

The paradigm of *transactions* [20, 26, 35] is appealing for its simplicity but implementing it efficiently is challenging. Ideally a transactional system should not introduce any contention between transactions beyond that inherently due to the actual code of the transactions. In other words, if two transactions access disjoint sets of data items, then none of these transactions should delay the other one, i.e., these transactions should not *contend* on any base object. This requirement has been called *strict disjoint-access-parallelism* [2, 22]. *Base objects* are low-level objects, which typically provide atomic *primitives* like *read/write*, *load linked/store conditional*, *compare-and-swap*, used to implement transactional systems. Two transactions *contend* on some base object if both access that object during their executions and one of them performs a *non-trivial* operation on that object, i.e. an operation which updates its state.

Strict disjoint access parallelism can be ensured by blocking transactional memory (TM) systems; indeed, TL [14], a lock-based TM algorithm, ensures strict disjoint-access-parallelism and strict serializability [30]. It was shown in [21] that a strictly disjoint-access-parallel TM algorithm cannot ensure both obstruction-freedom (i.e. a weak non-blocking liveness condition) and serializability (i.e. a consistency condition weaker than strict serializability). Specifically, *obstruction-freedom* [25] ensures that a transaction is aborted only if step contention is encountered during the course of its execution. *Serializability* [30] ensures that, in any execution, all committed transactions (and some that have not completed yet) execute like in a legal sequential execution.

In this paper, we study the following question: can we ensure strict disjoint-access-parallelism and obstruction freedom if we weaken safety? In other words, is serializability indeed a major factor against strong parallelism? We focus on a new weak consistency condition that we introduce in this paper, called *weak adaptive consistency*. Weak adaptive consistency is weaker than (a weak variant of) snapshot isolation [10] and processor consistency [19]. Thus, it is weaker than serializability, causal serializability and all other consistency conditions that are stronger than processor consistency (or snapshot isolation or even the union of both). Our PCL theorem states that even with weak safety and weak liveness, the described task is still impossible: specifically, it is not possible to implement a transactional memory system which ensures strict disjoint-access-parallelism (Parallelism), weak adaptive consistency (Consistency), and obstruction-freedom (Liveness).

Weak adaptive consistency weakens snapshot isolation in two ways: (1) each process is allowed to have its own sequential view and (2) it is possible to partition the transactions of an execution in such a way that each set of transactions in the partition satisfies either snapshot isolation or processor consistency. *Snapshot isolation* [10] requires that transactions should be executed as if every read operation observes a consistent snapshot of the memory that was taken when

*Currently with École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, as an EcoCloud visiting professor.

SPAA '14 Prague, Czech Republic

http://dx.doi.org/10.1145/2612669.2612690.

the transaction started. To make our result stronger, in our definition of snapshot isolation, we do not require the extra constraint (met in the literature [10, 16, 33] for snapshot isolation) that from two concurrent transactions writing to the same data item, only one can commit, and we do not impose any restriction on the value that a *read* on some data item x by a transaction T may return if T has written x before invoking this read. *Processor consistency* [19, 3] allows each process to have its own sequential view which should respect the process-order of writes, additionally it requires writes to the same data item appear in the same order in all sequential views. Processor consistency is stronger than *PRAM consistency* [28, 3], which does not require writes to the same data item to appear in the same order in all sequential views, but weaker than *causal serializability* [32], which requires each sequential view to respect a relation on transactions, called *causality relation*.

The proof of our impossibility result is based on indistinguishability arguments. The main difficulty comes from the fact that the read operations of a transaction do not have to be serialized at the same point as its write operations. Basically, snapshot isolation and especially weak adaptive consistency allow more executions to be correct and it is much harder to construct an execution which violates it. We end up constructing two legal executions where a transaction must read the same values for data items. We then prove that in one of these two executions this is not the case.

This paper is structured as follows. Section 2 gives an overview of the related work. Section 3 gives a system model and all necessary definitions. Section 4 gives the PCL theorem and its proof. Section 5 presents concluding remarks.

2. RELATED WORK

The notion of disjoint-access-parallelism appears in the literature [2, 8, 15, 22, 27, 31] in many flavors. Disjoint-access-parallelism was first introduced in [27] through the notion of conflicting transactions. Later variants [2, 8, 15] employed the concept of a conflict graph. A *conflict graph* is a graph whose nodes represent transactions (or operations) performed in an execution interval α (i.e. the execution interval of those transactions overlap with α) and an edge exists between two nodes if the corresponding transactions (operations) access the same data item in α (i.e they *conflict* in α). In most of these definitions, *disjoint-access-parallelism* requires any two transactions to contend on a base object only if there is a path in the conflict graph of the minimal execution interval that contains the execution intervals of both transactions such that every two consecutive transactions in the path conflict. In [2, 5, 6, 27], additional constraints are placed on the length of the path in the conflict graph, resulting in what is known as *d-local contention property*, where d is an upper bound on the length of the path. In [27], where disjoint-access-parallelism originally appeared, an additional constraint on the step complexity of each operation was provided in the definition. Stronger versions of disjoint-access-parallelism usually result in more parallelism and therefore they are highly desirable when designing TM implementations. Weaker versions of disjoint-access-parallelism may result in less parallelism but are easier to implement.

Attiya *et al.* [8] proved that no disjoint-access-parallel TM implementation can support wait-free and *invisible* read-only transactions. A *read-only* transaction does not perform writes on data items; an *invisible* transaction does not perform non-trivial operations on base objects when reading data items. The variant of disjoint-access-parallelism considered in [8] stipulates that processes executing two transactions concurrently contend on a base object only if there is a path between the two transactions in the conflict graph. Although our impossibility does not hold for this variant of disjoint-access-parallel, our impossibility result considers a much weaker liveness property and holds even for TM algorithms where read-only transactions are visible.

Recent work [12] proved that, if the TM algorithm does not have access to the code of each transaction, a property similar to *wait-freedom*, called *local progress*, cannot be ensured by any TM algorithm. In [15], it was proved that *wait-freedom* cannot be achieved even if this restriction is abandoned (given that each time a transaction aborts, it restarts its execution), if the TM algorithm ensures strict serializability and a weak version of disjoint-access-parallelism, called *feeble disjoint-access-parallelism*. Thus, one must consider weaker consistency or progress properties as we do here.

Pelerman *et al.* [31] proved that no disjoint-access-parallel TM algorithm can be strictly serializable and MV-permissive. The impossibility result holds under the assumptions that the TM algorithm does not have access to the code of transactions and the code for reading and writing data items terminates within a finite number of steps. Pelerman *et al.* [31] considered the same variant of disjoint-access-parallelism as in [8]. A TM implementation satisfies *MV-permissiveness* if a transaction aborts only if it is a write transaction that conflicts with another write transaction. This impossibility result can be beaten [7] if the stated assumptions do not hold. Our impossibility result holds if the TM ensures just weak snapshot isolation, even if it is MV-permissive.

Several software TM implementations [35, 14, 17, 29, 36, 25] are disjoint-access-parallel: TL [14] ensures strict disjoint-access-parallelism but is blocking since it uses locks; the rest satisfy weaker forms of disjoint-access-parallelism [8]. Among them OSTM [17] is lock-free. The TM in [35] is also lock-free but it has been designed for *static* transactions that access a pre-determined set of memory locations. Aparently, our impossibility result does not contradict these implementations because all of them, except TL, ensure weaker variants of disjoint-access-parallelism and some of them weaker progress as well. Also, our impossibility result does not contradict TL since TL uses locks and consequently does not ensure obstruction-freedom. Linearizable *universal constructions* [23, 24], which ensure some form of disjoint-access-parallelism, are presented in [1, 9, 15, 37]. Barnes [9] implementation is lock-free. The universal construction in [15] ensures wait-freedom when applied to objects that have a bound on the number of data items accessed by each operation they support, and lock-freedom in other cases. Disjoint-access-parallel wait-free universal constructions when each operation accesses a fixed number of predetermined memory locations are provided in [2, 37].

Snapshot isolation was originally introduced as an isolation level for database transactions [10, 16] to increase throughput for long read-only transactions. In TM computing, snapshot isolation has been studied in [4, 13, 33, 34]. An STM algorithm, called SI-STM, which ensures snapshot isolation is presented in [33]. SI-STM employs a global clock mechanism and therefore, it is not disjoint-access-parallel. In [13], static analysis techniques are presented to detect, at compile time, consistency anomalies that may arise when

the TM algorithm satisfies snapshot isolation or other safety properties. Snapshot isolation on TM for message-passing systems has been studied in [4].

Our definition of snapshot isolation is weaker than that defined for database transactions [10] for the following reasons. First, we do not put any constraint on the value returned by any read that occurs after a write to the same data item in the same transaction. Second, we do not place the "first committer wins" rule, i.e. we abandon the requirement to abort one out of two concurrent transactions that are writing to the same data item. By introducing these constraints, we would make our impossibility result weaker.

3. PRELIMINARIES

System. We consider an *asynchronous system* with n processes which communicate by accessing shared base objects. A *base object* provides atomic *primitives* to access or modify its state. The system may support various types of base objects like read/write registers, CAS, etc. A primitive that does not change the state of an object is called *trivial* (otherwise it is called *non-trivial*).

Transactions. *Transactional memory* (TM) employs *transactions* to execute pieces of sequential code in a concurrent environment. Each piece of code contains accesses to pieces of data, called *data items*, that may be accessed by several processes when the code is executed concurrently; so TM should synchronize these accesses. To achieve this, a TM algorithm usually provides a shared representation for each data item by using base objects. A transaction may either *commit*, in which case all its updates become visible to other transactions, or *abort* and then its updates are discarded.

A TM algorithm provides implementations for the routines $x.read()$, which returns a value for x if the operation was successful or A_T if the transaction has to abort, and $x.write(v)$, which writes value v to data item x and returns *ok* if the write was successful or A_T if the transaction has to abort. In addition, a TM algorithm provides implementations for the routines $begin_T$, which is called when a transaction T starts and returns *ok*, $commit_T$, which is called when T tries to commit and returns either C_T (commit) or A_T (abort), and $abort_T$, which aborts T and returns A_T. Each time a transaction calls one of these routines we say that it *invokes* an *operation*; when the execution of the routine completes, a *response* is returned.

Executions and configurations. A *configuration* is a vector with components comprising the state of each process and the state of each base object. In an *initial configuration*, processes and base objects are in initial states. A *step* of a process consists of a single primitive on a single base object, the response to that primitive, and zero or more local operations that are performed after the access and which may cause the internal state of the process to change; each step is executed atomically. Invocations and responses performed by transactions are considered as steps. An *execution* α is a sequence of steps. An execution is *legal* starting from a configuration C if the sequence of steps performed by each process follows the algorithm for that process (starting from its state in C) and, for each base object, the responses to the operations performed on the object are in accordance with its specification (and the state of the object at configuration C). We use $\alpha \cdot \beta$ to denote the execution α immediately followed by the execution β and say that α is a *prefix* of $\alpha \cdot \beta$. An execution is *solo* if every step is performed by the same process. Two executions α_1 and α_2 starting from configurations C_1 and C_2, respectively, are *indistinguishable* to some process p, if the state of p is the same in C_1 and C_2, and the sequence of steps performed by p (and thus also the responses p receives) are the same during both executions.

Fix an execution α in which a transaction T is executed. Transaction T *completes* in α, if α contains C_T or A_T. Transaction T *accesses* x in α, if α contains either $x.write()$ or $x.read()$. The *execution interval* of a completed transaction T in α is the subsequence of consecutive steps of α starting with the first step executed by any of the operations invoked by T and ending with the last such step. The *execution interval* of a transaction T that does not complete in α is the suffix of α starting with the first step executed by any of the operations invoked by T. The *active execution interval* of any transaction (completed or not) T in α is the subsequence of consecutive steps of α starting with the first step executed by any of the operations invoked by T and ending with the last such step. A TM algorithm is *obstruction-free* if a transaction T can be aborted only when other processes take steps during the execution interval of T.

Histories. A *history* H is a sequence of invocations and responses performed by transactions. Given an execution α, we denote by H_α the sequence of invocations and responses performed by the transactions in α. We denote by $H|T$ the longest subsequence of H consisting only of invocations and responses of a transaction T. Transaction T is in history H if $H|T$ is not empty. History H is *well-formed* if for every transaction T in H the following holds for $H|T$: (i) $H|T$ is a sequence of alternating invocations and responses starting with $begin_T \cdot ok$, (ii) each read invocation is followed either by a value or by A_T, (iii) each write invocation is followed by either an *ok* response or A_T, (iv) each invocation of $commit_T$ is followed by C_T or A_T, (v) each invocation of $abort_T$ is followed by A_T, (vi) no invocation follows after C_T or A_T. Herein, we consider only well-formed histories. We say that T *commits* (*aborts*) in H if $H|T$ ends with C_T (A_T). If T does not commit or abort in H, then T is *live* in H. If $H|T$ ends with an invocation of $commit_T$, then T is *commit-pending*. Transaction T_1 *precedes* transaction T_2 in execution α (denoted $T_1 <_\alpha T_2$), if T_1 is not live in H_α and A_{T_1} or C_{T_1} precedes $begin_{T_2}$ in H_α. If $T_1 \not<_\alpha T_2$ and $T_2 \not<_\alpha T_1$, then T_1 and T_2 are *concurrent* in α.

A history H is *sequential* if no two transactions are concurrent in H. H is *complete* if it does not contain any live transactions. Transaction T is *legal* in *a sequential history* H, if for every $x.read()$ by T which returns some value v the following holds: (i) if T executes an $x.write()$ before $x.read()$, then v is the argument of the last such $x.write()$ invocation in T; otherwise, (ii) if there is an invocation of $x.write()$ by some committed transaction that precedes T, then v is the argument of the last such $x.write()$ in H; otherwise (iii) v is the initial value of x. A complete sequential history H is *legal* if each transaction is legal in H.

Disjoint-access-parallelism. For proving the impossibility result presented in Section 4, we consider a collection of simple static transactions. Hence, to simplify the definitions in this paragraph, we assume that transactions are static and predefined[1], i.e. we assume that the data items on which T invokes *read* and *write* (in any execution con-

[1]Apparently, this assumption makes the impossibility result proved in Section 4 stronger.

taining T) are the same and can be derived by inspecting T's code; we call the set of these data items the *data set* $D(T)$ of T. Note that the set of data items accessed by T in a specific execution might be a proper subset of $D(T)$ if T is not committed in this execution. For example, consider a transaction T whose code implies that T accesses data items x and y and let α be an execution in which transaction T invokes $x.read()$ and gets A_T as a response; then, $D(T) = \{x, y\}$ but T accesses only data item x in α.

We say that two transactions T_1 and T_2 *conflict*, if $D(T_1) \cap D(T_2) \neq \emptyset$. We say that two executions *contend* on a base object o if they both contain a primitive operation on o and one of these primitive operations is non-trivial. Denote by $\alpha|T$ the subsequence of α consisting of all steps executed by T. A TM implementation $\mathcal{I}$ is *strict disjoint-access-parallel*, if in each execution α of $\mathcal{I}$, and for every two transactions T_1 and T_2 executed in α, $\alpha|T_1$ and $\alpha|T_2$ contend on some base object, only if T_1 and T_2 conflict.

Consistency. A read operation $x.read()$ by some transaction T is *global* if T has not invoked $x.write()$ before invoking $x.read()$. Let T be a committed or commit-pending transaction executed by a process p_i in a history H. Let $T|read_g$ be the longest subsequence of $H|T$ consisting only of global read invocations and their corresponding responses and $T|write$ be the longest subsequence of $H|T$ consisting only of write invocations and their corresponding responses. Let λ denote the empty history. Then we define transactions T_{gr} and T_w (both executed by p_i) in the following way:

- (1) $T_{gr} = T|read_g \cdot commit_{T_{gr}} \cdot C_{T_{gr}}$ if $T|read \neq \lambda$, and $T_{gr} = \lambda$ otherwise, and
- (2) $T_w = T|write \cdot commit_{T_w} \cdot C_{T_w}$ if $T|write \neq \lambda$, and $T_w = \lambda$ otherwise.

Definition 3.1 (Snapshot isolation). *An execution α satisfies* snapshot isolation*, if (i) there exists a set $com(\alpha)$ consisting of all committed and some of the commit-pending transactions in α and (ii) it is possible to insert (in α) a global read serialization point $*_{T,gr}$ and a write serialization point $*_{T,w}$, for each of transactions $T \in com(\alpha)$, so that if σ_α is the sequence defined by these serialization points, the following holds:*

1. *$*_{T,gr}$ precedes $*_{T,w}$ in σ_α,*
2. *both $*_{T,gr}$ and $*_{T,w}$ are inserted within the active execution interval of T,*
3. *if H_{σ_α} is the history we get by replacing each $*_{T,gr}$ with T_{gr} and each $*_{T,w}$ with T_w in σ_α, then H_{σ_α} is legal.*

An STM implementation I satisfies snapshot isolation, if each of the executions produced by I satisfies snapshot isolation.

Since we require neither consistency for local reads nor aborting two concurrent transactions writing to the same data item, our definition of snapshot isolation is weaker than standard definitions of snapshot isolation for databases [10]. This makes our impossibility result stronger.

To strengthen our impossibility result even more, we prove it for a much weaker consistency property, called *weak adaptive consistency* which allows (i) each process to have its own sequential view and (ii) to switch between snapshot isolation and processor consistency (defined below) during the course of an execution; specifically, the transactions of the execution can be partitioned into groups so that either snapshot isolation or processor consistency is ensured for the transactions of each group. Processor consistency is a safety property which allows each process to have its own sequential view but requires that writes to the same data item occur in the same order in each sequential view.

Definition 3.2 (Processor consistency). *An execution α is processor consistent if (i) there exists a set $com(\alpha)$ consisting of all committed and some of the commit-pending transactions in α and (ii) for each process p_i, it is possible to insert (in α) a serialization point $*_T$, for each transaction $T \in com(\alpha)$, so that if σ^i_α is the sequence defined by these serialization points, the following holds:*

1. *$\forall T_1, T_2 \in com(\alpha)$:*
   (a) *if T_1 and T_2 are executed by the same process and $T_1 <_\alpha T_2$, then $*_{T_1}$ precedes $*_{T_2}$ in σ^i_α,*
   (b) *if T_1 and T_2 write to the same data item and $*_{T_1}$ precedes $*_{T_2}$ in σ^i_α, then $\forall j \in \{1, \ldots, n\}$, $*_{T_1}$ precedes $*_{T_2}$ in σ^j_α,*
2. *if $H_{\sigma^i_\alpha}$ is the history we get by replacing each $*_T$ in σ^i_α with $H|T$, if T commits in α, or with $H|T \cdot C_T$, if T is commit-pending in α, then every transaction executed by p_i is legal in $H_{\sigma^i_\alpha}$.*

Let T_l and T_r be two transactions in an execution α such that either T_l and T_r are the same or the invocation of $begin_{T_l}$ precedes the invocation of $begin_{T_r}$. A *consistency group* $G(T_l, T_r)$ of α is a set of transactions from α such that: (1) T_l and T_r belong to $G(T_l, T_r)$, and (2) a transaction T_k belongs to $G(T_l, T_r)$ if the invocation of $begin_{T_k}$ occurs in α between the invocation of $begin_{T_l}$ and the invocation of $begin_{T_r}$. In other words, a consistency group $G(T_l, T_r)$ of α is a set containing the transactions that start their execution between the beginning of T_l and the beginning of T_r (inclusive). An *active execution interval of* $G(T_l, T_r)$ is the longest execution interval of α which includes all steps in α from the first step of T_l to the last step of any transaction from $G(T_l, T_r)$. A consistency group $G(T_l, T_r)$ *precedes* a consistency group $G(T'_l, T'_r)$ in α, if the last step of any transaction from $G(T_l, T_r)$ precedes the first step of T'_l in α.

A *consistency partition* $P(\alpha)$ of an execution α is a sequence $G(T_{l,1}, T_{r,1}), G(T_{l,2}, T_{r,2}), \ldots G(T_{l,n}, T_{r,n})$ of consistency groups such that:

1. $T_{l,1}$ is the transaction which invokes the first *begin* in α and $T_{r,n}$ is the transaction which invokes the last *begin* in α,
2. $\forall k \in \{1, \ldots, n\}$, either $T_{l,k} = T_{r,k}$ or $begin_{T_{l,k}}$ precedes $begin_{T_{r,k}}$ in α,
3. $\forall k \in \{1, \ldots, n-1\}$, $begin_{T_{r,k}}$ precedes $begin_{T_{l,k+1}}$ in α, and there is no transaction that invokes *begin* between $begin_{T_{r,k}}$ and $begin_{T_{l,k+1}}$ in α.

Definition 3.3 (Weak adaptive consistency). *An execution α satisfies* weak adaptive consistency *if it is possible to do all of the following: (i) choose a consistency partition $P(\alpha)$, (ii) partition all groups in $P(\alpha)$ into two disjoint sets of groups: a set $SI(P(\alpha))$ of* snapshot isolation groups *and a set $PC(P(\alpha))$ of* processor consistency groups*, (iii) choose a set $com(\alpha)$ consisting of all committed and some of*

the commit-pending transactions in α, and (iv) for each process p_i insert (in α) a global read serialization point $*_{T,gr}$ and a write serialization point $*_{T,w}$, for each transaction $T \in com(\alpha)$, so that if σ_α^i is the sequence defined by these serialization points, the following holds:

1. *$*_{T,gr}$ precedes $*_{T,w}$ in σ_α^i,*
2. *$\forall T_1, T_2 \in com(\alpha)$, if T_1 and T_2 write to the same data item and $*_{T_1,w}$ precedes $*_{T_2,w}$ in σ_α^i, then $\forall j \in \{1, \ldots, n\}$, $*_{T_1,w}$ precedes $*_{T_2,w}$ in σ_α^j,*
3. *for each group $G(T_{l,k}, T_{r,k}) \in SI(P(\alpha))$ the following holds: $\forall T_m \in G(T_{l,k}, T_{r,k}) \cap com(\alpha)$, both $*_{T_m,gr}$ and $*_{T_m,w}$ are inserted within the active execution interval of T_m,*
4. *for each group $G(T_{l,k}, T_{r,k}) \in PC(P(\alpha))$ the following holds: $\forall T_m \in G(T_{l,k}, T_{r,k}) \cap com(\alpha)$, no other serialization point is inserted between $*_{T_m,gr}$ and $*_{T_m,w}$ and both $*_{T_m,gr}$ and $*_{T_m,w}$ are inserted within the active execution interval of $G(T_{l,k}, T_{r,k})$,*
5. *if $H_{\sigma_\alpha^i}$ is the history we get by replacing each $*_{T,gr}$ with T_{gr} and each $*_{T,w}$ with T_w in σ_α^i, then every transaction executed by p_i is legal in $H_{\sigma_\alpha^i}$.*

Consider an execution α that satisfies weak adaptive consistency, let σ_α^i be the sequence of serialization points for process p_i. For simplicity, we use the following notation: $*_{T,l_1} <_i *_{T',l_2}$, where $T, T' \in com(\alpha)$ and $l_1, l_2 \in \{gr, w\}$, to identify that $*_{T,l_1}$ precedes $*_{T',l_2}$ in σ_α^i. We remark that items 3 and 4 of Definition 3.3 imply that the global read and write serialization points of any transaction $T \in com(\alpha)$ are placed within the active execution interval of the consistency group to which T belongs.

An STM implementation I satisfies weak adaptive consistency, if each of the executions produced by I satisfies weak adaptive consistency. Weak adaptive consistency is weaker than processor consistency, and consequently is weaker than causal serializability, serializability, opacity and any other property stronger than processor consistency. This is so because if an execution α satisfies processor consistency, then there exists a consistency partition $P(\alpha) = G(T_l, T_r)$ consisting only of one processor consistency group such that the active execution interval of $G(T_l, T_r)$ is exactly α, and therefore, serialization points of transactions from $G(T_l, T_r)$ can be inserted anywhere in α. Weak adaptive consistency is weaker than snapshot isolation because in the definition of snapshot isolation there is only one sequential view σ_α, and condition 2 of the above definition trivially holds for the case of a single sequential view σ_α. In fact, weak adaptive consistency is even weaker than the union of snapshot isolation and processor consistency.

4. THE PCL THEOREM

In this section we show that it is impossible to implement a TM which ensures weak adaptive consistency, obstruction-freedom, and strict disjoint-access-parallelism. The main idea behind the proof is the following. We design two legal executions $\alpha = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \alpha_7$ and $\alpha' = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot s_1 \cdot \alpha_7'$, where α_1 and α_2 are parts of solo executions of some transactions T_1 (executed by process p_1) and T_2 (executed by process p_2), respectively, s_1 and s_2 are single steps by T_1 and T_2, respectively, and α_7 and α_7' are solo executions of T_7 (executed by process p_7) until it commits. We prove that s_1 and s_2 are steps accessing different base objects, so α_7 is indistinguishable from α_7' to process p_7. We also prove that there exists a data item in T_7's read set for which T_7 reads a different value in α_7 from the value read for the same data item in α_7', which is a contradiction.

THEOREM 4.1 (THE PCL THEOREM). *There is no TM implementation which is strict disjoint-access-parallel and satisfies weak adaptive consistency and obstruction-freedom.*

PROOF. Assume, by contradiction, that there exists an obstruction-free implementation I which is strict disjoint-access-parallel and satisfies weak adaptive consistency.

We use the following notation: we denote by b_k, c_k, d_k data items written by transaction T_k and by $e_{k,m}$ data items written by both transactions T_k and T_m. Consider the following transactions (the initial value of every data item is considered to be 0):

- T_1, executed by process p_1, which reads data items b_3 and b_7, and writes the value 1 to data items a, b_1, c_1, d_1, $e_{1,3}$,
- T_2, executed by process p_2, which reads data items b_5 and b_7, and writes the value 2 to data items a, b_2, c_2, d_2, $e_{2,5}$, $e_{2,7}$,
- T_3, executed by process p_3, which reads data items b_1 and b_4, and writes the value 1 to data items b_3, c_3, $e_{1,3}$, $e_{3,4}$,
- T_4, executed by process p_4, which reads data items d_2 and c_3, and writes the value 1 to data items b_4, $e_{3,4}$,
- T_5, executed by process p_5, which reads data items b_2 and b_6, and writes the value 1 to data items b_5, c_5, $e_{2,5}$, $e_{5,6}$,
- T_6, executed by process p_6, which reads data items d_1 and c_5, and writes the value 1 to data items b_6, $e_{5,6}$,
- T_7, executed by process p_7, which reads data items a, c_1, and c_2, and writes the value 1 to data items b_7, $e_{2,7}$.

Definition of α_1 and s_1: Let transaction T_1 be executed solo from the initial configuration C_0. Because p_1 runs solo and I is obstruction-free, T_1 eventually commits. In the resulting execution, T_1 reads the value 0 for data items b_3 and b_7 because I satisfies weak adaptive consistency and there is no transaction that writes to these data items in this execution. Let C' be the configuration resulting from the execution of the last step of T_1.

If T_3 is executed solo from the initial configuration C_0, then in the resulting execution, T_3 reads 0 for b_1 (since I satisfies weak adaptive consistency and no transaction writes to b_1 in this execution).

Consider now the execution δ_1 where transaction T_3 is executed solo from C' until it commits. We prove that T_3 reads the value 1 for data item b_1 in δ_1. Since I satisfies weak adaptive consistency, there exists a consistency partition $P(\delta_1)$ which satisfies the conditions of Definition 3.3. We consider the following cases:

- Assume first that $P(\delta_1) = G(T_1, T_3)$ and $SI(P(\delta_1)) = \{G(T_1, T_3)\}$. Since $G(T_1, T_3)$ is a snapshot isolation group, then $*_{T_1,w}$ must be placed within the active execution interval of T_1 and $*_{T_3,gr}$ must be placed within

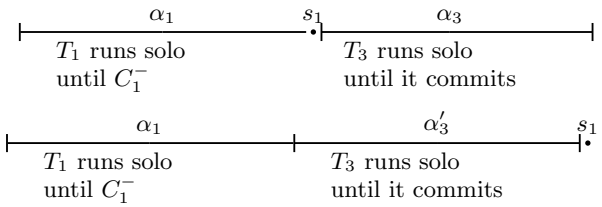

Figure 1: Executions α_1, α_3, α_3', and step s_1.

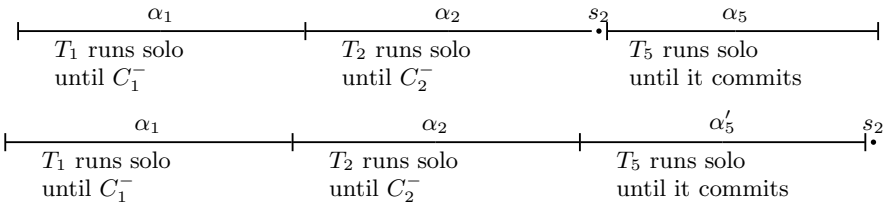

Figure 2: Executions α_1, α_2, α_5, α_5', and step s_2.

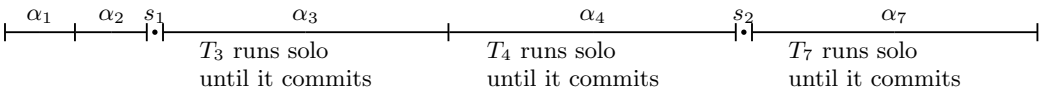

Figure 3: Execution β.

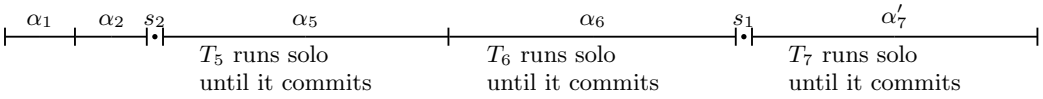

Figure 4: Execution β'.

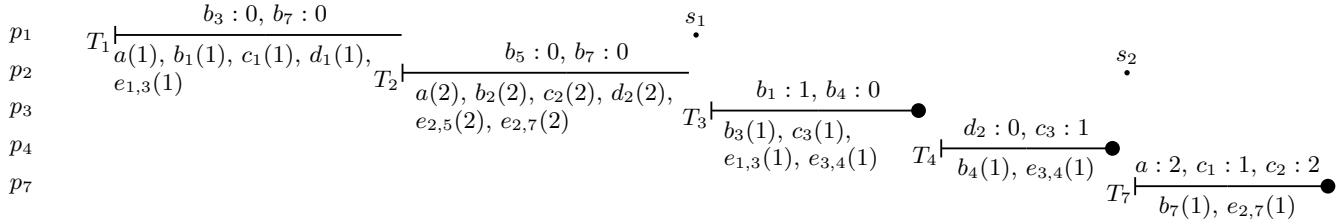

Figure 5: Values read by transactions in execution β. Where $x : v$ denotes a read from x which returns value v, $x(v)$ denotes a write to x which writes value v, and $\bullet$ denotes a commit event.

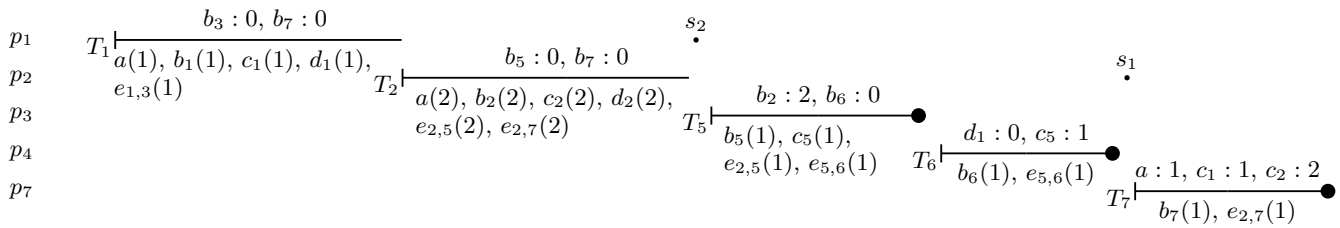

Figure 6: Values read by transactions in execution β'.

the active execution interval of T_3[2]. Since $T_1 <_{\delta_1} T_3$, then $*_{T_1,w} <_3 *_{T_3,gr}$. It follows that T_3 must observe the update performed on data item b_1 by T_1, and consequently T_3 must read 1 for b_1.

- Assume that $P(\delta_1) = G(T_1, T_3)$ and $PC(P(\delta_1)) = \{G(T_1, T_3)\}$. Because T_1 reads 0 for b_3 in δ_1, it follows that $*_{T_1,gr} <_1 *_{T_3,w}$. Since $G(T_1, T_3)$ is a processor consistency group, no serialization point is inserted between $*_{T_1,gr}$ and $*_{T_1,w}$. Thus, $*_{T_1,w} <_1 *_{T_3,w}$. Because T_1 and T_3 write to the same data item $e_{1,3}$, it follows that $*_{T_1,w} <_3 *_{T_3,w}$. Since no serialization point is inserted between $*_{T_3,gr}$ and $*_{T_3,w}$, it follows that $*_{T_1,w} <_3 *_{T_3,gr}$, so T_3 must read 1 for b_1.
- Assume now that $P(\delta_1) = G(T_1, T_1), G(T_3, T_3)$. Because $T_1 <_{\delta_1} T_3$, it follows that $G(T_1, T_1)$ precedes $G(T_3, T_3)$ in δ_1. Since $*_{T_1,w}$ should be placed within the execution interval of $G(T_1, T_1)$ and $*_{T_3,gr}$ should be placed within the execution interval of $G(T_3, T_3)$, $*_{T_1,w} <_3 *_{T_3,gr}$. Hence, T_3 must read 1 for b_1.

Since in the solo execution of T_3 from C_0, T_3 reads 0 for b_1, whereas in the solo execution of T_3 from C', T_3 reads 1 for b_1, it follows that there exists some step s_1 in the solo execution of T_1 from C_0, resulting in a configuration C_1, such that: (I) if α'_3 is the solo execution of T_3 from configuration C_1^- (where C_1^- is the configuration just before s_1), then, in α'_3, T_3 reads 0 for b_1; and (II) if α_3 is the solo execution of T_3 from configuration C_1, then in α_3, T_3 reads 1 for b_1. (If there are more than one steps with this property, let s_1 be the first of them.)

Denote by α_1 the solo execution of T_1 from C_0 until C_1^- is reached (Figure 1). Note that T_3 reads 0 for b_4 in α_3 since there is no transaction in $\alpha_1 \cdot s_1 \cdot \alpha_3$ which writes to b_4.

Claim 1: *Transaction T_1 invokes $commit_{T_1}$ in α_1.*
Proof: Assume, by contradiction, that T_1 does not invoke $commit_{T_1}$ in α_1. We argue that the execution $\alpha_1 \cdot s_1 \cdot \alpha_3$ does not satisfy weak adaptive consistency. This is so because, by definition of s_1, T_3 reads 1 for b_1 in this execution but T_1 is not yet commit-pending and therefore we cannot assign a write serialization point to T_1 in this execution. □

Claim 2: *Step s_1 applies a non-trivial operation op on some base object o_1 for which the following holds: T_3 reads o_1 in α_3 and α'_3.*
Proof: If op is a trivial operation on o_1 or T_3 does not read o_1 in α_3 (or in α'_3), then α_3 and α'_3 are indistinguishable to p_3. This is a contradiction, since by the definition of s_1, T_3 reads a different value for b_1 in these two executions. □

Definition of α_2 and s_2: Using a similar reasoning as above, we can show that in an execution where T_2 is executed solo from C_1^- until it commits, there is a step s_2, resulting in a configuration C_2, such that:

1. if α'_5 is the solo execution of T_5 from configuration C_2^-, where C_2^- is the configuration just before the execution of s_2, then T_5 reads 0 for b_2 in α'_5;
2. if α_5 is the solo execution of T_5 from configuration C_2, then T_5 reads 2 for b_2 and 0 for b_6 in α_5,
3. if α_2 is the solo execution of T_2 from C_1^- until C_2^- is reached (Figure 2), then T_2 invokes $commit_{T_2}$ in α_2,
4. T_2 reads the value 0 for data items b_5 and b_7 in α_2,
5. s_2 applies a non-trivial operation on some base object o_2 which is read in α_5 and α'_5.

[2] For simplicity, throughout the proof, we use the term *execution interval* instead of *active execution interval*, whenever it is clear from the context.

Claim 3: $o_1 \neq o_2$
Proof: Assume that $o_1 = o_2$. Consider an execution $\alpha_1 \cdot \alpha_2 \cdot s'_1 \cdot \gamma_3$, where s'_1 is a single step by p_1 and γ_3 is a solo execution of T_3 by p_3 until T_3 commits. We argue that $s'_1 = s_1$ and γ_3 is indistinguishable from α_3 to p_3. Obviously, since s'_1 is the step that p_1 is poised to perform after α_1, s_1 and s'_1 access the same base object, namely object o_1. Thus, if s_1 and s'_1 are different, they differ in their response.

Since T_3 and T_2 do not conflict, strict disjoint-access-parallelism implies that α_2 does not contain any non-trivial operation on base objects read in γ_3. Thus, the prefix of α_3 until the point that o_1 is first accessed is also a prefix of γ_3. Therefore, T_3 reads o_1 in γ_3 (as it does in α_3).

Because γ_3 reads o_1, and T_3 and T_2 do not conflict, strict disjoint-access-parallelism implies that α_2 does not contain a non-trivial operation on $o_1 = o_2$. It follows that $s_1 = s'_1$. This and the fact that T_3 and T_2 do not conflict imply that γ_3 is indistinguishable from α_3 to p_3. So, execution $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3$ is legal.

Since p_2 is poised to execute a step which applies a non-trivial operation on $o_2 = o_1$ after $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3$ and o_1 is read in α_3, it follows that in execution $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot s'_2$, where s'_2 is a single step by p_2, strict disjoint-access-parallelism is violated. This is a contradiction. Thus, $o_1 \neq o_2$. □

Consider executions $\alpha = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \alpha_7$ and $\alpha' = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot s_1 \cdot \alpha'_7$, where α_7 and α'_7 are solo executions of T_7 until T_7 commits. Since steps s_1 and s_2 access different base objects, α_7 is indistinguishable from α'_7 to process p_7.

Consider an execution $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot \alpha_4 \cdot s''_2$, where α_4 is the solo execution of T_4 until it commits. We first argue that $s''_2 = s_2$.

Recall that $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3$ is legal, so $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot \alpha_4$ is legal. Notice that s''_2, like s_2, accesses base object o_2. It remains to argue that the response of s''_2 is the same as that of s_2. Recall that α_3 does not contain a non-trivial operation on o_2. It remains to argue that the same is true for α_4.

Consider the execution $\delta_2 = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot \alpha_4 \cdot \alpha'_5$. Recall that α'_5 is the solo execution of T_5 from configuration C_2^-. Since T_5 does not conflict with T_1, T_3, and T_4, strict disjoint-access-parallelism implies that $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot \alpha_4 \cdot \alpha'_5$ is legal. Since α'_5 reads o_2 and T_5 does not conflict with T_4, strict disjoint-access-parallelism implies that α_4 does not contain a non-trivial operation on o_2. It follows that $s''_2 = s_2$.

Let $\beta = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot \alpha_4 \cdot s_2 \cdot \alpha_7$ (see Figure 3). We first argue that β is legal. This is so because T_7 does not conflict neither with T_3 nor with T_4, so α_7 does not access any base object modified in α_3 or α_4.

We now argue that T_7 reads 2 for data items a and c_2, and 1 for data item c_1 in α_7.

Claim 4: *T_7 reads the value 2 for data items a and c_2, and the value 1 for data item c_1 in α_7.*

Proof: *We first prove that transaction T_4 reads 0 for d_2 in α_4.* Recall that $\delta_2 = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot \alpha_4 \cdot \alpha'_5$ is legal. Since I satisfies weak adaptive consistency, there exists a consistency partition $P(\delta_2)$ and a set $com(\delta_2)$ of committed and commit-pending transactions from δ_2 which satisfy the conditions of Definition 3.3. We argue that $T_2 \notin com(\delta_2)$.

Assume, by contradiction, that $T_2 \in com(\delta_2)$. We consider the following cases.

- Assume first that $P(\delta_2)$ includes $G(T_i, T_5)$ for some $i \in \{3, 4, 5\}$. Since the execution intervals of T_2 and T_1 precede the execution intervals of T_3, T_4, and T_5, and the execution intervals of T_3, T_4, and T_5 do not overlap in δ_2, it follows that the execution interval of the consistency group containing T_2 in $P(\delta_2)$ precedes the execution interval of $G(T_i, T_5)$. Thus, $*_{T_2,w} <_5 *_{T_5,gr}$. This contradicts the fact that T_5 reads 0 for b_2 in α'_5.

- Assume that $P(\delta_2)$ includes $G(T_i, T_5)$ for some $i \in \{1, 2\}$ and $G(T_i, T_5) \in SI(P(\delta_2))$. Since $G(T_i, T_5)$ is a snapshot isolation group and the execution interval of T_2 precedes the execution interval of T_5 in δ_2, it follows that $*_{T_2,w} <_5 *_{T_5,gr}$. This contradicts the fact that T_5 reads 0 for b_2 in α'_5.

- Assume that $P(\delta_2)$ includes $G(T_i, T_5)$ for some $i \in \{1, 2\}$ and $G(T_i, T_5) \in PC(P(\delta_2))$. Because T_2 reads 0 for b_5 in α_2, it follows that $*_{T_2,gr} <_2 *_{T_5,w}$. Since no point is inserted between $*_{T_2,gr}$ and $*_{T_2,w}$, it follows that $*_{T_2,w} <_2 *_{T_5,w}$. Because T_2 and T_5 write to the same data item $e_{2,5}$, it follows that $*_{T_2,w} <_5 *_{T_5,w}$. Since no point is inserted between $*_{T_5,gr}$ and $*_{T_5,w}$, it follows that $*_{T_2,w} <_5 *_{T_5,gr}$. This contradicts the fact that T_5 reads 0 for b_2 in α'_5.

Hence, $T_2 \notin com(\delta_2)$, and consequently T_4 reads 0 for d_2 in α_4.

We next argue that T_4 reads 1 for c_3 in α_4. Since $\delta_3 = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \alpha_3 \cdot \alpha_4$ is legal and I satisfies weak adaptive consistency, there exists a consistency partition $P(\delta_3)$ and a set $com(\delta_3)$ of committed and commit-pending transactions from δ_3 which satisfy the conditions of Definition 3.3. We consider the following cases:

- Assume first that $P(\delta_3)$ includes $G(T_4, T_4)$. Since the execution interval of any consistency group containing transaction T_3 precedes the execution interval of $G(T_4, T_4)$, it follows that $*_{T_3,w} <_4 *_{T_4,gr}$.

- Assume now that $P(\delta_3)$ includes $G(T_i, T_4)$ for some $i \in \{1, 2, 3\}$ and that $G(T_i, T_4) \in SI(P(\delta_3))$. Since $T_3 <_{\delta_3} T_4$, it follows that $*_{T_3,w} <_4 *_{T_4,gr}$.

- Assume now that $P(\delta_3)$ includes $G(T_i, T_4)$ for some $i \in \{1, 2, 3\}$ and that $G(T_i, T_4) \in PC(P(\delta_3))$. Since T_3 reads 0 for b_4 in α_3, it follows that $*_{T_3,gr} <_3 *_{T_4,w}$. Since no point is inserted between $*_{T_3,gr}$ and $*_{T_3,w}$, it follows that $*_{T_3,w} <_3 *_{T_4,w}$. Because T_3 and T_4 write to the same data item $e_{3,4}$, it follows that $*_{T_3,w} <_4 *_{T_4,w}$. Since no point is inserted between $*_{T_4,gr}$ and $*_{T_4,w}$, it follows that $*_{T_3,w} <_4 *_{T_4,gr}$.

We conclude that (in all cases) $*_{T_3,w} <_4 *_{T_4,gr}$. Thus, T_4 reads 1 for c_3 in α_4.

We now prove that T_7 reads 2 for c_2 in α_7. Notice that $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \alpha_5$ is legal. This is so because s_1 and s_2 are steps on different base objects and T_5 does not conflict with T_1, so it does not access o_1 in α_5. We argue that $\delta_4 = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \alpha_5 \cdot \alpha_7$ is also legal. This is so because $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \alpha_5$ and $\alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \alpha_7$ are legal and T_7 does not conflict with T_5.

Since I satisfies weak adaptive consistency, there exists a consistency partition $P(\delta_4)$ and a set $com(\delta_4)$ of committed and commit-pending transactions from δ_4 which satisfy the conditions of Definition 3.3. Since T_5 reads 2 for b_2 in α_5, it follows that $T_2 \in com(\delta_4)$ and $*_{T_2,w} <_5 *_{T_5,gr}$. We consider the following cases.

- Assume first that $P(\delta_4)$ includes $G(T_i, T_7)$ for some $i \in \{5, 7\}$. Since the execution intervals of T_1 and T_2 do not overlap with the execution intervals of T_5 and T_7 in δ_4, it follows that the execution interval of the consistency group of $P(\delta_4)$ that contains transaction T_2 precedes the execution interval of $G(T_i, T_7)$. Therefore, $*_{T_2,w} <_7 *_{T_7,gr}$, and consequently T_7 must read 2 for c_2 in α_7.

- Assume now that $P(\delta_4)$ includes $G(T_i, T_7)$, for some $i \in \{1, 2\}$ and that $G(T_i, T_7) \in SI(P(\delta_4))$. Since the execution interval of T_2 precedes the execution interval of T_7 and $T_2, T_7 \in G(T_i, T_7)$, which is a snapshot isolation group, it follows that $*_{T_2,w} <_7 *_{T_7,gr}$. Thus, T_7 must read 2 for c_2 in α_7.

- Assume now that $P(\delta_4)$ includes $G(T_i, T_7)$, for some $i \in \{1, 2\}$ and that $G(T_i, T_7) \in PC(P(\delta_4))$. Because T_2 reads 0 for b_7 in α_2, it follows that $*_{T_2,gr} <_2 *_{T_7,w}$. Since no point is inserted between $*_{T_2,gr}$ and $*_{T_2,w}$, it follows that $*_{T_2,w} <_2 *_{T_7,w}$. Because T_2 and T_7 write to the same data item $e_{2,7}$, it follows that $*_{T_2,w} <_7 *_{T_7,w}$. Since no point is inserted between $*_{T_7,gr}$ and $*_{T_7,w}$, it follows that $*_{T_2,w} <_7 *_{T_7,gr}$. Thus, T_7 must read 2 for c_2 in α_7.

Since T_3 reads 1 for b_1 in β, it follows that $*_{T_1,w} <_3 *_{T_3,gr} <_3 *_{T_3,w}$. Because T_1 and T_3 write to the same data item $e_{1,3}$, it follows that $*_{T_1,w} <_4 *_{T_3,w}$. Since T_4 reads 0 for d_2 and 1 for c_3, it follows that $*_{T_3,w} <_4 *_{T_4,gr} <_4 *_{T_2,w}$. Thus, $*_{T_1,w} <_4 *_{T_2,w}$. Because T_1 and T_2 write to the same data item a, it follows that $*_{T_1,w} <_7 *_{T_2,w}$. Since T_7 reads 2 for c_2, it follows that $*_{T_2,w} <_7 *_{T_7,gr}$. Thus, $*_{T_1,w} <_7 *_{T_2,w} <_7 *_{T_7,gr}$, and consequently, it follows that T_7 reads 2 for a and 1 for c_1 in α_7 (see Figure 5). □

Consider now execution $\alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5 \cdot \alpha_6 \cdot s''_1$, where α_6 is the solo execution of T_6 until it commits. We first argue that $s''_1 = s_1$.

Recall that $\alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5$ is legal, so $\alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5 \cdot \alpha_6$ is legal. Notice that s''_1, like s_1, accesses base object o_1. It remains to argue that the response of s''_1 is the same as that of s_1. Recall that α_5 does not contain a non-trivial operation on o_1. It remains to argue that the same is true for α_6.

Consider an execution $\delta_5 = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5 \cdot \alpha_6 \cdot \alpha'_3$. Since T_3 does not conflict with T_2, T_5, and T_6, strict disjoint-access-parallelism implies that $\alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5 \cdot \alpha_6 \cdot \alpha'_3$ is legal. Recall that α'_3 reads o_1. Since T_3 does not conflict with T_6, strict disjoint-access-parallelism implies that α_6 does not contain a non-trivial operation on o_1. It follows that $s''_1 = s_1$.

Let $\beta' = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5 \cdot \alpha_6 \cdot s_1 \cdot \alpha'_7$ (see Figure 4). We first argue that β' is legal. This is so because T_7 does not conflict neither with T_5 nor with T_6, so α'_7 does not access any base object modified in α_5 or α_6.

Claim 5: *T_7 reads 1 for a in α'_7.*

Proof: *We first prove that transaction T_6 reads 0 for d_1 in α_6.* Recall that $\delta_5 = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5 \cdot \alpha_6 \cdot \alpha'_3$ is legal. Since I satisfies weak adaptive consistency, there exists a consistency partition $P(\delta_5)$ and a set $com(\delta_5)$ of committed and commit-pending transactions from δ_5 which satisfy the conditions of Definition 3.3. We argue that $T_1 \notin com(\delta_5)$.

Assume, by contradiction, that $T_1 \in com(\delta_5)$. We consider the following cases.

- Assume first that $P(\delta_5)$ includes $G(T_i, T_3)$ for some $i \in \{2, 3, 5, 6\}$. Since the execution interval of T_1 precedes the execution intervals of T_2, T_3, T_6 and T_5, and the executions intervals of T_2, T_3, T_6 and T_5 do not overlap in δ_5, it follows that the execution interval of the consistency group containing T_1 precedes the execution interval of $G(T_i, T_3)$ in δ_5. Thus, $*_{T_1,w} <_3 *_{T_3,gr}$. This contradicts the fact that T_3 reads 0 for b_1 in α'_3.
- Assume now that $P(\delta_5) = G(T_1, T_3)$ and that it holds that $SI(P(\delta_5)) = \{G(T_1, T_3)\}$. Since the execution interval of T_1 precedes the execution interval of T_3 in δ_5, it follows that $*_{T_1,w} <_3 *_{T_3,gr}$. This contradicts the fact that T_3 reads 0 for b_1 in α'_3.
- Assume now that $P(\delta_5) = G(T_1, T_3)$ and that it holds that $PC(P(\delta_5)) = \{G(T_1, T_3)\}$. Because T_1 reads 0 for b_3 in α_1, it follows that $*_{T_1,gr} <_1 *_{T_3,w}$. Since no point is inserted between $*_{T_1,gr}$ and $*_{T_1,w}$, it follows that $*_{T_1,w} <_1 *_{T_3,w}$. Because T_1 and T_3 write to the same data item $e_{1,3}$, it follows that $*_{T_1,w} <_3 *_{T_3,w}$. Since no point is inserted between $*_{T_3,gr}$ and $*_{T_3,w}$, it follows that $*_{T_1,w} <_3 *_{T_3,gr}$. This contradicts the fact that T_3 reads 0 for b_1 in α'_3.

Hence, $T_1 \notin com(\delta_5)$, and consequently T_6 reads 0 for d_1 in α_6.

We next argue that T_6 *reads* 1 *for* c_5 *in* α_6. Since $\delta_6 = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \alpha_5 \cdot \alpha_6$ is legal and I satisfies weak adaptive consistency, there exists a consistency partition $P(\delta_6)$ and a set $com(\delta_6)$ of committed and commit-pending transactions from δ_6 which satisfy the conditions of Definition 3.3. We consider the following cases:

- Assume first that $P(\delta_6)$ includes $G(T_6, T_6)$. Since the execution interval of any consistency group containing transaction T_5 must precede the execution interval of $G(T_6, T_6)$, it follows that $*_{T_5,w} <_6 *_{T_6,gr}$.
- Assume now that $P(\delta_6)$ includes $G(T_i, T_6)$ for some $i \in \{1, 2, 5\}$ and that $G(T_i, T_6) \in SI(P(\delta_6))$. Since $T_5 <_{\delta_6} T_6$, it follows that $*_{T_5,w} <_6 *_{T_6,gr}$.
- Assume now that $P(\delta_6)$ includes $G(T_i, T_6)$ for some $i \in \{1, 2, 5\}$ and that $G(T_i, T_6) \in PC(P(\delta_6))$. Since T_5 reads 0 for b_6 in α_5, it follows that $*_{T_5,gr} <_5 *_{T_6,w}$. Since no point is inserted between $*_{T_5,gr}$ and $*_{T_5,w}$, it follows that $*_{T_5,w} <_5 *_{T_6,w}$. Because T_5 and T_6 write to the same data item $e_{5,6}$, it follows that $*_{T_5,w} <_6 *_{T_6,w}$. Since no point is inserted between $*_{T_6,gr}$ and $*_{T_6,w}$, it follows that $*_{T_5,w} <_6 *_{T_6,gr}$.

We conclude that (in all cases) $*_{T_5,w} <_6 *_{T_6,gr}$. Thus, T_6 reads 1 for c_5 in α_6.

Because α'_7 and α_7 are indistinguishable to p_7, it follows that T_7 reads the same values in α'_7 and α_7. Since T_5 reads 2 for b_2 in β', it follows that $*_{T_2,w} <_5 *_{T_5,gr} <_5 *_{T_5,w}$. Because T_2 and T_5 write to the same data item $e_{2,5}$, it follows that $*_{T_2,w} <_6 *_{T_5,w}$. Since T_6 reads 0 for d_1 and 1 for c_5, it follows that $*_{T_5,w} <_6 *_{T_6,gr} <_6 *_{T_1,w}$. Thus, $*_{T_2,w} <_6 *_{T_1,w}$. Because T_1 and T_2 write to the same data item a, it follows that $*_{T_2,w} <_7 *_{T_1,w}$. Since T_7 reads 1 for c_1, it follows that $*_{T_1,w} <_7 *_{T_7,gr}$. Thus, $*_{T_2,w} <_7 *_{T_1,w} <_7 *_{T_7,gr}$ in β', and consequently, it follows that T_7 reads 1 for a in β' (see Figure 6) and in α'_7. □

Claim 4 states that T_7 reads 2 for data item a in α_7, and Claim 5 states that T_7 reads 1 for data item a in α'_7. Since α_7 is indistinguishable from α'_7 to process p_7 this is a contradiction. □

5. DISCUSSION

We proved the PCL theorem: in transactional systems it is impossible to ensure strict disjoint-access-parallelism (Parallelism), weak adaptive consistency (Consistency), and obstruction-freedom (Liveness). To circumvent the impossibility result it is sufficient to weaken just one of the three requirements. Weakening obstruction-freedom to a blocking liveness property makes it possible to ensure strict disjoint-access-parallelism and strong consistency (e.g. strict serializability) by using locks; these are the properties ensured by TL [14]. Likewise, weakening consistency makes it possible to ensure strict disjoint-access-parallelism and strong liveness. For example, allowing writes to the same data item to be viewed differently, as in PRAM consistency [28], makes it possible to trivially ensure strict disjoint-access-parallelism and wait-freedom, the strongest liveness property, without any synchronization between processes. In [11], we design a simple variant of DSTM [25], which satisfies snapshot isolation, obstruction-freedom, and the following weakening of strict disjoint-access-parallelism: two write operations on different data items contend on the same base object only if there is a chain of transactions starting with the transaction that performs one of these write operations and ending with the transaction that performs the other, such that every two consecutive transactions in the chain *conflict*. The PCL theorem shows that the distance between strict disjoint-access-parallelism and its non-strict forms draws a sharp line in the design of transactional systems.

Our theorem might at first glance look close to the CAP theorem [18] which states that it is impossible to ensure *consistency*, *availability*, and *partition* in a distributed system and weakening at least one of these requirements circumvents the impossibility result. In fact, they are different results. While consistency can be viewed as a safety property, and availability can be viewed as a liveness property, partition is not analogous to disjoint-access-parallelism. Specifically, partition tolerance ensures that the system tolerates arbitrary network partitions. Disjoint-access-parallelism on the other hand, does not ensure tolerance against failures but imposes that logical components of a system (transactions or operations) do not contend at low level (i.e. on base objects) if they do not conflict at high level (i.e. do not access the same data items).

Our definition of snapshot isolation is incomparable to strict serializability [30] and opacity [22]. This is because strict serializability and opacity are defined in terms of execution intervals whereas our definition of snapshot isolation is based on *active* execution intervals. The same holds for previous definitions of snapshot isolation, both in the database world [10], and in TM computing [4, 33]. We can easily remedy this problem by defining snapshot isolation in terms of execution intervals. Indeed, we did so in [11] and we were able to prove [11] that no TM implementation satisfies that version of snapshot isolation, obstruction-freedom, and strict disjoint-access-parallel. This impossibility result [11] also holds if a primitive accesses up to k base objects in one atomic step.

6. ACKNOWLEDGEMENTS

This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project and by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project.

7. REFERENCES

[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of ACM STOC '95*.

[2] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations (extended abstract). In *Proceedings of ACM PODC '97*.

[3] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proceedings of ACM SPAA '93*.

[4] M. S. Ardekani, P. Sutra, and M. Shapiro. The impossibility of ensuring snapshot isolation in genuine replicated stms. In *WTTM'11*.

[5] H. Attiya and E. Dagan. Universal operations: unary versus binary. In *Proceedings of ACM PODC '96*.

[6] H. Attiya and E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. In *Proceedings of ACM DISC'06*.

[7] H. Attiya and E. Hillel. Single-version stms can be multi-version permissive. In *Proceedings of ICDCN'11*. Springer-Verlag.

[8] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of ACM SPAA '09*.

[9] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of ACM SPAA '93*.

[10] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Rec.*, 24(2):1–10, 1995.

[11] V. Bushkov, D. Dziuma, P. Fatourou, and R. Guerraoui. Snapshot isolation does not scale either. Technical Report TR-437, FORTH-ICS, 2013.

[12] V. Bushkov, R. Guerraoui, and M. Kapałka. On the liveness of transactional memory. In *Proceedings of ACM PODC '12*.

[13] R. J. Dias, J. Seco, and J. M. Lourenço. Snapshot isolation anomalies detection in software transactional memory. In *Proceedings of InForum 2010*.

[14] D. Dice and N. Shavit. What really makes transactions faster? In *Proceedings of ACM TRANSACT'06*.

[15] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of ACM PODC '12*.

[16] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2), 2005.

[17] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[18] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[19] J. R. Goodman. Cache consistency and sequential consistency. Technical report, Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[20] J. Gray. A transaction model. In *Proceedings of ICALP '80*. Springer-Verlag.

[21] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of ACM SPAA '08*.

[22] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of ACM PPoPP '08*.

[23] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM SIGPLAN Not.*, 25(3), 1990.

[24] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[25] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of ACM PODC'03*.

[26] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2), 1993.

[27] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of ACM PODC '94*.

[28] R. J. Lipton and J. S. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[29] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of DISC'05*. Springer-Verlag.

[30] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4), 1979.

[31] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *Proceedings of ACM PODC '10*.

[32] M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *Proceedings of EUROMICRO '97*.

[33] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Proceedings of ACM TRANSACT'06*.

[34] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguiça. On the scalability of snapshot isolation. In *Euro-Par Parallel Processing*. Springer Berlin Heidelberg, 2013.

[35] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of ACM PODC '95*.

[36] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. Nztm: nonblocking zero-indirection transactional memory. In *Proceedings of ACM SPAA '09*.

[37] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of ACM PODS '92*.