Verifying Pipelined-RAM Consistency over Read/Write Traces of Data Replicas

Hengfeng Wei, Marzio De Biasi, Yu Huang, *Member, IEEE*, Jiannong Cao, *Fellow, IEEE*, and Jian Lu

Abstract—Data replication technologies in distributed storage systems introduce the problem of data consistency. For high performance, data replication systems often settle for weak consistency models, such as Pipelined-RAM consistency. To determine whether a data replication system provides Pipelined-RAM consistency, we study the problem of *verifying Pipelined-RAM consistency* over read/write traces (VPC, for short). Four variants of VPC (labeled VPC-SU, VPC-MU, VPC-SD, and VPC-MD) are identified according to whether there are Multiple shared variables (or one Single variable) and whether write operations can assign Duplicate values (or only Unique values) to each shared variable. We prove that VPC-SD is NP-complete (so is VPC-MD) by reducing the strongly NP-complete problem 3-Partition to it. For VPC-MU, we present the Read-Centric algorithm with time complexity $O(n^4)$, where n is the number of operations. The algorithm constructs an operation graph by iteratively applying a rule which guarantees that no overwritten values can be read later. It incrementally processes all the read operations one by one, and exploits the total order between the dictating writes on the same variable to avoid redundant applications of the rule. The experiments have demonstrated its practical efficiency and scalability.

Index Terms—Pipelined-RAM, consistency model, verification, replication

1 Introduction

Data replication consists of maintaining multiple copies of data, called replicas, on separate computing entities. It is a critical enabling technology in distributed systems, improving system performance (e.g., latency), reliability, and scalability [1], [2], [3]. Practically, it is desirable for a data replication system to achieve three properties simultaneously, namely data consistency (C), availability (A), and partition-tolerance (P) [4]. However, this has been theoretically proved impossible by the CAP theorem [5]. Furthermore, as soon as a distributed system replicates data, a tradeoff between consistency and latency arises [6]. Consequently, stronger consistency generally comes along with reduced availability and lower performance. For better availability and performance, modern commercial data replication systems often choose to sacrifice consistency. To this end, researchers have developed various weak consistency models such as Pipelined-RAM consistency [7], cache consistency [8] (a.k.a. memory coherence [9]), causal consistency [10], and eventual consistency [11], besides the strong ones such as linearizability [12] (a.k.a. atomicity [13]) and sequential consistency [14]. For example, Yahoo!'s PNUTS [2] provides per-record timeline consistency. Amazon's Dynamo [1] only promises eventual consistency. Nowadays, the weak consistency models have been widely used in distributed systems, with the prevalence of cloud data storage services, mobile devices, and wireless communications.

In this work, we focus on Pipelined-RAM consistency [7], one of the well-known weak consistency models. Informally, a read/write trace satisfies Pipelined-RAM consistency if and only if write operations performed by a single process are observed by all the other processes in the order they were issued, whereas write operations from different processes may be observed in different orders by different processes [15]. Pipelined-RAM consistency does not require *all* the processes to agree on the same view of the order in which operations occur, and thus allows high-performance implementations. On the other hand, it requires the operation orders on the *per-process* basis (i.e., program order formally defined in Section 2). Therefore, it can provide per-session guarantees for users, including Read-Your-Writes, Monotonic-Reads, and Monotonic-Writes [16], [11]. As is pointed out in [2], managing session state is a common task across many different web applications. The per-session guarantees (thereby Pipelined-RAM consistency) have been developed in the context of the Bayou project at Xerox PARC [16], which aimed to build a replicated storage system for mobile computing users who may only be intermittently connected. To demonstrate its practical usefulness of Pipelined-RAM consistency, we discuss three motivating examples, *one for each per-session guarantee*.

First, we consider the baseball application described in [17]. Being the only person who updates the score, the official scorekeeper can rely on the *Read-Your-Writes* guarantee to retrieve the most up-to-date previous score and update it. The Read-Your-Writes guarantee can be enforced by Pipelined-RAM consistency due to its program order constraint.

• *H. Wei, Y. Huang, and J. Lu are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China. E-mail: hengxin0912@gmail.com, {yuhuang, lj}@nju.edu.cn.*
• *Marzio De Biasi is with the "Computational Complexity, Puzzles and Machines" organization, http://www.nearly42.org/, Italy. E-mail: marziodebiasi@gmail.com.*
• *J. Cao is with the Hong Kong Polytechnic University. E-mail: csjcao@comp.polyu.edu.hk.*

Manuscript received 30 Dec. 2014; revised 15 May 2015; accepted 25 June 2015. Date of publication 7 July 2015; date of current version 13 Apr. 2016. Recommended for acceptance by G. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2453985

Second, we consider a replicated e-mail system which runs on a weakly consistent replicated database [16]. The user Bob first issues a Read request to retrieve a summary of new mail messages, and then issues another Read request to show one of these messages. The *Monotonic-Reads* guarantee can be used to avoid the anomaly in which Bob is incorrectly informed that the message does not exist. The Monotonic-Reads guarantee can be enforced by Pipelined-RAM consistency due to the read-from relation and its program order (between these two Read requests) constraint.

Third, we consider a source-code repository, which is replicated among multiple servers [16]. Suppose that the programmer Bob first updates a library to add functionality in an upward compatible way. This new library is propagated to other servers in the background. Bob then updates an application, which relies on the newly updated library. In this scenario, the code will not be compiled successfully on the servers which receive the new application code before receiving the new library code. This annoying problem can be solved by the *Monotonic-Writes* guarantee, which is enforced by Pipelined-RAM consistency because Pipelined-RAM consistency requires write operations performed by a single process (e.g., server in the scenario) to be observed by all the other processes in the order they were issued.

Service Level Agreement (SLA, for short) [1] between users and a storage service requires the service provider to offer some consistency model as negotiated. Otherwise, the users have the right to claim a refund. A major challenge to the service provider is that a theoretically proved correct protocol may suffer from flawed implementations and unexpected runtime failures. Verifying consistency models serves as a complementary means to help the service provider boost their confidence regarding the correctness of the implementation. On the other hand, the implementation of such a system, when published as commercial web services, is often inaccessible to users. As a result, the users can only test the system by collecting its logs (i.e., read/write traces of operations) and verifying whether it is indeed delivering the promised consistency model [18].

Though weak consistency models are regarded important, to the best of our knowledge, their verification problems have not been sufficiently studied yet. In this work, we systematically study the problem of *verifying Pipelined-RAM consistency over read/write traces* (VPC, for short). Specifically,

- First, we identify four variants of VPC according to *a*) whether there are Multiple shared variables (or one Single variable), *and b*) whether write operations can assign Duplicate values (or only Unique values) to each shared variable; the four variants are labeled VPC-SU, VPC-MU, VPC-SD, and VPC-MD (Section 2). For the VPC problem, all read operations are totally ordered on the same process p_0.
- Second, we prove that VPC-SD is NP-complete (so is VPC-MD) by reducing the strongly NP-complete problem 3-PARTITION [19] to it (Section 3).
- Third, we present a polynomial algorithm, called READ-CENTRIC, for VPC-MU (Section 4). The READ-CENTRIC algorithm constructs an operation graph by iteratively applying a rule which guarantees that no overwritten values are allowed to be read. The key point is that for VPC-MU, the total order between all read operations on the same variable leads to a total order between their corresponding (*dictating*) writes. The READ-CENTRIC algorithm makes use of the total order between these writes to avoid redundant applications of the rule, achieving the time complexity of $O(n^4)$, where n is the number of operations in the trace. Moreover, the READ-CENTRIC algorithm is *incremental* in the way that it processes all the read operations on process p_0 one by one (hence it is named "READ-CENTRIC").

TABLE 1
Notations Used in the VPC Problem

Notation	Explanation
o	operation
r, w	read / write operation
n	number of operations in a trace
$w = D(r)$	w is the dictating write of the read operation r
$\prec_{PO}, \preceq_{PO}$	program order and its reflexive closure
$\prec_{WR}$	write-to order
$\prec_{W'W}$	w'wr order
$\prec, \preceq$	precedence relation and its reflexive closure
$r_{\Downarrow}$	r-downset: the set of operations preceding r, plus r itself
r_{δ}	r-delta: relative complement of $r'_{\Downarrow}$ w.r.t $r_{\Downarrow}$ i.e., $r_{\Downarrow} \setminus r'_{\Downarrow}$); here r' is the previous read of r
$RR[w]$	*ReachableRead*: the *first* read a write operation w can reach via the precedence relation $\prec$
PW of o	*PrecedingWrite*: summary of the dictating writes preceding o, one per variable
$\mathcal{G}$	operation graph constructed in the RW-CLOSURE algorithm and the READ-CENTRIC algorithm
$o_1 \rightarrow o_2$	edge from operation o_1 to operation o_2 in $\mathcal{G}$
$o_1 \rightsquigarrow o_2$	path from operation o_1 to operation o_2 in $\mathcal{G}$
$\pi_{\mathcal{G}}$	a legal schedule constructed from $\mathcal{G}$ via a specific topological sorting

The rest of this paper is organized as follows. Section 2 defines the problem of verifying Pipelined-RAM consistency over read/write traces and its four variants. Section 3 gives the NP-completeness proof of VPC-SD (so is VPC-MD). Section 4 presents the polynomial READ-CENTRIC algorithm for VPC-MU. Section 5 experimentally evaluates the practical efficiency and scalability of the READ-CENTRIC algorithm. Section 6 discusses the related work, and Section 7 concludes the paper.

2 PROBLEM DEFINITION

In this section, we first define read/write traces of data replicas and Pipelined-RAM consistency, and then define the problem of verifying Pipelined-RAM consistency over read/write traces. Notations used in this work are listed in Table 1.

2.1 Read/Write Trace

We model the data replicas as a collection of shared variables supporting read/write operations, and the computing entities as a collection of processes.

An operation o is a tuple $(t, p, v, d) \in \{R, W\} \times P \times V \times D$ where,

- $t \in \{R, W\}$ is the type of operation (R for read and W for write);
- $p \in P$ is the process issuing the operation;
- $v \in V$ is the variable involved in the operation;
- $d \in D$ is the returned value (for read) or written value (for write) on the variable v.

For an operation $o = (t, p, v, d)$, the process is denoted by $p(o)$. The variable and the value involved are denoted by $var(o)$ and $val(o)$ respectively. Generally, we use o for any operation, r for any read operation, w for any write operation, O for the set of all operations, R for the set of all read operations, W for the set of all write operations, and W_v for the set of all write operations on the same variable v.

There are two basic partial orders over operations. Program order, denoted by $\prec_{PO}$, is the order in which operations are issued by each process. We employ $\preceq_{PO}$ to denote the reflexive closure of $\prec_{PO}$. Write-to order, denoted by $\prec_{WR}$, associates each read with a write from which it reads the value. Formally,

Definition 2.1 (Write-to Order ($\prec_{WR}$)). *Write-to order, $\prec_{WR}$, is a partial order satisfying the condition that for each read r, there exists a unique write w with $var(w) = var(r)$ and $val(w) = val(r)$ such that $w \prec_{WR} r$.*

A *read/write trace* of data replicas comprises multiple process histories, each of which consisting of a finite sequence of read and write operations in the program order. Fig. 2 in Section 4.1 shows a trace consisting of four processes, where the operation, for instance, $(p_0, W, y, 1)$ is abbreviated by $Wy1$.

2.2 Pipelined-RAM Consistency Model

The Pipelined-RAM consistency model is one of the well-known weak consistency models [7], [15]. Informally, a read/write trace satisfies Pipelined-RAM consistency if and only if write operations performed by a single process are observed by all other processes in the program order they were issued, whereas write operations from different processes may be observed in different orders by different processes [15].

There are two key points to explain. First, Pipelined-RAM consistency is weak because it does not require all the processes to agree on the same view of the order in which operations occur. Thus, each process can be checked against Pipelined-RAM consistency separately. Second, the operations *visible* to each process p are all write operations and its own read operations, while ignoring read operations from other processes. Notice that for process p, its *visible* read operations are on the same process (i.e., p itself).

Pipelined-RAM consistency is formally defined in terms of schedules. A *schedule* π of a set of operations is a permutation of them. Given a schedule, the *precedence relation* over the operations in it is denoted by '$\prec$'. We employ $\preceq$ to denote the reflexive closure of $\prec$. Moreover, we define $\min(o_1, o_2) = o_1$ if $o_1 \preceq o_2$.

A schedule is said to *respect* some partial order if and only if it is a linearization of the partial order. A schedule π is *legal* if and only if each read reads the value from the most recently preceding write in π on the same variable.

TABLE 2
A Summary of Complexity Results for VPC Problem
([$*$] : new results)

	(S)ingle variable	*(M)ultiple variables*
write (D)uplicate values	VPC-SD (NPC) [$*$]	VPC-MD (NPC) [$*$]
write (U)nique value	VPC-SU (P) [18]	VPC-MU (P) [$*$]

Definition 2.2 (Pipelined-RAM Consistency). *A read/write trace satisfies Pipelined-RAM consistency if and only if for each individual process, there exists a* legal schedule *of its* visible *operations, respecting both program order and write-to order.*

According to Definition 2.2, we can verify each process against Pipelined-RAM consistency separately. In the remainder of this paper, we consider the verification problem with respect to some particular process and distinguish it with p_0.

2.3 The Problem of Verifying Pipelined-RAM Consistency

The problem of Verifying Piplined-RAM Consistency (VPC, for short) over read/write traces is defined as a decision problem.

Definition 2.3 (Verifying Pipelined-RAM Consistency Problem).

- *INSTANCE: A read/write trace T. Its size (denoted by n) is defined as the total number of operations in it.*
- *QUESTION: Does T satisfy Pipelined-RAM consistency?*

Following the terminology in [20], we identify four variants of the general VPC problem from two orthogonal dimensions: *a*) whether there are Multiple shared variables (or one Single variable), *and b*) whether write operations can assign Duplicate values (or only Unique values) to each shared variable. The case of "writing unique values" means that any two write operations on the same variable must write different values to it. That is, every value written to the variable now must have never been assigned to it before. The four variants are labeled VPC-SU, VPC-MU, VPC-SD, and VPC-MD.

As summarized in Table 2, the VPC problem turns out to be NP-complete if write operations can assign duplicate values to each shared variable. Notice that in this case, there may be non-determinism in the write-to order for some read operations. In contrast, it is polynomially tractable if only unique values are allowed. Specifically, we first prove that VPC-SD is NP-complete (so is VPC-MD) by reducing the strongly NP-complete problem 3-PARTITION [19] to it (Section 3). On the other hand, we present a polynomial algorithm, called READ-CENTRIC with time complexity $O(n^4)$, for VPC-MU (Section 4). Notice that the VPC-SU variant can be solved in polynomial time, following from [18].

3 The VPC-SD and VPC-MD Problems are NP-Complete

In this section, we show that the VPC-SD problem is NP-complete (so is VPC-MD) by reducing the strongly NP-complete problem 3-Partition [19] to it.

Definition 3.1 (*3-Partition* **).**

- *INSTANCE: Set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$, and a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$ such that $\sum_{a \in A} s(a) = mB$.*
- *QUESTION: Can A be partitioned into m disjoint sets $A_1, A_2, \ldots, A_m$, such that each A_i contains exactly three elements from A, and for $1 \le i \le m$, $\sum_{a \in A_i} s(a) = B$?*

Notice that the definition of 3-Partition adopted here is a generalization of that in [19] (by removing $B/4 < s(a) < B/2$), and thus is NP-complete. We choose to reduce from 3-Partition because it is NP-complete even if the inputs $a \in A$ and B are provided in unary [19]. We use the Unary 3-Partition problem.

Theorem 3.1. *VPC-SD is* NP*-complete.*

Proof. *VPC-SD is in* NP*:* Given a schedule of the VPC-SD instance, it is straightforward to check whether it is legal by scanning it in polynomial time.

VPC-SD is NP*-hard:* To show that VPC-SD is NP-hard, we shall give a polynomial reduction from Unary 3-Partition to it. Let $A = \{a_1, a_2, \ldots, a_{3m}\}$, $B \in \mathbb{Z}^+$ (given in unary), and $s(a_1), s(a_2), \ldots, s(a_{3m}) \in \mathbb{Z}^+$ (given in unary) constitute an arbitrary instance of Unary 3-Partition. In the corresponding VPC-SD instance, we assume that integers a, a', b, b', c, c' used as variable values are distinct. Here Wxa (Rxa) denotes the operation of writing (reading) value a to (from) variable x.

The basic idea of the reduction is straightforward: when a schedule encounters a read sequence like "$Rxa\ Rxa'$", even if the last write of x before the sequence is a Wxa', the Rxa operation forces the schedule to "use" another Wxa' to satisfy the Rxa'.

We represent each $a_i \in A$ with a process P_{a_i} consisting of $a_i + 2$ write operations: the first operation is a write operation Wxa' (red boxes in Fig. 1), followed by a_i write operations Wxb' (blue boxes), followed by a single write operation Wxc' (cyan boxes). The middle a_i write operations Wxb' are to simulate the element $a_i \in A$ in *unary*. The (non-deterministic) choice of the element $a_i \in A$ is simulated by scheduling the front Wxa' and the Wxc' behind.

We then add three auxiliary processes $P_{c_1}, P_{c_2}, P_{c_3}$. Specifically, P_{c_1} comprises $3m$ write operations Wxa. P_{c_2} comprises $mB = \sum_{a \in A} s(a)$ write operations Wxb. P_{c_3} comprises $3m$ write operations Wxc.

Now we construct the process P_0 consisting only of read operations by concatenating m *slot sequences*; each slot sequence is made of:

- a leading *open subsequence* "$Rxa\ Rxa'\ Rxa\ Rxa'\ Rxa\ Rxa'$", that forces to pop three operations Wxa' from three distinct P_{a_i} and open those processes;

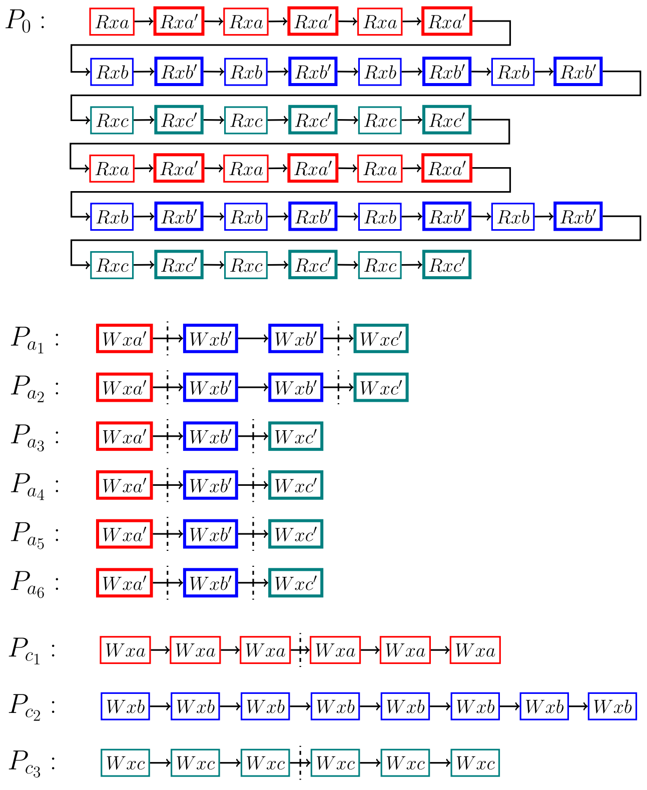


Fig. 1. The VPC-SD trace corresponding to an instance of the Unary 3-Partition problem (in which $A = \{2, 2, 1, 1, 1, 1\}, m = 2,$ and $B = 4$) obtained with the reduction of Theorem 3.1.

- followed by a *sum subsequence* "$Rxb\ Rxb'$" repeated B times, that forces to pop B operations Wxb' from the processes that are currently open;
- followed by a trailing *close subsequence* "$Rxc\ Rxc'\ Rxc Rxc'\ Rxc\ Rxc'$", that forces to pop three operations Wxc' from the end of the processes that are currently open.

Interacting with processes P_{a_i}, the leading *open subsequence* and the trailing *close subsequence* collectively complete the choice of three elements from A, and the *sum subsequence* makes sure that they sum to the target B.

Fig. 1 shows an example of the VPC-SD trace corresponding to the Unary 3-Partition instance in which $A = \{2, 2, 1, 1, 1, 1\}, m = 2,$ and $B = 4$.

The reduction is polynomial: The size (i.e., total number of operations) of the VPC-SD instance is

$$\underbrace{(6 + 2B + 6)m}_{P_0} + \underbrace{(6m + Bm)}_{P_{a_i}} + \underbrace{3m + Bm + 3m}_{P_{c_1}, P_{c_2}, P_{c_3}}$$
$$= 24m + 4Bm.$$

The a_i's and B are given in unary, so it is polynomial in m and B and the reduction is polynomial.

We now prove that the Unary 3-Partition instance has a solution if and only if the VPC-SD instance has a solution.

($\Rightarrow$) *If the* Unary 3-Partition *instance has a solution* A_1, $A_2, \ldots, A_m$, we construct a legal schedule π for the VPC-SD instance. Let the elements of A_i be $a_{i_1}, a_{i_2}, a_{i_3}$ (in unary). Each A_i corresponds to a subsequence π_i of π in the following way: P_0 uses the leading *open subsequence* of its ith *slot sequence* to open each process of $P_{a_{i_1}}, P_{a_{i_2}}$, and $P_{a_{i_3}}$ by using its Wxa', meanwhile "consuming" three Wxa from process P_{c_1}. The following *sum subsequence* completes the B write operations Wxb' from the three currently open processes and B write operations Wxb from P_{c_2}. Finally, the trailing *close subsequence* is scheduled together with B write operations Wxc' from the three currently open processes and B write operations Wxc from P_{c_3}. It is straightforward to ensure that the schedule is legal during this construction.

($\Leftarrow$) *If the VPC-SD instance has a legal schedule* π, we construct a solution to the UNARY 3-PARTITION instance. Notice that in π, read operations and write operations must be scheduled *alternately*; otherwise write operations would run out and some read operations were left unscheduled. Thus for each *slot sequence* of P_0, P_0 has to first use its leading *open subsequence* to open three processes of the m unary P_{a_i}. We claim that the total number of Wxb' in the three opened processes equals B. Otherwise, there are two cases: *1*) The total number of Wxb' is greater than B. This means that a process is opened, the corresponding *sum subsequence* of P_0 is consumed, and some Wxb' are still there. In order to complete the current trailing *close subsequence*, we pop them (without corresponding Rxb') to reach the final Wxc'. However, in one of the next *slot sequences* there will be not enough Wxb' to schedule and to reach its *close subsequence*. *2*) The total number of Wxb' is less than B. This means that we are in the middle of a *sum subsequence* and we need a Wxb', but we have already reached the end of all the currently opened processes. We cannot open another process to recover a Wxb' to complete the *sum subsequence*. Otherwise in one of the next *slot sequences* there will be not enough Wxa' to complete an *open subsequence*. Thus, VPC-SD is NP-hard and in NP. Therefore VPC-SD is NP-complete. □

Notice that the largest integer value assigned to the variables in the VPC-SD instance can be constant (e.g., $a = 1, a' = 2, b = 3, b' = 4, c = 5, c' = 6$), so it is trivially polynomially bounded by the instance size. Therefore we can further conclude that VPC-SD is NP-complete in the strong sense [19].

Since VPC-MD is a generalization of VPC-SD, we have:

Corollary 3.1. *VPC-MD is* NP*-complete.*

4 THE READ-CENTRIC ALGORITHM FOR VPC-MU

In this section, we present a polynomial VPC-MU algorithm, called READ-CENTRIC, with worst-case time complexity $O(n^4)$. Specifically, Sections 4.1-4.5 describe the READ-CENTRIC algorithm. Sections 4.6 and 4.7 deal with its correctness and time complexity, respectively.

Notice that in the trace of VPC-MU instance, any two write operations on the same variable must write different values to it. This assumption is key to the computational tractability of VPC-MU, as demonstrated by the READ-CENTRIC described shortly. To realize the assumption in practice, each write operation can be tagged with a globally unique identifier, e.g., by combining its process id and a local sequence number. Thus, for each read operation r, there is at most one write (denoted by $D(r)$ for dictating write) from which r reads the value. A write is said to be a *dictating* one if it has at least one *dictated* read.

4.1 The Basic Idea

The basic idea of the READ-CENTRIC algorithm is straightforward: It models the read/write trace as a directed graph $\mathcal{G}$ with operations as nodes and precedence relations between operations as directed edges. Pipelined-RAM consistency is captured by three kinds of edges corresponding to the following three rules.

- (*Rule A: program order*) For any pair of operations o_1 and o_2, if $o_1 \prec_{PO} o_2$, then add an edge from o_1 to o_2.
- (*Rule B: write-to order*) For any pair of operations w and r, if $w \prec_{WR} r$, then add an edge from w to r.
- (*Rule C: w′ wr order*) For any triple of operations w, r, and w' on the same variable, if $w = D(r) \wedge w' \prec r$, then add an edge from w' to w, leading to $w' \prec_{W'W} w \prec_{WR} r$. Notice that we denote the precedence relation between such w' and w by $\prec_{W'W}$.

Rule C can be justified by the observation that in a legal schedule, between each read operation r on variable v and its dictating write operation $w = D(r)$, there cannot be any other write (denoted by w') on the same variable v [15], [21].

To construct the graph $\mathcal{G}$ and meanwhile, to verify Pipelined-RAM consistency, the principle of the READ-CENTRIC algorithm is similar in spirit to that of the following procedure, which is referred to as the RW-CLOSURE algorithm.

- Step 1: **Apply** Rule A to add edges for program order.
- Step 2: **Apply** Rule B to add edges for write-to order. **If** some read r has no dictating write, it halts and **returns** *false*.
- Step 3: Compute the transitive closure of $\mathcal{G}$.
- Step 4: **Foreach** pair of some read r and its dictating write $w = D(r)$, identify all the w's ($w' \neq w$) such that w' performs on the same variable with that of r and there exists a path from w' to r. **Apply** Rule C to each triple (w', w, r) to add edges for w′wr order.
- Step 5: **If** any edges are added by Rule C in Step 4, **goto** Step 3.
- Step 6: It concludes that the trace satisfies Pipelined-RAM consistency if and only if the resulting graph $\mathcal{G}$ is acyclic (i.e., a DAG).

We defer the correctness proof of the RW-CLOSURE algorithm to Section 4.6.1. A running example mainly illustrating Steps 3-5 is shown in Fig. 2. For instance, after the application of Rule C to triple $Wy2, Wy1,$ and $Ry1$ (label 4), a new path from $Wf2$ to $Rf1$ arises (via edges with label 3 and label 4), leading to another application of Rule C to triple $Wf2, Wf1,$ and $Rf1$ (label 5). In this example, we can figure out a *legal schedule* of all the operations as a witness to Pipelined-RAM consistency (Fig. 3).

At its heart, the READ-CENTRIC algorithm carries out the basic idea of RW-CLOSURE, mainly involving the computations of transitive closure in Step 3, the applications of Rule C in Step 4, and the cycle detection in Step 6, in an *incremental* and *efficient* way. The key point here is that for VPC-MU, the total order between all read operations on the

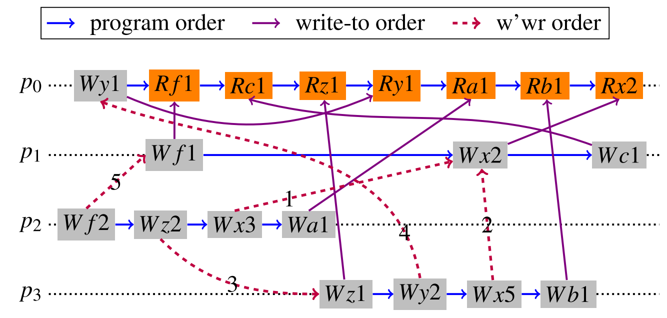


Fig. 2. Repeatedly applying Rule C to the transitive closure of the operation graph according to the RW-Closure algorithm. *(The edges added by Rule C are denoted by dashed lines, with labels indicating the order in which they are added.)*

same variable (on process p_0) leads to a total order between their *dictating writes*. The Read-Centric algorithm makes use of the total order between dictating writes to avoid redundant applications of Rule C in RW-Closure. Moreover, the Read-Centric algorithm is *incremental* in the way that it processes all the read operations on process p_0 one by one (hence it is named "Read-Centric"). For each read operation, it applies Rule C locally and in a reverse topological order of the subgraph induced by this read. The "incremental" feature makes the Read-Centric algorithm practically more efficient because it can terminate early once a certification for inconsistency is found, without having to collect all the operations.

4.2 Overview of the Read-Centric Algorithm

As mentioned above, the Read-Centric algorithm is incremental with respect to the read operations in program order (of p_0). To facilitate this idea, we first introduce two notations.

Intuitively, r-downset consists of all the operations which must be scheduled before r, plus r itself.

Definition 4.1 (r-downset ($r_{\Downarrow}$)). *r-downset of a read operation r is a set $r_{\Downarrow}$ of operations such that*

- $r \in r_{\Downarrow}$;
- $o \in r_{\Downarrow} \wedge o' \prec o \Rightarrow o' \in r_{\Downarrow}$.

Let r be a read operation and r' the previous read of r. We use r-delta to refer to the "extra" operations which are also scheduled before r, besides those in r'-downset. In other words,

Definition 4.2 (r-delta (r_{δ})). *r-delta of a read operation r is a set (of operations) which equals the relative complement of $r'_{\Downarrow}$ with respect to $r_{\Downarrow}$ (i.e., $r_{\Downarrow} \setminus r'_{\Downarrow}$). Here r' is the previous read of r.*

For the first read operation r on process p_0, we define $r_{\delta} = r_{\Downarrow}$.

In terms of both r-downset and r-delta, we now sketch how to carry out the basic idea of the RW-Closure algorithm

$$Wf2\ Wf1\ Wz2\ Wz1\ Wy2\ Wy1\ \mathbf{Rf1};$$
$$Wx5\ Wx3\ Wx2\ Wc1\ \mathbf{Rc1}; \mathbf{Rz1}; \mathbf{Ry1};$$
$$Wa1\ \mathbf{Ra1}; Wb1\ \mathbf{Rb1}; \mathbf{Rx2}.$$

Fig. 3. A legal schedule for the trace in Fig. 2. *(Read operations are shown in bold and separated by semicolons.)*

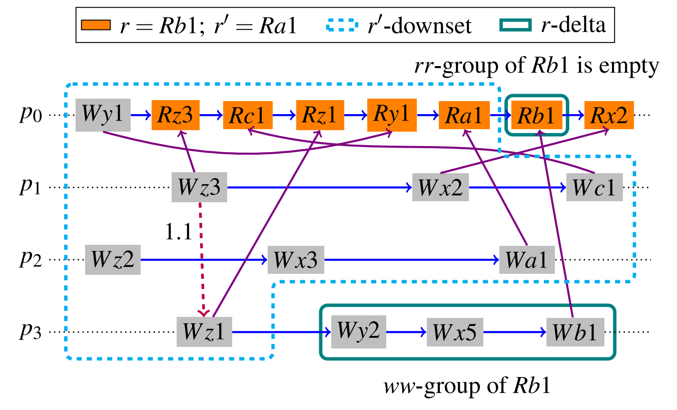


Fig. 4. Demonstrations of the definitions of r-downset and r-delta, case 1) in the Read-Centric algorithm, and the procedure Init-Reachability.

(Section 4.1) in an incremental and efficient way (see Algorithm 1 and Fig. 4). Let r be the current read operation under scrutiny, r' the previous read of r, and v the variable of r. After initializing the reachability relation concerning the incrementally new operations in $r_{\delta} = r_{\Downarrow} \setminus r'_{\Downarrow}$ (Line 9), the Read-Centric algorithm attempts to *schedule locally* on the $r_{\Downarrow}$-induced subgraph. Notice that both $r_{\Downarrow}$ and $r'_{\Downarrow}$ are obtained according to Definition 4.1 with respect to the dynamic graph $\mathcal{G}$*till that time.*

Algorithm 1. The Read-Centric algorithm (sketch).

```
1:  apply Rule A to add edges for program order
2:  apply Rule B to add edges for write-to order
3:  if $\exists r(r$ has no $D(r) \vee r \prec_{PO} D(r))$ then
4:      return false
5:  foreach read $r$ in program order of $p_0$ do
6:      Let $r'$ be the previous read of $r$
7:      $v \leftarrow var(r); w \leftarrow D(r)$
8:      //explained in Section 4.4.1
9:      Init-Reachability($r', r$)
10:     //schedule write operations on variable $v$
11:     foreach $w'$ in $r$-downset on variable $v$ do
12:         Apply-Rule-C($(w', w, r), r$)
13:     //schedule other write operations
14:     if $w \notin r'_{\Downarrow}$ then continue                //case 1)
15:     $cycle \leftarrow$ Topo-Schedule ($r_{\Downarrow}$)            //case 2)
16:     if $cycle$ then return false
17: return true
```

Specifically, the *schedule* procedure starts with a simple observation that r must read from its dictating write operation $D(r)$ (Lines 10 - 12). According to Rule C, any write operation w' in r-downset on the variable v other than $D(r)$ must be scheduled before $D(r)$. Thus the edges like $w' \rightarrow D(r)$ are added, updating the reachability relation between operations. Consequently, more applications of Rule C may be triggered. There are two cases to consider: *1) $D(r) \notin r'_{\Downarrow}$ and 2) $D(r) \in r'_{\Downarrow}$*. In the former case (Line 14; also as shown in Fig. 4), the new added edges like $w' \rightarrow D(r)$ have no effect on the reachability relation between the operations from $r'_{\Downarrow}$. In the latter one (Line 15; discussed in Section 4.4), the operations in $r_{\Downarrow}$ should be locally scheduled. This involves a serial of applications of Rule C. Different from those in the RW-Closure algorithm, the applications of Rule C here are carried out in a *reverse topological order* of the $r_{\Downarrow}$-induced subgraph (see procedure TOPO-SCHEDULE called in Line 15). Once some cycle is created, the algorithm aborts and outputs "false". Otherwise, it eventually terminates and outputs "true".

4.3 Data Structures For Reachability Relation

During the course of TOPO-SCHEDULE (Line 15), the r-downset induced subgraph is dynamic in the way that edges are added on demand due to Rule C. To capture the dynamic reachability relation, two data structures need to be dynamically maintained.

First, *ReachableRead* maintains, for each write operation, the *first* read it can reach via the present precedence relation (i.e., $\prec$). Recall that all read operations are program ordered on the process p_0. Formally,

Definition 4.3 (ReachableRead (RR)). ReachableRead *is a dictionary composed of a collection of* $(w, r) \in W \times R$ *pairs such that*

$$RR[w] = r \Leftrightarrow w \prec r \wedge \nexists_{r' \prec_{PO} r}(w \prec r').$$

Roughly speaking, *PrecedingWrite* of an operation compactly summarizes the *dictating* write operations *preceding* it, one per variable. More precisely,

Definition 4.4 (PrecedingWrite (PW)). PrecedingWrite *of an operation* o *is a dictionary composed of a collection of* $(v, w) \in V \times W$ *pairs such that* $PW[v] = w$ *if and only if*

1) $w \prec o \wedge var(w) = v$;
2) $\exists_{r \in R} \, (w \prec_{WR} r)$;
3) *for any other* w' *satisfying 1) - 2), we have* $w' \prec w$.

Condition 1) distinguishes the preceding write operations for each variable v. Condition 2) only concerns the *dictating* write operations. Condition 3) requires that

Proposition 4.1. *All* dictating *write operations on the same variable are totally ordered.*

Proof. This is true due to Rule C and the fact that all read operations are totally program ordered on process p_0. Furthermore, the total order between the dictating write operations (on the same variable) is the same with that between their dictated read operations (taking the first read if there are many). □

Being complementary, the two data structures *ReachableRead* and *PrecedingWrite* together capture the dynamic reachability relation and facilitate the procedure IDENTIFY-APPLY-RULE-C as described in Section 4.4.2 (i.e., its subprocedures IDENTIFY-RULE-C and CYCLE-DETECTION).

4.4 Detailed Design

In this section, we first describe INIT-REACHABILITY (called in Line 9 of Algorithm 1), a preparation for later scheduling. Then we describe procedures related to applying Rule C, including IDENTIFY-RULE-C, APPLY-RULE-C, and CYCLE-DETECTION. Finally, we describe the key procedure TOPO-SCHEDULE (called in Line 15 of Algorithm 1), which involves a serial of applications of Rule C and returns true once some cycle is created. We will elaborate on how to apply Rule C locally and in a reverse topological order of some subgraph.

4.4.1 Procedure INIT-REACHABILITY

Upon the read r and its previous read r', the procedure INIT-REACHABILITY initializes the reachability relation, in terms of both *ReachableRead* and *PrecedingWrite*, concerning the operations in $r_\delta = r_\Downarrow \setminus r'_\Downarrow$. (Here both $r_\Downarrow$ and $r'_\Downarrow$ are obtained according to Definition 4.1 with respect to the dynamic graph $\mathcal{G}$ *till the time* when INIT-REACHABILITY is called.)

Specifically, the *first* reachable read (i.e., *ReachableRead*) of each write in r_δ is initialized to be r. To initialize the *PrecedingWrite* for each operation in r-delta, we partition them (except r) into two groups (both could be empty): *1*) the rr-group consists of all the write operations between r' and r on process p_0 (both exclusive); *and 2*) the ww-group consists of the rest on the process of $D(r)$, the dictating write of r. In both groups, each operation updates its *PrecedingWrite* based on the *PrecedingWrite* of its previous operation in respective program order. Finally, the read r updates its *PrecedingWrite* twice, one time per group.

Fig. 4 shows an example where the rr-group of $r = Rb1$ is empty. In the ww-group, for example, $Wy2$ updates its *PrecedingWrite* based on $Wz1$: for $Wz1$, we have $PW[z] = Wz3$; for $Wy2$, we get $PW[z] = Wz1$ because $Wz1$ itself is a dictating write. The read $r = Rb1$ updates its *PrecedingWrite* based on both $Wb1$ and $Ra1$:

$$PW[a] = Wa1, PW[b] = Wb1, PW[c] = Wc1, PW[x] = Wx2, PW[y] = Wy1, PW[z] = Wz1.$$

4.4.2 Procedures Involving Applying Rule C

Due to *ReachableRead* and *PrecedingWrite*, the procedure of applying Rule C is efficient despite the dynamically changing reachability relation (Algorithm 2). In the following, we refer to the three operations involved in Rule C as "the $w', w,$ and r parts of Rule C" or simply as "the $w'wr$ triple".

Algorithm 2. Procedures involving applying Rule C.

```
1: //identify the $w'wr$ triple (or NIL) of Rule C
2: procedure IDENTIFY-RULE-C($w'$)
3:     $r_{old} \leftarrow$ the last value of $RR[w']$
4:     $r_{new} \leftarrow RR[w']$
5:     $R[r_{new} \dots r_{old}) \triangleq \{r \in R \mid r_{new} \preceq_{PO} r \prec_{PO} r_{old}\}$
6:     $r \leftarrow$ the first read in $R[r_{new} \dots r_{old})$ on $var(w')$
7:     if $r \neq NIL$ then return $(w', w = D(r), r)$
8:     return NIL

1: //return true if some cycle is created
2: //$r_{cur}$: the current read under scrutiny
3: procedure APPLY-RULE-C($(w', w, r), r_{cur}$)
4:     add edge $w' \rightarrow w$
5:     if CYCLE-DETECTION($w', w$) then return true
6:     //update reachability within $r_{cur}$-downset
7:     //explained in Section 4.4.2
8:     UPDATE-REACHABILITY ($w', w, r_{cur}$)
9:     return false

1: //return true if a cycle with $w' \rightarrow w$ is created
2: procedure CYCLE-DETECTION($w', w$)
3:     //$PW[var(w')]$ of $w' \rightsquigarrow w' \rightarrow w$ already exists
4:     if $w \preceq PW[var(w')]$ of $w'$ then return true
5:     return false
```

First, to identify the $w'wr$ triple of Rule C (procedure IDENTIFY-RULE-C*):* For some w', it is sufficient to check whether new paths like from w' to r arise. The data structure *ReachableRead* (Definition 4.3) serves the purpose. For w' (on variable v) in check, suppose that its first reachable read $RR[w']$ has been changed from r_{old} to r_{new}. It means that w' can now reach the read operations in $R[r_{new} \ldots r_{old})$ which denotes the set of read operations between r_{new} and r_{old} on process p_0 (formally, $R[r_{new} \ldots r_{old}) \triangleq \{r \in R \mid r_{new} \preceq_{PO} r \prec_{PO} r_{old}\}$) (Lines 3-5). For each read r on variable v in $R[r_{new} \ldots r_{old})$, a triple of $w', w = D(r), r$ is identified. If there are more than one such r, we only takes the *first* one (in program order) and its corresponding triple (Line 6). This choice is justified in Lemma 4.2.

Second, cycle detection (procedure CYCLE-DETECTION*):* After identifying a $w'wr$ triple (on variable v) of Rule C and adding the edge $w' \rightarrow w$, procedure CYCLE-DETECTION is called to check whether some cycle involving $w' \rightarrow w$ is created. To complete a cycle with $w' \rightarrow w$, a path from w to w' (denoted by $w \rightsquigarrow w'$) is needed. The data structure *PrecedingWrite* (Definition 4.4) serves the purpose. $PW[v]$ of w' maintains the *last* dictating write on variable v which precedes w'. Thus cycle detection amounts to figuring out whether or not w precedes (or is) $PW[v]$ of w' (Line 4). This is easy because w concerned here is also a dictating write on v, and is thus totally ordered with $PW[v]$ of w' (Proposition 4.1).

Third, to update the reachability relation (procedure UPDATE-REACHABILITY*):* If no cycle is created, APPLY-RULE-C continues to update the reachability relation, namely *ReachableRead* of w' and *PrecedingWrite* of w and its successors. First, the *ReachableRead* of w' is updated to the read $RR[w]$ if $RR[w] \prec_{PO} RR[w']$. The update of *ReachableRead* of the predecessors of w' has been integrated in procedure TOPO-SCHEDULE. Second, the *PrecedingWrite* of w and its successors are updated (in a topological sorting order) to take into account the *PrecedingWrite* of w'.

4.4.3 Procedure TOPO-SCHEDULE

Procedure TOPO-SCHEDULE mainly involves a serial of applications of Rule C and returns true once some cycle is created. The key is that the applications of Rule C are carried out locally and in a reverse topological order of some subgraph.

First, the operations which may act as the w' parts of Rule C are all *locally* in $D(r)$-downset. Second, they are carried out in a *reverse topological order* of the $D(r)_{\Downarrow}$-induced subgraph. The former claim follows from a simple argument: *a*) whether to apply Rule C is determined by *ReachableRead* of its w' part (procedure IDENTIFY-RULE-C); *b*) *ReachableRead* of w' is updated only due to its successors; *and c*) the procedure TOPO-SCHEDULE is called immediately after some Rule C edges to $D(r)$ are added (Lines 10 - 12 of Algorithm 1).

In the following, we show how to organize the applications of Rule C (Algorithm 3). The basic idea is to integrate the applications of Rule C with a (reverse) topological sorting algorithm. In such a reverse topological sorting algorithm, a queue is used to maintain the *sink operations* that have no successors (Lines 11 - 13). Each time we pick up (and remove) one of the sink operations (denoted by w'), update its *ReachableRead* based on its direct successors, and apply Rule C if necessary (Lines 14 - 24). After w' has been scheduled, it is marked DONE and the dependencies on it are erased. The new sink operations are put into the queue (Lines 30 - 36). However, the applications of Rule C can introduce new edges into the $D(r)_{\Downarrow}$-induced subgraph $\mathcal{G}_{D(r)_{\Downarrow}}$. Suppose that an edge from w' to w is added. In is particularly subtle if $w \in D(r)_{\Downarrow}$ (meaning that it is possible for w to act as the w' part of Rule C) and w has not been marked DONE yet. In this case, it is necessary to process w first *before* marking w' DONE. This is implemented by imposing dependency of w' on w (Lines 25 - 29).

Algorithm 3. Procedure TOPO-SCHEDULE

```
1:  //return true once some cycle is created
2:  //$r$: the current read under scrutiny
3:  procedure TOPO-SCHEDULE($r_{\Downarrow}$)
4:    //data structures for reverse topological sorting
5:    $\mathcal{G}_{D(r)_{\Downarrow}} \leftarrow D(r)_{\Downarrow}$- induced subgraph
6:    traverse $\mathcal{G}_{D(r)_{\Downarrow}}$ to compute for each $o \in D(r)_{\Downarrow}$:
7:    (a) COUNT: number of direct successors
8:    (b) SUCLIST: list of direct successors
9:    (c) PRELIST: list of direct predecessors
10:   (d) DONE: initially false indicator whether it
      has been scheduled
11:   //queue to maintain "sink" operations
12:   QZERO ← empty queue
13:   enqueue(QZERO, $D(r)$) //start from $D(r)$
14:   //in a reverse topological order of $\mathcal{G}_{D(r)_{\Downarrow}}$
15:   while QZERO is not empty do
16:     $w'$ ← dequeue(QZERO) //the "$w'$ part"
17:     //apply Rule C if necessary
18:     if $w' \in W \wedge w'$.DONE = false then
19:       $\boldsymbol{RR}[w'] \leftarrow \min_{o \in w'.\mathrm{SUCLIST}}(\boldsymbol{RR}[w'], \boldsymbol{RR}[o])$
20:       //we don't care about the "$r$ part"
21:       $(w', w, *)$ ← IDENTIFY-RULE-C($w'$)
22:       if $(w', w, *) \neq NIL$ then
23:         $cycle$ ← APPLY-RULE-C($(w', w, *), r$)
24:         if $cycle$ then return true
25:         //dep. on non-scheduled write $w$
26:         if $w \in D(r)_{\Downarrow} \wedge (w$.DONE = false$)$ then
27:           insert $w'$ into $w$.PRELIST
28:           insert $w$ into $w'$.SUCLIST
29:           $w'$.COUNT ← $w'$.COUNT + 1
30:     //$w'$ has been scheduled successfully
31:     if $w'$.COUNT = 0 then
32:       $w'$.DONE ← true
33:       foreach $o \in w'$.PRELIST do
34:         $o$.COUNT ← $o$.COUNT - 1
35:         if $o$.COUNT = 0 then
36:           enqueue(QZERO,$o$)
37:   return false
```

In Lemma 4.3 (Section 4.7), we prove that each write operation in the $D(r)_{\Downarrow}$-induced subgraph plays the "w' role" of Rule C *at most once*, which justifies the efficiency of the procedure TOPO-SCHEDULE.

4.5 An Illustrating Example

We show a running example of the READ-CENTRIC algorithm in Fig. 5. It mainly concerning its sketch and the key procedure TOPO-SCHEDULE. Assume that $Rx2$ is now under

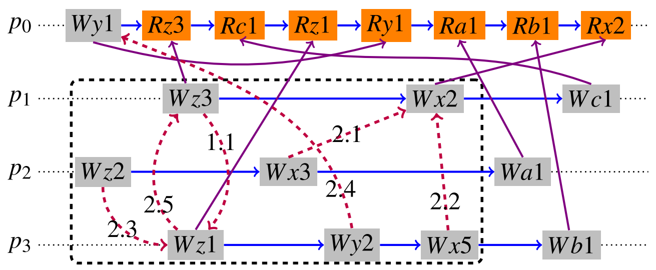


Fig. 5. Illustration of the READ-CENTRIC algorithm.

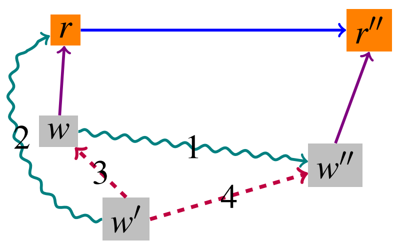


Fig. 6. Procedure IDENTIFY-RULE-C only considers the *first* read operation r for Rule C.

scrutiny (i.e., $r = Rx2$ in Line 5 of Algorithm 1). Notice that the edge $Wz3 \rightarrow Wz1$ (label 1.1) has already been added due to $Rz1$. The *schedule* procedure starts with adding edges $Wx3 \rightarrow Wx2$ (label 2.1) and $Wx5 \rightarrow Wx2$ (label 2.2) (Lines 10-12). It then calls the procedure TOPO-SCHEDULE in the case of $Wx2 \in Rb1_{\Downarrow}$ (Line 15).

In procedure TOPO-SCHEDULE (Algorithm 3), the operations which may act as the w' parts of Rule C are in $Wx2$-downset (in a rectangle dotted box). Suppose in the course of reverse topological sorting, $Wz2$ is processed *before* $Wy2$ and $Wz1$. By Rule C, an edge $Wz2 \rightarrow Wz1$ (label 2.3) is added. Since $Wz1$ is not DONE, we have to process $Wz1$ first before marking $Wz2$ DONE (Lines 25-29). According to the reverse topological order, $Wy2$ is processed and an edge $Wy2 \rightarrow Wy1$ (label 2.4) is added. Then it is $Wz1$'s turn. Since there is a path $Wz1 \rightsquigarrow Rz3$ via the edge $Wy2 \rightarrow Wy1$, Rule C is applied and an edge $Wz1 \rightarrow Wz3$ (label 2.5) is added. A cycle involving $Wz1$ and $Wz3$ is thus created.

4.6 Correctness Proof

In this section, we first prove the correctness of the RW-CLOSURE algorithm in Section 4.1 and then establish the correctness of the READ-CENTRIC algorithm by showing that it is *equivalent* to RW-CLOSURE in the sense that their resulting graphs $\mathcal{G}$ have the same reachability relation.

4.6.1 Correctness Proof of the RW-CLOSURE Algorithm

The correctness of the RW-CLOSURE algorithm is stated in the following theorem.

Theorem 4.1. *The VPC-MU instance satisfies Pipelined-RAM consistency if and only if the resulting graph $\mathcal{G}$ of the RW-CLOSURE algorithm is acyclic.*

Proof. ($\Rightarrow$) *By contradiction*. If the resulting graph $\mathcal{G}$ is not a DAG, there exists some operation scheduled before itself.

($\Leftarrow$) *If the resulting graph $\mathcal{G}$ is acyclic*, we expect to construct some *legal schedule* (denoted by $\pi_{\mathcal{G}}$) as a witness to Pipelined-RAM consistency. We state the construction of $\pi_{\mathcal{G}}$ and the proof that $\pi_{\mathcal{G}}$ is legal in Definition 4.5 and Lemma 4.1, respectively. □

To construct $\pi_{\mathcal{G}}$, a *specific topological sorting* on $\mathcal{G}$ is performed.

Definition 4.5 (DAG-schedule ($\pi_{\mathcal{G}}$)). *Given the resulting DAG $\mathcal{G}$ of the RW-CLOSURE algorithm, the legal schedule $\pi_{\mathcal{G}}$ (initially, it is an empty sequence) is constructed as follows:*

- *Repeatedly take each read operation r on process p_0 in program order, perform any topological sorting on r_{δ}-induced subgraph, and append it to $\pi_{\mathcal{G}}$.*

The example in Section 4.1 gives an illustration of such schedule (Fig. 3).

Lemma 4.1. *If the resulting graph $\mathcal{G}$ of the RW-CLOSURE algorithm is acyclic, the schedule $\pi_{\mathcal{G}}$ constructed in Definition 4.5 is legal.*

Proof. Please refer to Section 1 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2015.2453985, for the proof. □

4.6.2 Correctness Proof of the READ-CENTRIC Algorithm

As mentioned before, we establish the correctness of the READ-CENTRIC algorithm by showing that it is *equivalent* to RW-CLOSURE in the sense that their resulting graphs have the same reachability relation.

Because the edges for both program order and write-to order are static, they are the same for two algorithms. The set of $w'wr$ triples identified in READ-CENTRIC is a *subset* of that identified in RW-CLOSURE. The only possible *missing* of $w'wr$ triples is due to procedure IDENTIFY-RULE-C.

Lemma 4.2. *In procedure IDENTIFY-RULE-C, for w', only the* first *r in $R[r_{new} \ldots r_{old})$ is considered for Rule C. This choice does not reduce any reachability relation of the resulting graph of the RW-CLOSURE algorithm.*

Proof. It is sufficient to show that each missing edge for w'wr order is implied by other existing edges. This is illustrated in Fig. 6 in which all operations perform on the same variable and $w = D(r), w'' = D(r'')$. For w' there exists a path $w' \rightsquigarrow r$ (label 2). By Rule C, both the edge $w' \rightarrow w$ (label 3) and the edge $w' \rightarrow w''$ (label 4) should be added. However, the latter one is implied by: *1*) a path $w \rightsquigarrow w''$ (label 1) whose existence is guaranteed by $r \prec_{PO} r''$ and Proposition 4.1; *and 2*) the edge $w' \rightarrow w$ (label 3). □

Hence, the correctness of the READ-CENTRIC algorithm directly follows from that of the RW-CLOSURE algorithm.

Theorem 4.2. *The VPC-MU instance satisfies Pipelined-RAM consistency if and only if the READ-CENTRIC algorithm terminates with a DAG.*

4.7 Time and Space Complexity

The worst-case time complexity of the READ-CENTRIC algorithm is dominated by the cost of the procedure TOPO-SCHEDULE. The efficiency of the latter is justified by the following lemma.

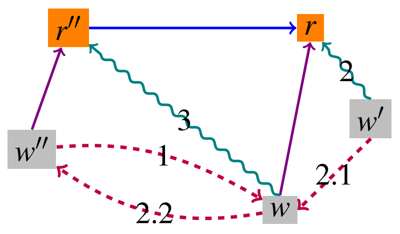


Fig. 7. Procedure Topo-Schedule applies Rule C at most once for w'.

Lemma 4.3. *Let r be the read operation under scrutiny. For each $w' \in D(r)_{\Downarrow}$, procedure* Topo-Schedule *applies Rule C* at most once *with it as the "w' part".*

Proof. In procedure Topo-Schedule, the only case in which w' will be *checked* for Rule C more than once is that an edge $w' \to w$ is added, w is in $D(r)_{\Downarrow}$, and w has not been marked DONE yet (Lines 25-29 in Algorithm 3). In this case, we show that Rule C is *not applicable* when w' is checked again. This is illustrated in Fig. 7 in which all operations perform on the same variable v and $w = D(r), w'' = D(r'')$. The *first* application of Rule C to triple $w', w,$ and r has introduced the edge $w' \to w$ (label 2.1). Assume, *by contradiction*, that Rule C is applicable when w' is checked again. It requires that via w a new read operation r'' on variable v with $r'' \prec_{PO} r$ be now reachable. Back to the time when w was checked, r'' was then reachable from w (label 3). An edge $w \to w''$ (label 2.2) was added, closing a cycle with the edge $w'' \to w$ (label 1) whose existence is guaranteed by $r'' \prec_{PO} r$ and Proposition 4.1. The procedure Topo-Schedule would then abort. □

The following theorem gives the overall worst-case time complexity of the Read-Centric algorithm.

Theorem 4.3. *The worst-case time complexity of the* Read-Centric *algorithm is $O(n^4)$.*

Proof. Suppose that read operation r is under scrutiny. There are at most n operations in $r_{\Downarrow}$ and $m = O(n^2)$ edges between them. The time complexity of procedure Topo-Schedule comprises 1) $O(n+m)$ for reverse topological sorting; 2) $O(n \cdot c_{apply})$ for at most n applications of Rule C (due to Lemma 4.3), each of which costs:

$$c_{apply} = \underbrace{O(n)}_{\text{Identify-Rule-C}} + \underbrace{O(1)}_{\text{Cycle-Detection}} + \underbrace{O(1+n+m+n\cdot n)}_{\text{Update-Reachability}} = O(n^2).$$

Thus procedure Topo-Schedule costs $O(n^3)$ in the worst case. Then the worst-case time complexity of the Read-Centric algorithm is $O(n^4)$:

$$\underbrace{O(n)}_{\text{iterations}} \cdot (\underbrace{O(n^2)}_{\text{Init-Reachability}} + \underbrace{O(n^3)}_{\text{Topo-Schedule}}) = O(n^4).$$

□

The space complexity of the Read-Centric algorithm is $O(n^2)$:

$$\underbrace{O(n)}_{ReachableRead} + \underbrace{O(n^2)}_{PrecedingWrite} = O(n^2).$$

5 Performance Evaluation

The worst-case time complexity of the Read-Centric algorithm for the VPC-MU problem given in Section 4.7 is asymptotic and overly pessimistic. In this section, we evaluate their practical efficiency and scalability by experiments. For comparison, we have also implemented the RW-Closure algorithm in Section 4.1. However, we have not compared our algorithms with other existing approaches in these experiments, because we are not aware of any directly related work for the VPC-MU problem. For more discussions, please refer to the Related Work Section (Section 6).

5.1 Experimental Design

In our experiments, we use two types of read/write traces. In our first experiment, we evaluate both algorithms on randomly generated traces. (Each read operation r and its dictating write $w = D(r)$ are generated in pair.) Without sufficient priori knowledge, we adopt the random traces to cover most of the cases our algorithms need to handle in verifying Pipelined-RAM consistency. In the second experiment, we use Pipelined-RAM consistent traces. Notice that for Pipelined-RAM consistent traces, both the RW-Closure and the Read-Centric algorithms will always process all operations in the traces. In comparison, on randomly generated traces, they may terminate early once a certificate for inconsistency is found. Thus, Pipelined-RAM consistent traces actually serve as the worst case inputs for both algorithms.

For each experiment, both the number of processes and the number of operations are tunable, in order to investigate their impacts on the time cost (in *ms*). The number of processes varies from 2 to 20, and the number of operations varies from 5, 000 to 30, 000. All the experiments are performed on an Intel Core i7 3.40 GHz machine with 4 GB RAM.

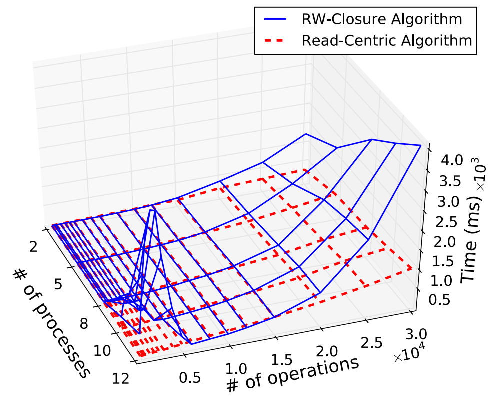
5.2 Experimental Results

As shown in Fig. 8, the Read-Centric algorithm outperforms the RW-Closure algorithm in time cost over both random traces and Pipelined-RAM consistent traces. The more the operations are, the more significant the advantage becomes. On the other hand, when both running over the Pipelined-RAM consistent traces, the Read-Centric algorithm performs much more efficiently than the RW-Closure algorithm. For instance, the Read-Centric algorithm gains at most 694X speedup over the Pipelined-RAM consistent trace with 20 processes and 8, 000 operations.

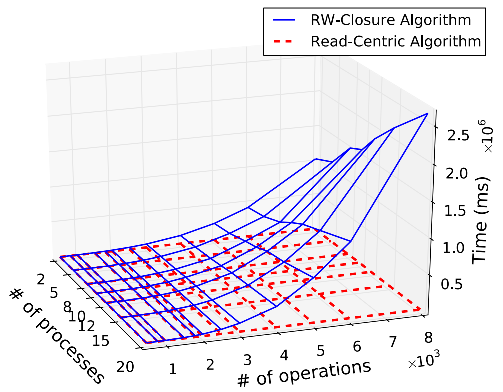
To further evaluate the scalability of the Read-Centric algorithm, we increase the number of operations in the Pipelined-RAM consistent traces to different scales. As shown in Fig. 9, the Read-Centric algorithm can efficiently deal with the Pipelined-RAM consistent traces with 20 processes and up to 60,000 operations (in less than 600 s). In contrast, it takes the RW-Closure algorithm more than 3,000 s over the Pipelined-RAM consistent trace with 20 processes and 8,000 operations.

6 Related Work

Many efforts have been made on the verification problems with respect to other consistency models than Pipelined-

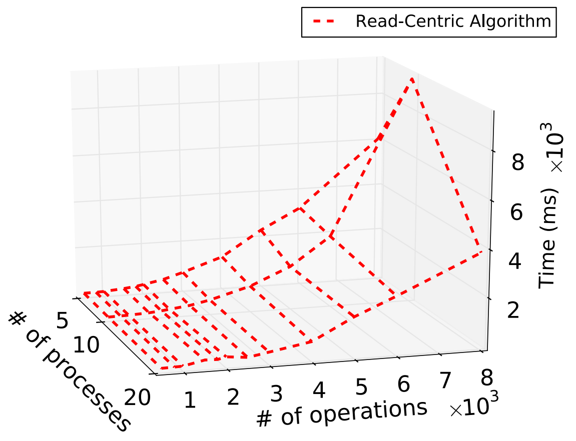


(a) Time cost of the RW-CLOSURE algorithm and the READ-CENTRIC algorithm over random traces.

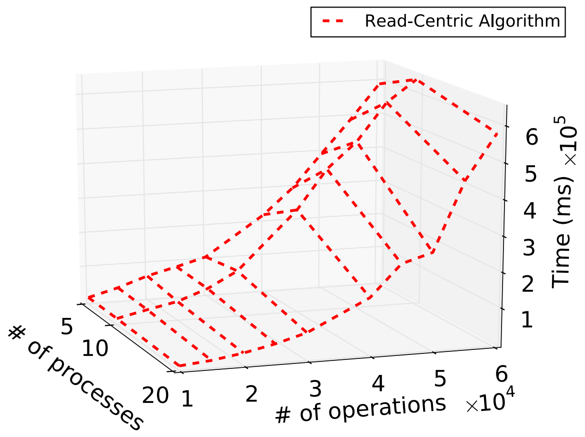


(b) Time cost of the RW-CLOSURE algorithm and the READ-CENTRIC algorithm over Pipelined-RAM consistent traces.

Fig. 8. Time cost of the READ-CENTRIC algorithm over both random traces and Pipelined-RAM consistent traces.



(a) Time cost of the READ-CENTRIC algorithm over Pipelined-RAM consistent traces with less than 10, 000 operations.



(b) Time cost of the READ-CENTRIC algorithm over Pipelined-RAM consistent traces with more than 10, 000 operations.

Fig. 9. Scalability of the READ-CENTRIC algorithm in terms of time cost over Pipelined-RAM consistent traces.

RAM. In their seminal work, Gibbons and Korach [20] study the verifying sequential consistency (VSC) and the verifying linearizability (VL) problems. Both problems are proved to be NP-complete in general. In addition, they define the VSC-read problem, in which a *read-mapping* is known, and prove that it remains NP-complete. Here a read-mapping is a function mapping each read operation to some write operation which is responsible for the value read. Cantin et al. [22] show that the verifying memory coherence problem (VMC) is NP-complete. They also prove that the problem of verifying sequential consistency for executions that are memory coherent (VSCC) remains NP-complete. Golab et al. [18] study the verification problems with respect to safety, regularity, atomicity, and sequential consistency. Beyond a yes/no answer, they seek online algorithms to detect a consistency violation as soon as it appears. They also consider how to quantify the severity of violations. More recently, Golab et al. [23] solve the verification problem of 2-atomicity (2-AV) and show that the weighted k-AV problem is NP-complete. In this work we investigate the *verifying Pipelined-RAM consistency* (VPC) problem. As far as we know, we are the first to systematically study this problem.

In the context of shared memory multiprocessor, some relaxed memory consistency models have been studied [21], [24], [25]. Specifically, Hangal et al. [24] develop TSOtool to verify the traces of programs against Total Store Order (TSO, for short) model when a read-mapping is known (VTSO-read). The time complexity of their algorithm is $O(n^5)$, where n is the number of operations in the trace. Roy et al. [21] also deal with the VTSO-read problem and present a fully parallelized algorithm with $O(n^4)$ time complexity. Baswana et al. [25] identify a graph problem called implied-set-closure as the abstraction of the bottleneck of the VTSO-read problem, and further reduce its time complexity to $O(n^3)$.

We have addressed a fundamentally different problem from that in the TSOtool paper [24]. We propose the READ-CENTRIC algorithm to verify the traces of data replicas against Pipelined-RAM consistency model, rather than against TSO model. Since these two consistency models have significant differences, their verification algorithms also have important differences (besides the resemblance). More precisely, both the TSO algorithm and our READ-CENTRIC algorithm follow a more general principle: most existing consistency models can be decomposed into a collection of fundamental orders [15]. For the VPC problem, all read operations are totally ordered on the same process p_0. The key point is that in the VPC-MU variant, the total order between all read operations on the same variable leads to the total order between their dictating writes. The READ-CENTRIC algorithm for VPC-MU makes use of the total order between dictating writes to avoid redundant applications of the w'wr order (represented by rule R6 in the TSOtool paper), and to completely avoid the use of another order (represented by rule R7 in the TSOtool paper).

Another major difference between our READ-CENTRIC algorithm and the TSO algorithm is that the TSO algorithm only conducts *approximate* checking because it is "solving" an NP-complete problem with a polynomial time algorithm. In contrast, we show that the VPC-MU problem (in which a read-mapping is known) can be *completely* solved in polynomial time. The correctness proof of the READ-CENTRIC algorithm is one of our key contributions. On the other hand, we prove that the VPC-SU problem is NP-complete (so is VPC-MU).

7 CONCLUSION

In this work, we have studied the problem of verifying Pipelined-RAM consistency over read/write traces (VPC, for short). Specifically, we proved that both the VPC-SD variant and the VPC-MD variant are NP-complete. We also presented a polynomial algorithm, called READ-CENTRIC, for the VPC-MU variant. The experiments have demonstrated its practical efficiency and scalability.

The verification problems with respect to other weak consistency models, e.g., causal consistency [10], are also worth investigation. Because Pipelined-RAM is a weakening of causal consistency, our NP-complete result also applies to the general problem of verifying causal consistency. However, it remains open to solve its restricted variant when writes can only assign unique values to each shared variable. Moreover, it would be interesting to further study the complexity issues of evaluating the severity of consistency violations [18], [23].

ACKNOWLEDGMENTS

This work is supported by the National 973 Program of China (2015CB352202) and the National Science Foundation of China (61272047, 91318301, 61321491). Yu Huang is the corresponding author.

REFERENCES

[1] 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 *Proc. 21st ACM SIGOPS Symp. Operating Syst. Principles*, Oct. 2007, pp. 205–220.

[2] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. 5th Biennial Conf. Innovative Data Syst. Res.*, Jan. 2011, pp. 223–234.

[4] E. Brewer, "Towards robust distributed systems," in *Proc. Annu. ACM SIGACT-SIGOPS Symp. Principles Distrib. Comput.*, Jul. 2000, p. 7.

[5] S. Gilbert and N. A. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[6] D. Abadi, "Consistency tradeoffs in modern distributed database system design," *IEEE Comput.*, vol. 45, no. 2, pp. 37–42, Feb. 2012.

[7] R. Lipton and J. Sandberg, "PRAM: A scalable shared memory," Princeton Univ., Princeton, NJ, USA, Tech. Rep. CS-TR-180-88, Sep. 1988.

[8] J. R. Goodman, "Cache consistency and sequential consistency," IEEE Scalable Coherent Interface (SCI) Working Group, University of Wisconsin-Madison, USA, Tech. Rep. 61, Mar. 1989.

[9] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989.

[10] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: Definitions, implementation, and programming," *Distrib. Comput.*, vol. 9, no. 1, pp. 37–49, Mar. 1995.

[11] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[12] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[13] L. Lamport, "On interprocess communication," *Distrib. Comput.*, vol. 1, no. 2, pp. 77–101, Jun. 1986.

[14] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[15] R. Steinke and G. Nutt, "A unified theory of shared memory consistency," *J. ACM*, vol. 51, no. 5, pp. 800–849, Sep. 2004.

[16] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session guarantees for weakly consistent replicated data," in *Proc 3rd Int. Conf. Parallel Distrib. Inform. Syst.*, 1994, pp. 140–149.

[17] D. Terry, "Replicated data consistency explained through baseball," *Commun. ACM*, vol. 56, no. 12, pp. 82–89, Dec. 2013.

[18] W. Golab, X. Li, and M. Shah, "Analyzing consistency properties for fun and profit," in *Proc. 30th Annu. ACM SIGACT-SIGOPS Symp. Principles Distrib. Comput.*, Jun. 2011, pp. 197–206.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: Freeman, 1979.

[20] P. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, no. 4, pp. 1208–1244, Aug. 1997.

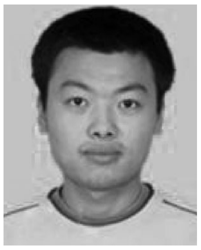
[21] A. Roy, S. Zeisset, C. Fleckenstein, and J. Huang, "Fast and generalized polynomial time memory consistency verification," in *Proc. Comput. Aided Verif.*, 2006, pp. 503–516.

[22] J. Cantin, M. Lipasti, and J. Smith, "The complexity of verifying memory coherence and consistency," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 7, pp. 663–671, Jul. 2005.

[23] W. Golab, J. Hurwitz, and X. Li, "On the k-atomicity-verification problem," in *Proc. 33rd Int. Conf. Distrib. Comput. Syst.*, Jul. 2013, pp. 591–600.

[24] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan, "TSOtool: A program for verifying memory systems using the memory consistency model," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, Jun. 2004, pp. 114.

[25] S. Baswana, S. Mehta, and V. Powar, "Implied set closure and its application to memory consistency verification," in *Proc. Comput. Aided Verif.*, 2008, pp. 94–106.



Hengfeng Wei received the BSc degree in computer science from Nanjing University, China, in 2009, where he is now working toward the PhD degree in computer science. His research interests include formal methods and theory of distributed computing.

Marzio De Biasi received the MSc degree in computer science from the University of Udine, Italy, in 2000. He is currently employed as a software engineer and is primarily focused on the development of internet applications to control and monitor remote hardware devices. He also continues studying theoretical computer science and doing independent research on the computational complexity of puzzles and games.

Yu Huang received the BSc and PhD degrees in computer science from the University of Science and Technology of China. He is currently an associate professor in the Department of Computer Science and Technology at Nanjing University. His research interests include distributed computing theory, formal specification and verification, and pervasive context-aware computing. He is a member of the IEEE and the China Computer Federation.

Jiannong Cao received the BSc degree from Nanjing University, China, in 1982, and the MSc and PhD degrees from Washington State University, in 1986 and 1990, respectively, all in computer science. He is currently a chair professor and the head of the Department of Computing at Hong Kong Polytechnic University. His research interests include parallel and distributed computing, computer networks, mobile and pervasive computing, fault tolerance, and middleware. He is the chair of Technical Committee on Distributed Computing, IEEE Computer Society, a fellow of the IEEE, a member of ACM, and a senior member of China Computer Federation.

Jian Lu received the BSc, MSc, and PhD degrees in computer science from Nanjing University, China. He is currently a professor in the Department of Computer Science and Technology and the director of the State Key Laboratory for Novel Software Technology at Nanjing University. He serves on the Board of the International Institute for Software Technology of the United Nations University (UNU-IIST). He also serves as the director of the Software Engineering Technical Committee of the China Computer Federation. His research interests include software methodologies, software automation, software agents, and middleware systems.