Consistency-Aware Scheduling for Weakly Consistent Programs

Maryam Dabaghchian, Zvonimir Rakamarić
School of Computing
University of Utah
Salt Lake City, UT, USA
maryam,zvonimir@cs.utah.edu

Burcu K. Ozkan
Max Planck Institute
for Software Systems
Kaiserslautern, Germany
burcu@mpi-sws.org

Erdal Mutlu, Serdar Tasiran
Computer Science
Koç University
Istanbul, Turkey
ermutlu,stasiran@ku.edu.tr

ABSTRACT

Modern geo-replicated data stores provide high availability by relaxing the underlying consistency requirements. Programs layered over such data stores are called weakly consistent programs. Due to the reduced consistency requirements, they exhibit highly nondeterministic behaviors, some of which might violate program invariants. Therefore, implementing correct weakly consistent programs and reasoning about them is challenging. In this paper, we present a systematic scheduling approach that is aware of the underlying consistency model. Our approach dynamically explores all possible program behaviors allowed by the used data store consistency model, and it evaluates program invariants during the exploration. We implement the approach in a prototype model checker for Antidote, which is a causally consistent key-value data store with convergent conflict handling. We evaluate our tool on several benchmarks. The results show that our approach is effective in detecting buggy behaviors in weakly consistent programs.

1. INTRODUCTION

Modern Internet-scale programs often rely on high-performance geo-replicated data stores. In such data stores, replicas are located in geographically separate locations to avoid latency in the wide area network and tolerate network partitioning. According to the Consistency, Availability, and Partition tolerance (CAP) theorem [16], partitioning is unavoidable, and data stores have to sacrifice either strong consistency or availability. Modern data stores provide high availability through weaker consistency models called *eventual consistency* [26]. We refer to an atomic step that updates some data in such data stores as an *event*. In general, eventual consistency guarantees that events occurred at each replica will eventually be propagated and become visible on all remote replicas.

Programs using such geo-replicated data stores maintain a copy of their data on different replicas. However, due to the often limited synchronization guarantees, it is possible to have conflicting concurrent events on different replicas. In order to provide eventual consistency, many replicated data types are equipped with conflict resolution mechanisms [8, 9, 23, 13]. Such data types are called *conflict-free replicated data types* (CRDTs) [24, 5].

Due to the relaxed consistency guarantees of the systems using CRDTs, a wider set of program behaviors is possible when compared to a strongly consistent system, some of which are unintuitive. This makes it harder for developers to reason about expected executions of their programs and specify the intended program behavior correctly. Such subtle *schedules* (i.e., execution orders) can violate the intended invariants of programs written with CRDTs.

In order to assist the developers in overcoming the challenges of writing correct CRDT programs, we introduce a systematic scheduling approach that is aware of the underlying consistency model. Our approach is parameterized both in terms of the used schedule exploration strategy and instantiated consistency model, i.e., it is consistency-aware. Since consistency-aware scheduling takes the consistency guarantee into consideration while generating new schedules, it is precise in the sense that the generated schedules satisfy the consistency requirements. Hence, it neither misses bugs due to exploring only strongly consistent schedules nor reports false bugs by exploring overly relaxed weakly consistent schedules.

Within our approach, we propose two schedule exploration strategies (random and extended delay-bounded [14]) to detect violations of the supplied program invariants. We implement our approach in a tool for the Antidote platform [3, 2], which is a highly available geo-replicated CRDT key-value data store. Our tool helps the developer to properly specify the consistency level needed for their program by providing counterexamples that break the invariants if the chosen consistency is too weak. Finally, we apply our tool on several use cases from the SyncFree project [25], and we successfully detected bug-inducing schedules. Our contributions are summarized as follows:

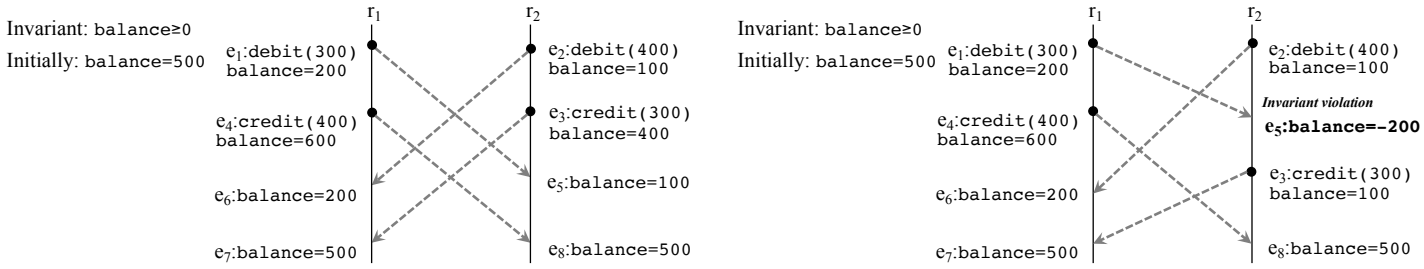
- We introduce and formalize a consistency-aware schedule exploration approach for weakly consistent systems that is parameterized by the scheduler and consistency model.
- We implement our approach in a prototype tool within the Antidote CRDT platform and include two schedule exploration strategies.
- We evaluate our tool on several benchmarks and show that it can efficiently find real bugs.

Our technical report provides more details on this work [12].

2. MOTIVATING EXAMPLE

We provide a virtual wallet example to explicate how an interleaving of a weakly consistent program, introduced by time nondeterminism, can result in an invariant violation. Our virtual wallet has a balance data field, defined as a CRDT counter, with an accompanying invariant of having a non-negative value at each replica. The balance can be updated using `credit` and `debit` events, where `debit` decrements the balance value by the specified amount only if the current balance is sufficient. We implemented the program using a causally consistent [1, 20] geo-replicated data store that guarantees the causal delivery of each event and convergence of the state in all replicas. Given the initial balance of 500 at every replica, Figure 1 gives two possible scheduling scenarios: one that satisfies and another that violates our invariant.

Figure 1a shows a bug-free scheduling scenario. Suppose two



(a) Bug-free scenario (b) Buggy scenario

Figure 1: Two scheduling scenarios in the virtual wallet example.

clients C_1 and C_2 are connected to two different replicas r_1 and r_2, respectively; the clients are issuing events to the same virtual wallet concurrently. First, C_1 debits 300 from the virtual wallet on r_1, thereby making the balance 200 (e_1). Then, C_2 debits 400 from the virtual wallet on r_2 (e_2) and credits 300, thereby making the balance 400 (e_3). Afterwards, C_1 credits 400 on r_1, and the balance becomes 600 (e_4). Now, r_1 propagates the C_1's first event to r_2, making the balance 100 (e_5); r_2 propagates both events issued by C_2 to r_1, which makes the balance 500 (e_6, e_7). Finally, the second event issued by C_1 is propagated to r_2, and the ending balance is 500 (e_8). In this scheduling scenario, the value of balance is always non-negative, and the state of both replicas converged in the end. Hence, a developer might think that the invariant always holds, while that is not the case, as our next scheduling scenario shows.

Figure 1b shows a buggy scheduling scenario, which starts the same as the bug-free one. First, C_1 debits 300 from r_1, making the balance 200 (e_1), and C_2 debits 400 from r_2, making the balance 100 (e_2). Then, C_1 credits 400, making the balance 600 on r_1 (e_4). Differently than in the bug-free scenario, but still allowed by weak consistency, r_1 now propagates C_1's first event to r_2, thereby making the balance value -200 (e_5). This violates our `balance>=0` invariant. Note that the two debit events e_1 and e_2 are concurrent. Due to the nondeterminism in weakly consistent systems, event e_5 can be received either before or after e_3; in fact, it can be received even before e_2! As shown in this schedule, if e_5 is scheduled right after e_2 and right before e_3, the program invariant is violated, although the schedule still guarantees causal consistency. Note that a scheduler guaranteeing a stronger consistency model (e.g., serializability) would fail to detect this bug. To catch such invariant violations, a developer has to take into consideration and be able to explore different orderings allowed under the given consistency model of the system. We address this need by providing a consistency-aware schedule exploration approach and a prototype implementation that helps developers discover scheduling scenarios leading to such deep-seated bugs. In this example, the invariant would be preserved if the balance is defined as a CRDT bounded-counter, which enforces strong consistency [22] on decrement operations.

3. WEAKLY CONSISTENT PROGRAMS

We formalize our approach based on the transactional consistency framework proposed by Cerone et al. [10]. Let $Rs = \{r_1, r_2, ..., r_n\}$ be the set of all replicas in the system and $n = |Rs|$ the total number of replicas. We define *Txns* as the set of messages (transactions) initiated by clients on replicas. We define *Logs* as the set of messages (transaction logs) transmitting between replicas in the system. Then, $Msgs = (Txns \cup Logs) \times Rs$ is the set of all messages transmitting between clients and replicas or between different replicas. For a message $msg = \langle t, r\rangle$, r denotes the originating replica of the transaction t. We formally define events (i.e., atomic steps in a program) as a set of tuples $Events = Msgs \times Rs \times \mathbb{Z}^n_{\geq 0}$. Each event consists of a message, a replica to which the message is being delivered, and a vector clock [15] denoting a snapshot of the system that captures message dependencies.

Let history $\mathcal{H} \subseteq \wp(Events)$ be the set of events $\{\langle msg, r, vc\rangle \mid vc \prec now^n\}$ that occurred in the system so far, where now^n denotes the current snapshot replica r has. So, the history at the initial state, denoted by $\mathcal{H}_0$, is an empty set. We define a commit time function $ct : Events \to \mathbb{Z}^n_{\geq 0}$, such that for every event $e = \langle\langle t, r'\rangle, r, vc\rangle$, $ct(e) = vc\big[r' \mapsto vc[r'] + 1\big]$ shows the visibility vector clock of e. Let *Obj* be the set of data store objects, and $obj : Events \to \wp(Obj)$ be a function mapping each event to a subset of objects that the event reads or updates. Then, we define function $relEvents : Events \times Rs \to \wp(Events)$ mapping every event e to a subset of events that act on at least one shared object as e does on the specified replica. For $e = \langle msg, r, vc\rangle$, *relEvents* is defined formally as $relEvents(e, r'') = \big\{\langle msg', r', vc'\rangle \mid r'' = r' \wedge obj(e) \cap obj(e') \neq \emptyset\big\}$.

3.1 Consistency Models

In this section, we introduce three well-known consistency models and formalize the dependency restrictions of each model. We informally specify the three models as follows:

Serializability Consistency (*SR*) guarantees that every transaction observes the effect of all other transactions updating shared objects before executing, and no such transactions are allowed to execute concurrently [22].

Eventual Consistency (*EC*) guarantees that the effect of a transaction is eventually transmitted and delivered to all other replicas [26].

Causal Consistency (*CC*) guarantees that the effect of a transaction is transmitted and delivered to every other replica after all of its dependencies (i.e., other transactions it depends on) have been delivered to that replica [1, 20].

To formalize these models, we first define a dependency function $updDep : CM \times Events \times \mathcal{H} \to Events$, where $CM = \{SR, EC, CC\}$ is the set of consistency models. Function *updDep* determines the dependency of an event by updating its vector clock based on the given system consistency model and history on which it is operating. Note that *updDep* is parameterized by the system consistency model. We also define a helper predicate $isAllowed : CM \times Events \times \mathcal{H} \to \mathbb{B}$ that determines if a given event is allowed to execute on its target replica under the specified consistency model, i.e., if all of events it depends on have already been executed.

In the *Causal Consistency* model, a transaction t depends on all transactions that update shared objects whose effects have been

seen by t. We define function *isAllowed* for event $e = \langle msg, r, vc\rangle$ where $msg = \langle t, r'\rangle$ under this consistency model as follows. Suppose $obsClock = \max_{e' \in relEvents(e,r)} ct(e')$ denotes the time when the related events are observable. Then,

$$isAllowed(CC, e, \mathcal{H}) = \begin{cases} true & vc \preceq obsClock \\ false & \text{otherwise.} \end{cases}$$

Finally, the *updDep* function for *Causal Consistency* is defined as follows:

$$updDep(CC, \langle msg, r, vc\rangle, \mathcal{H}) = \begin{cases} \langle msg, r, obsClock\rangle & isAllowed(CC, \langle msg, r, vc\rangle, \mathcal{H}) \\ \langle msg, r, vc\rangle & \text{otherwise.} \end{cases}$$

We provide the formalization of *SR* an *EC* models in our technical report [12].

3.2 Scheduler

In this section, we give a basic scheduler definition parameterized by a consistency model. A scheduler $M = \langle CM, D, \texttt{empty}, \texttt{give}, \texttt{take}\rangle$ is a tuple consisting of a consistency model CM, a datatype $D = \langle DS \times \mathcal{H}\rangle$ of scheduler objects (where DS is a datatype for maintaining scheduling events and set $\mathcal{H}$ is history as defined in the previous section), a scheduler constructor $\texttt{empty} \in D$, the function $\texttt{give} : D \times Events \to D$ that receives posted events, and the function $\texttt{take} : CM \times D \to \wp(D \times Events)$ that determines which event at which replica operates next.

For the given consistency model cm, the scheduler M is deterministic if for all $m \in DS$, $\texttt{take}(cm, \langle m, \mathcal{H}\rangle)$ has at most one element. It is non-blocking if all scheduled events are allowed, more formally if for all $e \in Events$ and $m, m' \in DS$:

$$\langle\langle m', \mathcal{H} \cup e\rangle, e\rangle \in \texttt{take}(cm, \langle m, \mathcal{H}\rangle) \implies isAllowed(cm, e, \mathcal{H}).$$

Definition 1. (Bag Scheduler) The multiset-based scheduler `bag` is defined on the multiset domain D_{bag} of events as

$$\begin{aligned} \texttt{empty}_{\texttt{bag}} &:= \emptyset \\ \texttt{give}_{\texttt{bag}}(\langle m, \mathcal{H}\rangle, e) &:= \langle m \cup \{e\}, \mathcal{H}\rangle \\ \texttt{take}_{\texttt{bag}}(cm, \langle m, \mathcal{H}\rangle) &:= \{\langle\langle m \backslash \{e\}, \mathcal{H} \cup \{e\}\rangle, e\rangle \mid e \in m\}. \end{aligned}$$

Accordingly, $\texttt{take}_{\texttt{bag}}$ returns a set of allowed events and thus the bag scheduler is nondeterministic.

4. SCHEDULING STRATEGIES

In this section, we propose two scheduling strategies for weakly consistent programs. Later in Section 5 we empirically evaluate and compare the two strategies.

4.1 Random Scheduling

We define a random scheduler, which randomly exercises possible program schedules. When an event is posted, it is added to a bag of events. Then, the random scheduler randomly selects and dispatches one of the legal events in the bag. We formally define such a random scheduler as a tuple $M = \langle CM, D_{bag}, \texttt{empty}_{\texttt{bag}}, \texttt{give}_{\texttt{bag}}, \texttt{take}_{\texttt{bag}}\rangle$, and we call it the ***C**onsistency-**A**ware **R**andom* (CAR) scheduler. The scheduler proceeds if the current event either (1) completes its operation or (2) is not allowed.

Definition 2. (Bag-based CAR Scheduler) Let *BCAR* be a bag-based scheduler defined as a tuple: $BCAR = \langle CM, (\{Events\} \times \wp(Events)), \langle \epsilon, \mathcal{H}_0\rangle, \texttt{give}_{\texttt{bag}}, \texttt{take}_{\texttt{bag}}\rangle$.

Let m, m' be two bags, where m maintains all events to be scheduled, and m' maintains all legal events with respect to the current history $\mathcal{H}$ and under the specified consistency model. Suppose *output* is a subset of $D \times Events$, and $e = \langle msg, r, vc\rangle$ where $msg = \langle t, r'\rangle$, such that t is in either *Txns* or *Logs*. Function $\texttt{give}_{\texttt{bag}}$ takes a scheduler object $\langle m, \mathcal{H}\rangle$ and an event e as the input, and then it updates the scheduler to $\langle m \cup \{e\}, \mathcal{H}\rangle$. Function $\texttt{take}_{\texttt{bag}}$ takes the underlying consistency model cm and a scheduler object $\langle m, \mathcal{H}\rangle$ as the input. If either m is an empty bag or there is no legal event e in m for the specified history $\mathcal{H}$, no event is scheduled, i.e., $\texttt{take}_{\texttt{bag}}$ returns an empty set. Otherwise, all legal events in m with respect to cm and $\mathcal{H}$ are maintained in m'. Thereby, for every event $e = \langle\langle t, r'\rangle, r, vc\rangle$ in m', $\texttt{take}_{\texttt{bag}}$ does the following: (1) updates the dependency of e if $t \in Txns$, according to $updDep(cm, e, \mathcal{H})$ as defined in Section 3; (2) adds e to the history $\mathcal{H}$; (3) adds the tuple $\langle\langle m \backslash \{e\}, \mathcal{H}\rangle, e\rangle$ to the *output* set; and (4) returns the set *output*.

4.2 Delay-bounded Scheduling

Delay-bounded scheduling as introduced by Emmi et al. [14] parameterizes a program search space by a deterministic scheduler and delay bound k. A k-delay bounded scheduler generates different schedules by delaying the execution of up to k events in the deterministic scheduler.

In this paper, we propose a delay-bounded scheduler that is aware of the consistency model of the underlying data store. In so doing, to limit the nondeterminism in the default scheduler, we employ a deterministic scheduler, and explore a limited number of deviations from that deterministic schedule. We define such delaying scheduler as $M = \langle CM, D, \texttt{empty}, \texttt{give}, \texttt{take}, \texttt{delay}\rangle$. The function $\texttt{delay} : D \times Events \to D$ allows the scheduler to postpone the execution of an event. When an event is posted, it is enqueued, and its execution could be postponed at the dispatch time. We call such a scheduler, augmented with delay function, the ***C**onsistency-**A**ware **D**elay-bounded* scheduler (CAD). The scheduler advances to the next event when the current event either (1) completes its operation, (2) is not allowed, or (3) is delayed. An execution is k-CAD when the number of delay operations in that execution is at most k.

Definition 3. (List-based CAD Scheduler) Let *LCAD* be the list-based delaying scheduler defined as a tuple: $LCAD = \langle CM, Events^* \times Events^* \times \mathbb{Z}_{\geq 0} \times \wp(Events), \langle \epsilon, \epsilon, 0, \mathcal{H}_0\rangle, \texttt{give}, \texttt{take}, \texttt{delay}\rangle$.

Let $\mathcal{H}$ be a set of events, denoting the history of the system, and m_r and m_d be two lists, where m_r maintains the events to be scheduled and m_d delayed events. Also, let event $e = \langle msg, r, vc\rangle$, where $msg = \langle t, r'\rangle$. Function `delay` takes a scheduler object $\langle m_r, m_d, i, \mathcal{H}\rangle$ and an event e as the input. Then, it delays the execution of e by appending it to m_d and returns the scheduler object $\langle m_r, m_d \oplus_l, i, \mathcal{H}\rangle$, where l is the length of m_d, and $\oplus_l$ operator inserts e to m_d at the position l (i.e., at the end of m_d). Function `give` takes a scheduler object $\langle m_r, m_d, i, \mathcal{H}\rangle$ and an event $e = \langle\langle t, r'\rangle, r, vc\rangle$ as the input. If $t \in Txns$, it inserts t in m_r at the position i and increments i; otherwise, if $t \in Logs$, it appends t to m_r. In the end, it returns the scheduler with the updated m_r, m_d, and i. Function `take` accepts the underlying consistency model and a scheduler object as the input. If either both m_r and m_d are empty lists or there is no legal event in m_r with respect to the specified consistency model and the current history $\mathcal{H}$ of the system, then no event is scheduled and an empty set is returned. Otherwise, if all events in m_r have been either

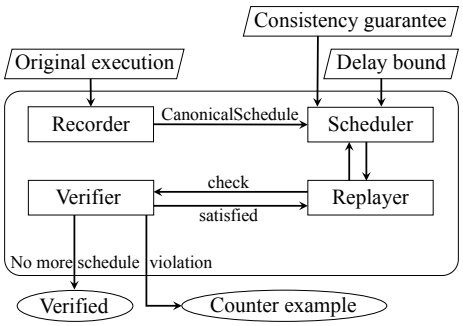


Figure 2: Overview of the COMMANDER architecture.

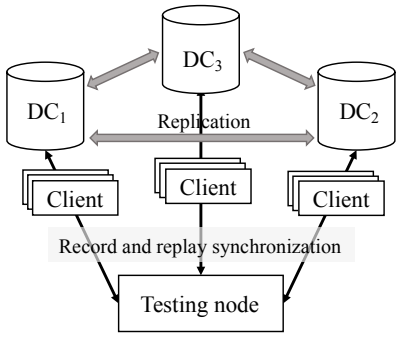


Figure 3: Testing environment for empirical evaluation.

scheduled or delayed, the scheduler substitutes m_r with m_d and m_d with an empty list and also sets i to 1. Then, while $m_r[i]$ is not a legal event, it delays $m_r[i]$ and increments i. Considering $m_r[i] = \langle\langle t, r'\rangle, r, vc\rangle$ as a legal event, this function first updates the dependency of $m_r[i]$ using $updDep(cm, m_r[i], \mathcal{H})$, if $t \in Txns$. Then, it adds $m_r[i]$ to the history $\mathcal{H}$ and returns a set consisting of a single tuple of the updated scheduler object and $m_r[i]$.

5. EMPIRICAL EVALUATION

We implement the proposed schedule exploration strategies in a prototype stateless model checker for weakly consistent programs named COMMANDER [11]. As shown in Figure 2, COMMANDER consists of four components: (1) *Recorder* is responsible for recording the events that occur during the execution of the test scenario written by the developer (the recorded sequence, called *CanonicalSchedule*, is a deterministic canonical schedule); (2) *Scheduler* reorders the events in *CanonicalSchedule*, using the selected scheduling strategy, which is currently either CAR or CAD; (3) *Replayer* exercises the events in the ordering that *Scheduler* provides; and (4) *Verifier* checks for program-specific invariant violations after each scheduled event is replayed. If invariants are not violated, *Replayer* replays the next scheduled event and so on. Otherwise, *Verifier* provides a counterexample to the developer.

We empirically evaluate our approach using one real world and three synthetic benchmarks. Since Antidote is a new data store, there is only one real world benchmark written for it to date, called *FMK Medical Application*. *FMK Medical Application* shares a medical profile among different health institutions. The invariant we check for this benchmark is that every prescription must be present in the corresponding patient's prescription list. In addition, we develop three benchmarks after the realistic use cases from the SyncFree project [25]. *Virtual Wallet* manages virtual economies of distributed computer game clients. Each client maintains a local state and issues credits and debits to it (see Section 2). The invariant we check for this benchmark is that the balance must not become negative. *Ad Counter* implements a distributed counter. Advertising platforms keep track of impressions and clicks for ads in order to analyze advertising data. The invariant we check for this benchmark is that the number of ad views must not exceed the upper bound. *Business to Business (B2B) Order* plays the role of a traveling salesman for large manufacturers. Client store employees can see a catalogue of products and place orders from a mobile device using a B2B order program, concurrently. The invariant we check for this benchmark is that the store budget must not become negative.

We perform our experiments in a testing environment with a topology consisting of three data centers (DCs) as shown in Figure 3. We create a testing node that hosts COMMANDER and communicates with every DC and client to record and replay events as described earlier in this section. We set up multiple DCs connected using TCP/IP protocol on a single 4.00 GHz Intel Core i7 machine with 62 GB of memory.

To evaluate the effectiveness of our approach in detecting invariant violations and to empirically compare the different scheduling strategies, we seed a bug in each of our three synthetic benchmarks. Then, we use COMMANDER to discover the seeded bugs using the proposed CAR and CAD schedulers. However, in our realistic benchmark, FMK, we found a real bug which CAD scheduler with delay bound of 0 missed it. The FMK system allows updating an entity, e.g., patient information, using its ID, even if that patient does not exist in a DC. Therefore, creating that patient later in a remote DC, after the update has been delivered, fails. The developers of the FMK system fixed this bug after we reported it.

Table 1 shows our experimental result. Given the inherent randomness of CAR, we run it 15 times with different random seeds on every benchmark, and we report min, max, median, and mean values for the numbers of explored schedules and runtimes. As the results show, CAR can generate buggy schedules almost immediately, but there is also a great variation in its effectiveness, which makes it brittle.

We implement two variations of our CAD scheduler. The first one is called for-CAD, and it delays events starting from the beginning of the canonical schedule. The second one is called back-CAD, and it delays events starting from the end of the canonical schedule. Table 1 shows the result we obtain using these two CAD scheduler variations. As the results show, the for-CAD variation finds bugs faster than the back-CAD variation. Since the events coming from clients are being executed first according to our canonical schedule and events' dependencies are assigned only when they are coming from clients, for-CAD quickly alters the dependencies between events, which makes buggy schedules more likely to be caught. If we compare the for-CAD variation against the CAR scheduler, we notice that their effectiveness is comparable, while for-CAD had the advantage of being predictable.

6. RELATED WORK

The most related approach to ours proposes a form of consistency called *explicit consistency* [4, 17, 21]. Similarly to our work, users can specify the required consistency model, and unsafe operations are identified under concurrent executions using program-specific invariants. However, the consistency rules must be manually specified using additional program-specific invariants. Hence, the correctness of the approach relies on the correctness of the provided consistency rules. On the other hand, we guarantee the selected consistency model of the underlying data store and require users

Table 1: Experimental results. Column *Txns* is the number of transactions in a benchmark; *Events* is the number of events; *Time* is runtime in min:sec; *#s* is the number of schedules explored by COMMANDER before it discovers a bug. With k we denote the delay bound, and with * we denote runs where COMMANDER misses a bug because of an insufficient delay bound.

			CAR Scheduler								**for-CAD Scheduler**						**back-CAD Scheduler**					
			min		**max**		**median**		**mean**		**k=0**		**k=1**		**k=2**		**k=0**		**k=1**		**k=2**	
Benchmark	**Txns**	**Events**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**
Virtual Wallet	30	90	1	1	37	42:06	8	9:10	11	12:31	1*	1:14	5	5:30	182	208:45	1*	1:15	54	61:32	53	60:04
Ad Counter	6	18	1	0:21	10	3:32	5	1:46	5	1:45	1*	0:26	8	2:48	10	3:27	1*	0:25	7	2:22	7	2:21
B2B Order	18	54	1	0:42	22	16:23	4	2:55	6	4:29	1*	0:50	7	5:07	173	128:52	1*	0:51	31	23:07	30	22:21
FMKe	70	210	1	0:21	11	27:07	4	8:07	4	8:00	1*	2:41	5	13:22	1	2:39	1*	2:39	>127	>328:00	>127	>328:00

to specify only program-specific invariants.

When it comes to checking of weakly consistent programs, ECRacer is a dynamic analysis tool that checks serializability of weakly consistent programs. It first records an execution of such a program and then performs an offline analysis to check for serializability. It does not take the dependency between user-initiated transactions into consideration, and therefore it can report false positives. Bouajjani et al. [6] propose a set of *bad patterns* to check causal consistency, causal memory, and causal convergence of an execution. If an execution contains a bad pattern with respect to a replicated data type, it is not consistent.

In a recent effort, Zeller et al. propose a verification framework called Repliss [28], which includes a property-based testing engine [27] to check program specific invariants of programs built on top of weakly consistent data stores. The testing engine is a model of the underlying data store schema, and it randomly exercises different executions of a given program. Lesani et al. propose Chapar [19], which includes a model checker targeting weakly consistent programs. Their work addresses an abstract model of programs in contrast to our work that performs execution-based model checking. Kim et al. [18] propose a consistency oracle that simulates a distributed data store. The proposed consistency oracle supports neither causal consistency nor transactions which are being widely used in different data stores.

Acknowledgments. This research is supported in part by European FP7 project 609 551 SyncFree (2013–2016).

7. REFERENCES

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[2] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, 2016.

[3] *Antidote Reference Platform.* `http://github.com/SyncFree/antidote`.

[4] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *EuroSys*, 2015.

[5] C. Baquero, P. S. Almeida, and A. Shoker. Making operation-based crdts operation-based. In *PaPEC*, 2014.

[6] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza. On verifying causal consistency. In *POPL*, 2017.

[7] L. Brutschy, D. Dimitrov, P. MÃijller, and M. Vechev. Serializability for eventual consistency: Criterion, analysis, and applications. In *POPL*, 2017.

[8] S. Burckhardt. *Principles of Eventual Consistency*. 2014.

[9] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *POPL*, 2014.

[10] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*, 2015.

[11] *Commander*. `http://github.com/Maryam81609/commander`.

[12] M. Dabaghchian, Z. Rakamarić, B. K. Ozkan, E. Mutlu, and S. Tasiran. Consistency-aware scheduling for weakly consistent programs. Technical Report UUCS-17-002, University of Utah, 2017.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[14] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *POPL*, 2011.

[15] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.

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

[17] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *POPL*, 2016.

[18] B. H. Kim, S. Oh, and D. Lie. Consistency oracles: Towards an interactive and flexible consistency model specification. In *HotOS XVI*, 2017.

[19] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *POPL*, 2016.

[20] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP*, 2011.

[21] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. The cise tool: Proving weakly-consistent applications correct. In *PaPoC*, 2016.

[22] C. H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 1979.

[23] *Riak - A Key-Value Store*. `http://basho.com/products/riak-overview`.

[24] M. Shapiro, N. M. PreguiÃğa, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[25] *SyncFree Project*. `https://syncfree.lip6.fr`.

[26] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[27] P. Zeller. Testing properties of weakly consistent programs with repliss. In *PaPoC*, 2017.

[28] P. Zeller and A. Poetzsch-Heffter. Towards a proof framework for information systems with weak consistency. In *SEFM*, 2016.