MULTI-WRITER CONSISTENCY CONDITIONS FOR SHARED MEMORY OBJECTS

A Thesis

by

CHENG SHAO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2007

Major Subject: Computer Science

MULTI-WRITER CONSISTENCY CONDITIONS FOR SHARED MEMORY OBJECTS

A Thesis

by

CHENG SHAO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Jennifer L. Welch
Committee Members,	Jianer Chen
	Alexander Parlos
	Vivek Sarin
Head of Department,	Valerie E. Taylor

December 2007

Major Subject: Computer Science

ABSTRACT

Multi-Writer Consistency Conditions for Shared Memory Objects. (December 2007)

Cheng Shao, B.E., Tianjin University;

M.E., Institute of Computing Technology, China Academia of Science

Chair of Advisory Committee: Dr. Jennifer L. Welch

Regularity is a shared memory consistency condition that has received considerable attention, notably in connection with quorum-based shared memory. Lamport's original definition of regularity assumed a single-writer model, however, and is not well defined when each shared variable may have multiple writers. In this thesis, we address this need by formally extending the notion of regularity to a multi-writer model. We have shown that the extension is not trivial. While there exist various ways to extend the single-writer definition, the resulting definitions will have different strengths. Specifically, we give several possible definitions of regularity in the presence of multiple writers. We then present a quorum-based algorithm to implement each of the proposed definitions and prove them correct. We study the relationships between these definitions and a number of other well-known consistency conditions, and give a partial order describing the relative strengths of these consistency conditions. Finally, we provide a practical context for our results by studying the correctness of two well-known algorithms for mutual exclusion under each of our proposed consistency conditions.

To My Family

ACKNOWLEDGMENTS

First, I would like to thank my advisor Dr. Jennifer L. Welch. You are a wonderful person and a great mentor. I will never forget that it is you who taught me the methodology of theoretical research, who showed me how to follow a rigorous research approach, who gave me the guidance and support in my difficult time and who, with great patience, accompanies me on this long path of my graduate study.

Dr. Evelyn Pierce made significant contribution during the early stage of the research work. It would be impossible to achieve this without her help. Wish her all the good luck in her future career.

I would also like to express my gratitude to the members of my advisory committee, Dr. Jianer Chen, Dr. Alexander Parlos, and Dr. Vivek Sarin, for all the help you extended to me during my study and research.

Finally, I would like to thank my parents and my family. I love you all.

This work was supported in part by NSF grant 0098305, NSF grant 0500265, Texas Higher Education Coordinating Board grant ARP-00512-0091-2001, Texas Higher Education Coordinating Board grant ARP-00512-0007-2006, and Texas Engineering Experiment Station funds.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

A. Overview

Distributed computer systems are ubiquitous today, ranging from multiprocessors to local area networks to wide-area networks such as the Internet. Shared memory, the exchange of information between processes by the reading and writing of shared objects, is an important mechanism for interprocess communications in distributed systems. A *consistency condition* in a shared memory system is a set of constraints on values returned by data accesses when those accesses may be interleaved or overlapping. A shared memory system with a strong consistency condition may be easy to design protocols for, but may require a high-cost implementation. Conversely, a shared memory system with a weak consistency condition may be easy to implement, but difficult for the user to program or reason about. Finding a consistency condition that can be implemented efficiently and that is nonetheless strong enough to solve practical problems is one of the aims of shared memory research.

A desirable consistency condition for shared memory objects is *atomicity* (or *linearizability*) ([16]), in which read and write operations behave as though they were executed sequentially, i.e, with no interleaving or overlap, in a sequence that is consistent with the relative order of non-overlapping operations. In many cases, however, this semantics is difficult to implement, particularly in distributed systems where variables are replicated and where the number of processes with access to the variable is not known in advance. For such systems, the related but weaker condition of *regularity* ([16]) may be easier to implement while retaining some usefulness. For this

---

The journal model is *IEEE Transactions on Automatic Control.*

reason, it has received considerable attention in its own right, notably in connection with *quorum-based shared memory* ([2], [20], [19] and [18]).

Informally speaking, regularity requires that every read operation return either the value written by the latest preceding write (in real time) or that of some write that overlaps the read. This description is sufficiently clear for the single-writer model[1], in which the order of the writes performed on a given variable in any execution is well-defined; in fact, it was for this model that Lamport gave his formal definition of regularity. In a multi-writer model, however, multiple processes may perform overlapping write operations to the same variable so that the "latest preceding write" for a given read may have no obvious definition.

A common way to circumvent this problem is to rely on a plausible generalization of the informal definition above, e.g. the following, which appears in [20]:

- A read operation that is concurrent with no write operations returns a value written by the last preceding write operation in some serialization of all preceding write operations, and

- A read operation that is concurrent with one or more write operations returns either the value written by the last preceding write operation in some serialization of all preceding write operations, or any of the values being written in the concurrent write operations.

Such a definition, however, leaves a good deal of room for interpretation. What is meant by "some serialization" in this context? Is there a single serialization of the writes for which the above is true for all read operations, or does it suffice for there to be some (possibly different) such serialization for *each* operation? Or should all

[1]In the *single-writer* model, only one process can write to a shared object; other processes can only read from it.

read operations of the same *process* perceive writes as occurring in the same order? Such ambiguities can be avoided with a formal definition of multi-writer regularity, but to our knowledge none has yet been proposed.

B. Contributions

In this thesis, we formally extend the notion of regularity to a multi-writer model. Specifically, we give several possible formal definitions of regularity in the presence of multiple writers. We then present a quorum-based algorithm to implement each of these definitions and prove the algorithms correct.

The definitions form a lattice as respect to their strength, and the implementations have varying costs with respect to number of messages, size of messages, time delay, and local memory requirements. Taken together, the set of definitions point out the ambiguity of the informal notion of regularity and the algorithms suggest that different costs may be associated with different choices for disambiguating.

If a consistency condition is said to be local, it means that the whole shared memory system satisfies the consistency condition if and only if the consistency condition is satisfied on per-variable basis. Locality is a desired property of consistency conditions: as mentioned in [12], locality enhances modularity and concurrency. In our study, we show that all the proposed definitions satisfy locality.

We also study the relationships between our definitions of multi-writer regularity and several existing consistency conditions: linearizability ([12]), sequential consistency ([15]), coherence ([10]), PRAM ([17]) and PCG ([1]). As part of this analysis, we give a partial order describing the relative strengths of these consistency conditions.

Finally, we provide a practical context for our results by studying the correctness of two well-known algorithms for mutual exclusion when the variables are im-

plemented under our proposed consistency conditions. The algorithms we examine are Peterson's algorithm for 2 processes ([23]) and Dijkstra's algorithm ([24]). We find that Peterson's algorithm is fully correct under all the conditions. Dijkstra's algorithm satisfies only some of the constraints of the mutual exclusion problem under any of the conditions.

C. Related Work

There is copious literature on consistency conditions for shared memory, both implementations and applications (e.g., [15], [17], [10], [12], [1], [4] and [25]). Our work builds on the the notion of regularity as introduced in [16]. A consistency condition called *Normality* was introduced in [11], which, when all the operations are unary[2], is equivalent to atomicity. As it turns out, all of our proposed definitions are weaker than Normality.

We use a similar approach as in [27] by identifying building blocks and using various combinations of the building blocks to explore potential consistency conditions. The benefit of this approach is that consistency conditions can be easily organized into a lattice. The difference between our work and [27] is that in [27] the building blocks are identified in the definition level while in our work, the building blocks are identified in the implementation level.

We use a similar framework and system model as introduced in [26] to define our proposed definitions as well as those well-established consistency conditions. The only difference is the partial order used in the frameworks. In [26], the partial order is a combination of per-process order and the 'read from' relation. In our work, the partial order is the real time order among operations.

---

[2]An operation is unary if it only involves a single shared object.

We follow the example of [3], [1] and [13] in using the mutual exclusion problem as an application for our consistency conditions. In [3], Attiya and Friedman revised Peterson's 2-process algorithm ([23]) to solve the mutual exclusion problem under their hybrid consistency model. In [1], Ahamad et al. examined the correctness of Peterson's algorithm and Lamport's bakery algorithm under the PCG consistency model, showing that Peterson's algorithm solves the mutual exclusion problem under PCG, while Lamport's algorithm fails to do so. In a later study, Higham et al. ([13]) investigated other mutual exclusion algorithms, including Dekker's and Dijkstra's, none of which guarantees mutual exclusion under PCG.

D. Roadmap of Thesis

The rest of the thesis is organized as follows. Chapter II defines our system model and gives a generic algorithm that uses quorum systems to implement a shared read/write object. Chapter III presents our proposed definitions of multi-writer regularity and their implementation algorithms. Chapter IV discusses the locality property of our proposed definitions and compares their relative strength. Chapter V studies the correctness of two mutual exclusion algorithms, Peterson's algorithm for two processes and Dijkstra's algorithm, when variables are implemented under our proposed definitions Chapter VI concludes this thesis and discusses future work.

A preliminary version of this thesis was published in [28].

CHAPTER II

PRELIMINARIES

A. Shared Read/Write Registers and Consistency Conditions

We assume a concurrent system composed of n application processes, $p_0, \ldots, p_{n-1}$, and some number of shared objects. In this thesis, we focus on *read/write registers*. Such a register, x, supports two operations, *read* and *write*, which can be executed by the processes. Each operation has a set of *invocations* and a set of *responses*. For a read operation, the invocation by process p_i is denoted $read_i(x)$ and the responses have the form $return_i(x, v)$, where v is the return value. For a write operation, the invocations by process p_i have the form $write_i(x, v)$, where v is the value to be written, and the response is denoted $ack_i(x)$.

The behavior of the shared register in the presence of concurrent accesses by different processes is defined with respect to the desired behavior of the register in the absence of concurrency, so we first define the latter.

When there are no concurrent accesses, the invocation of each operation is directly followed by its matching response, and this pair forms an indivisible operation. The *sequential specification* of a read/write register is the set of all sequences of read and write operations such that each read operation returns the value of the latest preceding write operation; if there is no preceding write, then the read returns the initial value of the register.

Definition 1 *A sequence of operations on a shared object is* legal *if it belongs to the sequential specification of the shared object.*

We now return to considering behavior of the register in the presence of concurrent accesses. In this situation, invocations and responses can be interleaved, although

we assume that each process has at most one operation pending at a time. To capture such "well-formedness" constraints, we define the notion of a schedule next. If σ is a sequence of operation invocations and responses, we denote by $\sigma|i$ the subsequence of σ containing all the invocations and responses performed by process p_i.

Definition 2 *A sequence σ of invocations and responses is a* schedule *if, for each i, $0 \leq i < n$, the following hold:*

- *$\sigma|i$ consists of alternating invocations and matching responses, beginning with an invocation; and*
- *if the number of steps taken by p_i is finite, then the last step by p_i is a response, i.e., every invocation has a matching response.*

Note that this definition of a schedule allows arbitrary asynchrony of process steps, i.e., no constraints are placed on the relative speed with which operations complete or on the time between operation invocations. However, for convenience of analysis, we follow the example of [16] and [9] in employing the useful abstraction of an imaginary global clock. All our references to "real time" in the sequel are with respect to this imaginary clock, which is not available to the processes themselves. This is equivalent to the global-time model introduced in [8] and [5].

The key remaining point is what values should be returned by the reads? This is defined by a consistency condition. Most consistency conditions define a connection between the behavior of the register in the presence of concurrency and the register's sequential specification. Formally, a *consistency condition* is specified by a particular set of schedules. Thus, the relative "strength" of two consistency conditions can be compared by considering the sets of schedules defining each condition, as follows. Given two consistency conditions C_1 and C_2, C_1 is *at least as strong as* C_2 if $C_1 \subseteq C_2$.

Furthermore, C_1 is *stronger than* C_2 if $C_1 \subset C_2$. It is also easy to define a consistency condition as the union or intersection of two other consistency conditions[1].

The following pieces of notation are useful for defining particular consistency conditions.

Given a schedule σ, we use the expression $ops(\sigma)$ to denote the set of all operations whose invocations and responses appear in σ[2]. By the definition of a schedule, each invocation has a matching response, namely the response by the same process that follows it most closely; an invocation and its matching response form an operation. Furthermore, $ops(\sigma|i)$ denotes the set of all operations that are performed in σ by process p_i.

For a shared variable x, $ops(\sigma|x)$ denotes the set of all operations that are performed on x.

Finally, we let $writes(\sigma)$ denote the set of all write operations in schedule σ.

Informally speaking, a permutation on a subset of $ops(\sigma)$ is σ-*consistent* if it preserves the partial order of the operations in σ.[3] Before giving a more formal defintion, we first define a partial order $<_\sigma$ on $ops(\sigma)$: For two operations o_1 and o_2 in σ, $o_1 <_\sigma o_2$ iff the response for o_1 precedes the invocation for o_2 in σ.

Definition 3 *Given a schedule σ, a permutation π of a subset of $ops(\sigma)$ is* σ-consistent *if, for any operations o_1 and o_2 in π, o_1 precedes o_2 in π whenever $o_1 <_\sigma o_2$.*

---

[1]For example, as we elaborate on later in the thesis, PCG can be represented by Coherence $\cap$ PRAM, where PCG, Coherence, and PRAM are known consistency conditions.

[2]Assume for convenience that each operation in sigma has a unique id, for instance, the j-th operation invoked by process i; this mechanism allows us to distinguish between two reads of the same value by the same process that occur at different points in the schedule.

[3]In most situations of interest, σ represents the order of operation invocations and responses in real time.

Furthermore, we use $op_1 \not\succ_\sigma op_2$ to denote that operation op_1 starts before op_2 ends in schedule σ. According to [16], the following relation holds:

$$\text{If } op_1 <_\sigma op_2 \not\succ_\sigma op_3 <_\sigma op_4, \text{ then } op_1 <_\sigma op_4$$

Several of the definitions we present in this thesis rely on the notion of a read operation "reading from" a write operation. Formally:

Definition 4 *Given a schedule σ, consider a function ρ from the set of read operations in σ to the set of write operations in σ. ρ is called a* reads-from *function if for every read operation r, r and $\rho(r)$ operate on the same shared variable, the value returned by r is the same as the value written by $\rho(r)$ and $\rho(r) \not\succ_\sigma r$. We say that r* reads from *$\rho(r)$.*

We conclude this section by giving an important example of a consistency condition in our framework. The original definition of regularity by Lamport ([16]) assumed that only one process performs writes on a given shared register and stated that every read returns either the value of the latest preceding write or the value of some overlapping write. Since there is only one writer and it performs operations sequentially, the notion of "latest preceding write" is well-defined. A rephrasing of this definition that links the concurrent behavior to the sequential specification is given next, where a *single-writer* schedule is a schedule in which every write operation is performed by the same process.

Definition 5 (Single-Writer Regularity or SWReg) *A single-writer schedule σ satisfies SWReg iff for every read r in $ops(\sigma)$, there exists a permutation π_r of* $\text{writes}(\sigma) \cup \{r\}$ *such that*

- *π_r is legal, and*

- π_r *is* σ*-consistent.*

B. Quorum Systems

Although having processes communicate through shared variables is generally viewed as desirable from a software development perspective, most distributed systems do not directly provide such functionality. However, the illusion of shared variables can be provided through a shared variable simulation layer that runs in a message-passing communication environment. This software layer simulates shared registers on top of the message-passing layer.

The algorithms in this thesis for simulating a shared register use the notion of a quorum system, which is a technique for handling replicated data. Some processes in the system play the role of "servers", which maintain replicas, while others play the role of "clients", which handle invocations of operations on the replicated data. There is one client process corresponding to each application process introduced in the previous section. Let $P_{\mathcal{S}}$ be the set of server processes and $P_{\mathcal{C}}$ be the set of client processes. (It is possible for a single physical node to host both a client and a server process.)

A quorum system $\mathcal{Q}$ (over $P_{\mathcal{S}}$) is a collection of subsets of $P_{\mathcal{S}}$, each of which is called a *quorum*, satisfying the property that for every two distinct quorums Q and Q', $Q \cap Q' \neq \emptyset$.

C. System Model

In this section, we provide definitions for modeling our execution environment. There is a collection $P = P_{\mathcal{S}} \cup P_{\mathcal{C}}$ of processes that communicate with each other through message-passing.

Each *process* is modeled as a (possibly infinite) state machine, with an initial state and a transition function. The state machine represents the code for the simulation layer that is running at the process.

A *configuration* of the system is a vector of local states, one per process. An *initial* configuration contains an initial state for each process.

There are two kinds of *events* that can occur in the system, input and output events. Each event occurs at a single process. The input events are the receipt of a message and the invocation of a shared register operation. The output events are the sending of a message and the response of a shared register operation.

Each input event triggers its corresponding process to take a step: the transition function is applied to the current state of the process and the particular event, and produces a new state of the process and a set of output events. The output events consist of a set of messages sent by the process and at most one shared-object operation response to occur at the process.

An *event list* is a sequence of events, all taking place at the same process, that begins with an input, followed by any number of message sends, and ends with at most one operation response.

An *execution* is a sequence $d_0\ell_1 d_1\ell_2 d_2 \ldots$ of alternating configurations d_k and event lists ℓ_k, starting with an initial configuration d_0, that satisfies the following conditions.

- Consider any $d_{k-1}\ell_k d_k$ in the sequence, where ℓ_k takes place at process q_i. Then applying q_i's transition function to q_i's state in d_{k-1} and the first event in ℓ_k produces the remaining events in ℓ_k and q_i's state in d_k. All other components of d_k are the same as in d_{k-1}. That is, the process states and events occurring in the sequence are consistent with the processes' transition functions (i.e., the

shared variable simulation algorithm).

- Every message sent is received exactly once and subsequent to its send; only messages sent are received. That is, the communication is reliable.

- If an operation invocation occurs in some event list ℓ_k occurring at process q_i, then the most recent preceding invocation or response at q_i (if any) is a response. That is, the application process that is generating the invocations to the client process waits for one operation to finish before invoking the next one.

We can now state our main correctness condition for simulating a register with a particular consistency condition.

Definition 6 *The system* implements a read/write register with consistency condition C *if, for every execution of the system, the projection onto the set of invocations and responses of the register is a schedule that is in (i.e., satisfies)* C.

D. Generic Algorithm

A generic algorithm that uses quorums to implement a shared read/write register with initial value v_0 is given in Figure 1. Upon receiving a read or write request on the shared object, a client process chooses a quorum using some quorum selection strategy and then queries each member of this quorum about its current "view" of the shared object, which consists of the value of the object and the timestamp associated with the value. After gathering all the responses, the process chooses a set of views to work with, using the function $ChooseViews()$, and then decides which timestamp among the chosen views is the latest, using function $MaxTS()$. The operations then continue as follows:

- **write:** The process increments the timestamp returned by $MaxTS()$, using the function $IncTS()$, and writes the new value and the incremented timestamp back to every member of some quorum.

- **read:** The process uses a function $GetValue()$ to obtain a value associated with the timestamp selected by $MaxTS()$, and returns that value as the result of the read. The reading process then uses the function $OptionalWriteBack()$ to notify a subset of the processes (which can be either no processes or a quorum of processes) of the value it plans to return before it actually returns.

By plugging in different implementations for the functions $MaxTS()$, $IncTS()$, $GetValue()$, and $ChooseViews()$, and choosing whether to call $OptionalWriteBack()$, we obtain registers satisfying different consistency conditions.

This algorithm is a generalization of several existing quorum-based protocols. For example, the appropriate instantiations of the functions yield the algorithms in [20], [7] and [21].

The following lemma states a property of the generic algorithm that we will use later.

Lemma 1 *The sequence of timestamp values taken on by the replica on any server is non-decreasing during any execution of the generic algorithm.*

We define the *timestamp of a write operation* as the timestamp the write operation uses to write to a quorum in Line 7 of the *write* procedure in Figure 1. The *timestamp of a read operation* is the timestamp value associated with the variable value returned by the read in Line 5 of the *read* procedure in Figure 1. For both cases, we use the expression $ts(op)$ to denote the timestamp of operation op.

We now define the reads-from function for the generic algorithm. Given a schedule σ, let ρ be any function from the read operations in σ to the write operations in

σ such that for every read r, r does not end before $\rho(r)$ starts, and $ts(r) = ts(\rho(r))$. There always exists such a function ρ since a read r will not observe a timestamp t unless it has already been written by some write that started before r ends.

Code for client process $c_i \in P_{\mathcal{C}}$:

local variables:
  val /* local copy of shared variable, initially v_0 */
  ts /* local copy of timestamp, initially smallest timestamp value */

$write_i(x, v)$:
1 for some quorum $Q \in \mathcal{Q}$, send $\langle \text{READ} \rangle$ to each $s_j \in Q$;
2 wait to receive $\langle \text{VIEW}, v, t \rangle$ from each $s_j \in Q$;
3 $V := ChooseViews()$;
4 $ts := MaxTS(V)$;
5 $ts := IncTS(ts)$;
6 $val := v$;
7 for some quorum $Q' \in \mathcal{Q}$, send $\langle \text{WRITE}, val, ts \rangle$ to each $s_j \in Q'$;
8 wait to receive $\langle \text{ACK} \rangle$ from each $s_j \in Q'$;
9 $ack_i(x)$;

$read_i(x)$:
1 for some quorum $Q \in \mathcal{Q}$, send $\langle \text{READ} \rangle$ to each $s_j \in Q$;
2 wait to receive $\langle \text{VIEW}, v, t \rangle$ from each $p_j \in Q$;
3 $V := ChooseViews()$;
4 $ts := MaxTS(V)$;
5 $val := GetValue(V, ts)$;
6 $OptionalWriteBack()$;
7 $return_i(x, val)$;

Code for server process $s_j \in P_{\mathcal{S}}$:

local variables:
  val /* local copy of shared variable, initially v_0 */
  ts /* local copy of timestamp, initially smallest timestamp value */

When s_j receives $\langle \text{READ} \rangle$ from c_i:
1 send $\langle \text{VIEW}, val, ts \rangle$ to c_i;

When s_j receives $\langle \text{WRITE}, v, t \rangle$ from c_i:
1 if $(ts < t)$ then
2 $val = v; ts = t;$
3 endif

4 send $\langle \text{ACK} \rangle$ to c_i;

Fig. 1. A generic quorum-based algorithm to implement a shared read/write register

CHAPTER III

MULTI-WRITER REGULAR VARIABLES: SPECIFICATIONS AND IMPLEMENTATIONS

Suppose we instantiate the generic algorithm as follows:

- The timestamp is chosen from the set of natural numbers $\mathcal{N}$;
- Function $MaxTS()$ returns the largest timestamp in numerical order and function $IncTS()$ increments the timestamp by 1;
- Function $GetValue()$ returns the value that is associated with the timestamp returned by $MaxTS()$;
- $ChooseViews()$ combines all the query results from a certain quorum to form a view;
- Function $OptionalWriteBack()$ does nothing;

If only one designated process invokes the write procedure, the resulting algorithm implements a read/write register that satisfies single-writer regularity (Definition 5).

What if more than one process is allowed to perform the write operation? Would the resulting behavior qualify as a possible specification for multi-writer regularity? In the rest of the thesis, we explore this question. But first, let us take a look at various ways the generic algorithm can be instantiated, which, as it turns out, gives us consistency conditions with different strength, and provides us possible definitions of multi-writer regularity.

A. Building Blocks in the Generic Algorithm

We identify three building blocks that we use to create a specific instantiation from the generic algorithm. Different combinations of the building blocks will give us different algorithms, which in turn yield shared variables with different consistency conditions. The three building blocks are:

- **Including unique id in timestamp**: If we do not use this building block, then the timestamp is simply chosen from the set of natural numbers $\mathcal{N}$. Function $MaxTS()$ returns the largest timestamp in numerical order, while $IncTS(ts)$ increments its argument by 1. Since timestamps are not necessarily unique when using this building block, several different values may share the same largest timestamp value. In this case, $GetValue()$ simply chooses one arbitrarily and returns the corresponding value.

  When we use this building block, we define the timestamp as a pair $\langle ts, id \rangle$, where ts is a natural number, and id is a unique process id. For timestamps of this format, we define $MaxTS()$ as the function that returns the largest timestamp in lexicographic order among the pairs. Because this timestamp is unique, $GetValue()$ simply returns the unique value associated with it. Finally, $IncTS()$ increments the ts field by 1 and places the calling process identifier in the id field. The cost of using this building block is an additional $O(\log n)$ bits to store a timestamp. However, this gives a total order on the timestamps which can yield stronger consistency conditions.

- **Write-back phase in the read procedure**: When we do not use this building block, function $OptionalWriteBack()$ in the read procedure of the generic algorithm will simply do nothing and return.

If the building block is used, function $OptionalWriteBack()$ will select a quorum and write the value and timestamp returned from $GetValue()$ and $MaxTS()$ to each of the servers in the quorum. Then it waits until it receives all the acknowledgements from each server. The cost of using this building block is an extra $O(c)$ messages, where c is the size of the biggest quorum in the system. Furthermore, the time for a read is increased by a round-trip message delay. However, we can obtain stronger consistency conditions by using this building block.

- **Local cache on clients**: If we do not keep a local cache at each client, function $ChooseViews()$ in the generic algorithm will collect the query result from each server in a certain quorum and form a view.

  If we do use this building block, then each client will keep a local cache that stores the latest value-timestamp pair that it knows about. In other words, if the latest operation this client has performed is a write operation, then the local cache holds the value-timestamp pair that has been sent to update a quorum of servers; if it is a read operation, then the local cache holds what the read has returned. Function $ChooseView()$ will form a view by combining the query result from a certain quorum and the current content in the local cache. The cost of using this building block is that clients now have to keep this information for each shared variable and thus it introduces space and robustness issues. As with the other two techniques, however, this will also yield a stronger consistency condition.

The overall result of our study is shown in Figure 2. The left lattice in Figure 2 shows all the algorithms instantiated from the generic algorithm by applying different combinations of the building blocks. We use "ID" to denote the fact that the first

building block gets used when instantiating the algorithm, and "WB" for the second, and "LC" the third. So for example, the algorithm represented by "ID, WB" is the one that uses process id in the timestamp and a write-back phase in the read. The right lattice in Figure 2 shows the corresponding consistency conditions that are yielded from the algorithms. An arrow from consistency condition A to consistency condition B indicates B is stronger than A. As we will present later in Section I, all the consistency conditions in the right lattice of Figure 2, except atomicity and MWReg+, are possible definitions of multi-writer regularity in the sense that when there is only a single writer in the system, all the definitions are equivalent to SWReg (MWReg+ is stronger than SWReg, but it is still weaker than atomicity.).

In the remainder of this chapter, we will walk up the lattice and introduce each of the possible definitions of multi-writer regularity and their implementation algorithms. It is worth noting that right now we only focus on a single-variable shared memory. Later in Chapter IV, we will show how to extend our results to a multi-variable shared memory system.

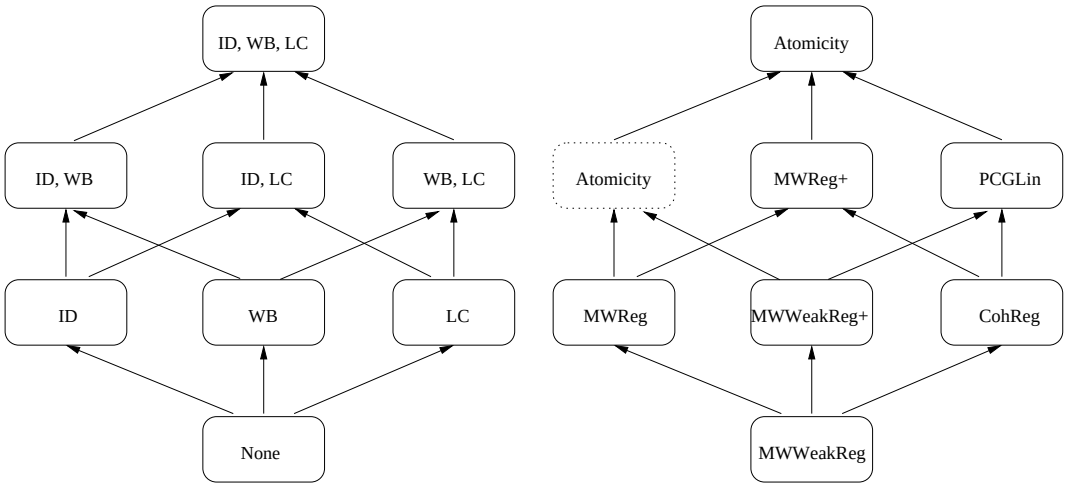

Fig. 2. Lattice of algorithms and consistency conditions

B. MWWeakReg

1. Specifying MWWeakReg

Our first proposed definition for multi-writer regularity is simply to use Definition 5, which we restate here with a new name.

Definition 7 (MWWeakReg) *A schedule* σ satisfies MWWeakReg *if, for all read operations* r *in* $ops(\sigma)$*, there exists a permutation* π_r *of* $writes(\sigma) \cup \{r\}$ *such that:*

- π_r *is legal, and*
- π_r *is* σ*-consistent.*

A shared memory object satisfies MWWeakReg if all schedules on that object satisfy MWWeakReg.

Informally, a schedule σ satisfies MWWeakReg if each read $r \in ops(\sigma)$ returns the value of some write w that either overlaps or precedes r in σ, as long as no other write falls completely between w and r. Note that this definition allows different reads to behave as though the set of writes occurred in different orders, as long as all such orderings are consistent with the partial order of the writes in σ.

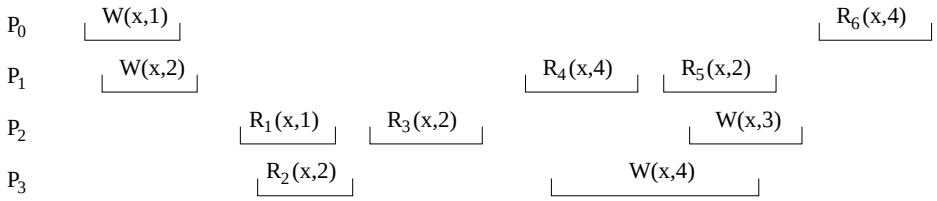

Fig. 3. Schedule that satisfies MWWeakReg

Figure 3 shows a schedule that satisfies MWWeakReg. (In our figures, $W(x, v)$ denotes a write operation that writes value v to variable x, and $R(x, v)$ denotes a

read operation on variable x that returns value v. We will use similar schedules to illustrate other proposed definitions. The schedules differ in the return values of some of the read operations.) The permutation for each read operation is given below. It is easy to verify that all the permutations are legal and σ-consistent.

$$\begin{array}{ll} R_1\text{:} & W(x,2), W(x,1), R_1(x,1), W(x,4), W(x,3) \\ R_2\text{:} & W(x,1), W(x,2), R_2(x,2), W(x,3), W(x,4) \\ R_3\text{:} & W(x,1), W(x,2), R_3(x,2), W(x,4), W(x,3) \\ R_4\text{:} & W(x,1), W(x,2), W(x,4), R_4(x,4), W(x,3) \\ R_5\text{:} & W(x,1), W(x,2), R_5(x,2), W(x,3), W(x,4) \\ R_6\text{:} & W(x,1), W(x,2), W(x,3), W(x,4), R_6(x,4) \end{array}$$

2. Implementing MWWeakReg

We name the implementation algorithm for MWWeakReg *Alg_None*, which does not use any of the building blocks introduced in Section A.

The following lemma states that the *timestamp order* (numerical order by timestamp) of certain operations extends the partial order $<_\sigma$.

Lemma 2 *For any schedule σ on a shared register implemented by Alg_None, there exist the following relationships between the operations and their timestamps:*

(a) For any read operation r and any write operation w: if $w <_\sigma r$, then $ts(w) \leq ts(r)$.

(b) For any two write operations w_1 and w_2: if $w_1 <_\sigma w_2$, then $ts(w_1) < ts(w_2)$.

Proof. (a) Suppose write w ends before read r begins in σ. Let s be a server process that is in the intersection of the quorum that w uses for its update (Lines 7-8) and the quorum that r uses for its query (Lines 1-2). According to Lemma 1, the sequence of timestamp values taken on at s is non-decreasing. Since w finishes before r starts, s returns to r a timestamp that is at least $ts(w)$. Since r chooses the value associated wtih the largest timestamp returned from its query, r's timestamp is no less than w's.

(b) Using a similar argument to that in (a), we can show that some server process s returns to w_2 a timestamp that is at least as large as $ts(w_1)$. Since $ts(w_2)$ is larger than the largest timestamp obtained in the query, w_2's timestamp is larger than w_1's. ■

Theorem 1 *Algorithm Alg_None implements MWWeakReg.*

Proof. Consider any execution of Alg_None and let σ be its schedule. For each read operation r in σ, we construct π_r as follows. We partition the set of writes into two subsets:

- The set of writes that begin before r ends and whose timestamps are at most that of r,
- the set of all remaining writes

Each of these two sets is arranged in increasing order of timestamp; writes with identical timestamps are ordered arbitrarily with the exception that for the first set, the write operation that r reads from will be arranged as the last operation in the first sequence. We append r at the end of the first sequence and then append the second sequence to the first sequence.

The reader can easily verify that the resulting sequence satisfies the two conditions of MWWeakReg. ■

Now let us take a look on how Alg_None can generate the schedule in Figure 3. We assume that we have a quorum system on three server processes A, B and C, with every set of two server processes forming a quorum. We also assume that the initial value and timestamp of the shared object is $(0, 0)$ in each server's local copy. Figure 4 shows an execution of Alg_None on four processes, whose projection onto the set of

invocations and responses of the shared register x is the schedule shown in Figure 3. The following items will help understand the figure:

- Steps in boldface are invocations or responses on the share register.
- $query(A) = (v, t)$ means querying server A gets a value-timestamp pair of (v, t).
- $update(A, (v, t))$ means updating the local copy of server A with value-timestamp pair of (v, t).
- $ts := 1$ means we choose timestamp value 1 for the succeeding updates.

We will use the same model and the same denotation afterwards when illustrating how an implementation algorithm could generate a certain schedule.

We now explain the purpose of demonstrating these executions. Suppose we have two consistency conditions C_1 and C_2 with $C_1 \subset C_2$, and algorithm A. If we prove that algorithm A implements consistency condition C_2 (i.e., every execution of A satisfies C_2), we have not shown every schedule in C_2 can be generated by A. In fact, it is possible that A actually generates the more stringent condition C_1. By showing that A generates at least one schedule not in C_1, we obtain some knowledge that A is not "too strong".

C. MWReg

MWWeakReg is actually a very weak consistency condition, as the read operations do not have a common view on the order of preceding write operations even for the read operations performed by the same process. For instance, in Figure 3, p_1's first read R_4 reflects the write of 4, but p_1's next read R_5 does not: the later read returns 2 even though the write of 2 precedes the write of 3. It might, therefore, be desirable to construct a stronger definition of regularity for the multi-writer case.

p_0	p_1	p_2	p_3
start $W(x,1)$			
$query(A) = (0,0)$	**start** $W(x,2)$		
$query(B) = (0,0)$	$query(A) = (0,0)$		
$ts := 1$	$query(C) = (0,0)$		
$update(C,(1,1))$	$ts := 1$		
$update(A,(1,1))$	$update(B,(2,1))$		
ack	$update(C,(2,1))$		
	ack		
		start $R_1(x)$	
		$query(A) = (1,1)$	**start** $R_2(x)$
		$query(B) = (2,1)$	$query(B) = (2,1)$
		return 1	$query(C) = (2,1)$
			return 2
		start $R_3(x)$	
		$query(A) = (1,1)$	
		$query(C) = (2,1)$	
		return 2	
	start $R_4(x)$		
			start $W(x,4)$
	$query(B) = (2,1)$		$query(A) = (1,1)$
			$query(B) = (2,1)$
			$ts := 2$
			$update(A,(4,2))$
	$query(A) = (4,2)$		
	return 4		
	start $R_5(x)$		
	$query(B) = (2,1)$	**start** $W(x,3)$	
		$query(B) = (2,1)$	
			$update(B,(4,2))$
	$query(C) = (2,1)$		**ack**
	return 2	$query(C) = (2,1)$	
		$ts := 2$	
		$update(A,(3,2))$	
		$update(C,(3,2))$	
		ack	
start $R_6(x)$			
$query(B) = (4,2)$			
$query(C) = (3,2)$			
return 4			

Fig. 4. An execution of Alg_None that generates the schedule in Figure 3. Time increases going down the page.

1. Specifying MWReg

Consider the schedule shown in Figure 5. We say a write is *relevant* to a read if the invocation of the write is before the response of the read. In this example, $W(x,1)$ and $W(x,2)$ are relevant to all the read operations, and $W(x,3)$ is relevant to R_5 and R_6 only. For any two read operations in this schedule, the write operations that are relevant to both reads will be observed in the same order and that order extends $<_\sigma$. For example, consider read operation R_1 and R_2. The write operations by p_0 and p_1 are relevant to both reads and they are observed by the two reads in the same order as $W(x,1)$, $W(x,2)$. This is not true in the schedule shown in Figure 3, as the same two reads cannot agree on the same order of the two writes.

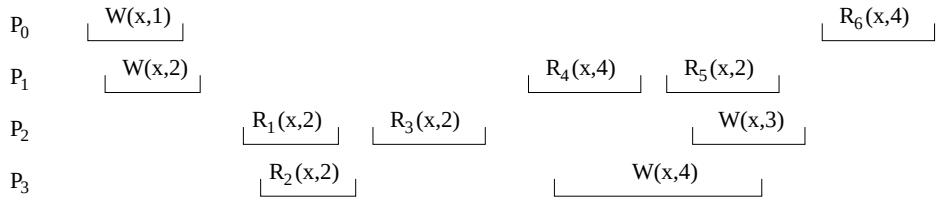

Fig. 5. Schedule that satisfies MWReg

The restriction on the common order of all the "relevant" writes is stronger than what MWWeakReg requires. In order to accommodate behavior of this kind, we propose a more sophisticated definition for our second and stronger version of multi-writer regularity, by requiring any pair of reads to agree only on the ordering of writes that are "relevant" to both of them. Toward this end, we use the following additional notation: $writes_{\leftarrow r}(\sigma) = \{w | w \in writes(\sigma)$ and w begins before r ends in $\sigma\}$.

Definition 8 (MWReg) *A schedule* σ satisfies MWReg *if there exists a permutation* π *of* $ops(\sigma)$ *such that, for all read operations* r *in* $ops(\sigma)$*, the projection* π_r *of* π *onto* $writes_{\leftarrow r}(\sigma) \cup \{r\}$ *satisfies:*

- π_r *is legal, and*
- π_r *is* σ*-consistent.*[1]

A shared memory object satisfies MWReg if all schedules on that object satisfy MWReg.

This definition is similar to that of MWWeakReg, except that for any two reads r_1 and r_2, the set of writes that do not strictly follow either r_1 or r_2 must be perceived by both reads as occurring in the same order. As before, each read returns the value of an overlapping write or of the last preceding write in the order.

The schedule in Figure 5 satisfies MWReg as shown by the following argument. Let $\pi = W(x,1)$,$W(x,2)$, $R_1(x,2)$, $R_2(x,2)$, $R_3(x,2)$, $R_5(x,2)$, $W(x,3)$, $W(x,4)$, $R_4(x,4)$, $R_6(x,4)$. Then the projections for the read operations are:

R_1: $W(x,1), W(x,2), R_1(x,2)$
R_2: $W(x,1), W(x,2), R_2(x,2)$
R_3: $W(x,1), W(x,2), R_3(x,2)$
R_4: $W(x,1), W(x,2), W(x,4), R_4(x,4)$
R_5: $W(x,1), W(x,2), R_5(x,2), W(x,3), W(x,4)$
R_6: $W(x,1), W(x,2), W(x,3), W(x,4), R_6(x,4),$

Notice that the projection for R_4 does not include $W(x,3)$ since it is not relevant to the read. It is easy to verify that all the projections are legal and σ-consistent.

2. Implementing MWReg

We implement a shared variable satisfying MWReg by using the first building block in the generic algorithm, that is, adding the process id (as in [20]) to the timestamps used by the generic algorithm. Since no individual process chooses the same ts value for two different writes, each write operation is guaranteed a unique timestamp value.

---

[1]Note that if there are only a finite number of reads in a given execution, the writes after the last read are not constrained by MWReg to appear in any particular order. We consider this to be acceptable, as such writes are never observed.

This ensures that no matter how many write operations overlap, all read operations that begin after all these write operations finish are able to agree on which is the "last" write. Note that this is a commonly used approach in the implementation of shared variables using quorum systems. We name the resulting algorithm *Alg_ID*. The proof of correctness of *Alg_ID* is based on the following supporting lemma:

Lemma 3 *The write operations performed using Algorithm Alg_ID are totally ordered by timestamp, and this total order extends* $<_\sigma$.

Proof. Because the timestamp includes the process id to break ties, the timestamp order is a total order. Since we still use the timestamp of *Alg_None* as the first field in the lexicographic order, Lemma 2 implies that $<_\sigma$ is preserved among the writes. ■

Theorem 2 *Algorithm Alg_ID implements MWReg.*

Proof. Consider any execution of Alg_ID and let σ be its schedule. We construct the permutation π of $ops(\sigma)$ as follows. We begin by ordering the write operations into a sequence according to their timestamp order. We then insert each read operation r after the write operation that r reads from and before the next write operation in the total order. Read operations with identical timestamps are ordered arbitrarily. Now we prove that for any r, the projection π_r of π satisfies the conditions in Definition 8.

The sequence π_r is legal by construction, as r appears immediately after the write that it reads from.

Now, consider any two operations op_1 and op_2 in π_r such that op_1 finishes before op_2 starts in σ. There are two possible cases:

- op_1 and op_2 are both write operations. Then according to Lemma 3 and our construction method, their order in π_r is σ-consistent.

- op_1 is a write operation and $op_2 = r$. If r reads from op_1, then op_1 appears immediately before r according to our construction. Otherwise, according to algorithm Alg_ID, r reads from a write w whose timestamp is larger than that of op_1. Therefore, the operations appear in π_r in the order op_1, w, and r, and thus the order of op_1 and r is again σ-consistent.

There are no other cases, as writes that begin after r completes are not included in $writes_{\leftarrow r}(\sigma)$, and thus do not appear in π_r. ■

Figure 6 shows an execution of Algorithm Alg_ID that can generate the schedule shown in Figure 5. Comparing with Figure 4, the timestamp includes the process id in addition to the integer counter.

D. MWWeakReg+

MWReg strengthens MWWeakReg by requiring any two read operations to agree on a total order of the write operations that are relevant to both of them and that total order extends $<_\sigma$. An alternative way to strengthen MWWeakReg is to require any two read operations to agree on a partial order of all the write operations and that extends $<_\sigma$. This section explores this alternative.

1. Specifying MWWeakReg+

Before we introduce MWWeakReg+, let us define a causal order relationship among read and write operations on a shared variable. For a given schedule σ and a reads-from function on σ, the *causal order* $<_{co}$ on $ops(\sigma)$ is defined as follows:

- if two operations $op_1 <_\sigma op_2$, then $op_1 <_{co} op_2$,
- if op_2 reads from op_1, then $op_1 <_{co} op_2$,

p_0	p_1	p_2	p_3
start $W(x,1)$			
$query(A) = (0,[0,0])$	**start** $W(x,2)$		
$query(B) = (0,[0,0])$	$query(A) = (0,[0,0])$		
$ts := [1,0]$	$query(C) = (0,[0,0])$		
$update(C,(1,[1,0]))$	$ts := [1,1]$		
$update(A,(1,[1,0]))$	$update(B,(2,[1,1]))$		
ack	$update(C,(2,[1,1]))$		
	ack		
		start $R_1(x)$	
		$query(A) = (1,[1,0])$	**start** $R_2(x)$
		$query(B) = (2,[1,1])$	$query(B) = (2,[1,1])$
		return 2	$query(C) = (2,[1,1])$
			return 2
		start $R_3(x)$	
		$query(A) = (1,[1,0])$	
		$query(C) = (2,[1,1])$	
		return 2	
	start $R_4(x)$		
			start $W(x,4)$
	$query(B) = (2,[1,1])$		$query(A) = (1,[1,0])$
			$query(B) = (2,[1,1])$
			$ts := [2,3]$
			$update(A,(4,[2,3]))$
	$query(A) = (4,[2,3])$		
	return 4		
	start $R_5(x)$		
	$query(B) = (2,[1,1])$	**start** $W(x,3)$	
		$query(B) = (2,[1,1])$	
			$update(B,(4,[2,3]))$
	$query(C) = (2,[1,1])$		**ack**
	return 2	$query(C) = (2,[1,1])$	
		$ts := [2,2]$	
		$update(A,(3,[2,2]))$	
		$update(C,(3,[2,2]))$	
		ack	
start $R_6(x)$			
$query(B) = (4,[2,3])$			
$query(C) = (3,[2,2])$			
return 4			

Fig. 6. An execution of Alg_ID that generates the schedule in Figure 5.

- if $op_1 <_{co} op_3$ and $op_3 <_{co} op_2$, then $op_1 <_{co} op_2$.

Given a schedule σ, and a reads-from function on σ, a permutation π of a subset of $ops(\sigma)$ is *co-consistent* if, for any operations o_1 and o_2 in π, o_1 precedes o_2 in π whenever $o_1 <_{co} o_2$. As we can see, co-consistency is stronger than σ-consistency since $<_{co}$ extends $<_\sigma$ by taking into consideration the "read from" relations between reads and writes[2].

Lemma 4 *$<_{co}$ is a partial order.*

Proof. Suppose in contradiction $<_{co}$ is not a partial order. Let C be a shortest cycle in $<_{co}$. Then C is of the form $op_0, op_1, \ldots, op_{m-1}$ for some even $m \geq 2$, where, for all i, $1 \leq i \leq m/2$,

- op_{2i-1}, a read on variable x_i, reads from op_{2i-2}, a write on x_i, and
- $op_{2i-1} <_\sigma op_{(2i) \bmod m}$.

I.e., the cycle consists of alternating reads and writes, with each read reading from the preceding write, and each write strictly following the preceding read. Note that read operations have odd index and write operations have even index.

For each i, $1 \leq i \leq m/2$, op_{2i-2} begins before op_{2i-1} ends (by definition of reads-from), and op_{2i-1} ends before $op_{(2i) \bmod m}$ begins (by definition of $<_\sigma$). Thus op_0 ends before op_0 begins, which is a contradiction. ■

MWWeakReg+ strengthens MWWeakReg by requiring that the permutation for each read is co-consistent.

---

[2]Our definition of co-consistency is not the same as causal consistency ([4]), which is weaker than real-time ordering, as operations at different processes that are not causally related do not have to be ordered the same as they appear in real time.

Definition 9 (MWWeakReg+) *A schedule* σ satisfies MWWeakReg+ *if there is a reads-from function on* σ *such that for all read operations* r *in* $ops(\sigma)$*, there exists a permutation* π_r *of* $writes(\sigma) \cup \{r\}$ *such that:*

- π_r *is legal, and*
- π_r *is co-consistent.*

A shared memory object satisfies MWWeakReg+ if all schedules on that object satisfy MWWeakReg+.

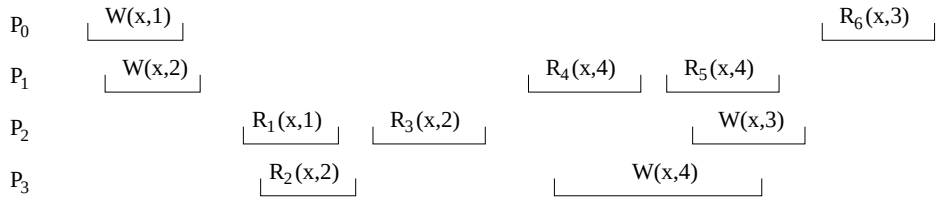

Fig. 7. Schedule that satisfies MWWeakReg+.

The schedules shown in Figure 3 and Figure 5 do not satisfy MWWeakReg+. In both schedules, $W(x,4) <_{co} W(x,3)$, since $W(x,4) <_{co} R_4(x,4)$ and $R_4(x,4) <_{co} W(x,3)$. Thus the value returned by R_6 violates the causal order. Now consider the example shown in Figure 7. The permutation for each read is given below, each of which is legal and co-consistent.

$$
\begin{array}{ll}
R_1\text{:} & W(x,2), W(x,1), R_1(x,1), W(x,4), W(x,3) \\
R_2\text{:} & W(x,1), W(x,2), R_2(x,2), W(x,4), W(x,3) \\
R_3\text{:} & W(x,1), W(x,2), R_3(x,2), W(x,4), W(x,3) \\
R_4\text{:} & W(x,1), W(x,2), W(x,4), R_4(x,4), W(x,3) \\
R_5\text{:} & W(x,1), W(x,2), W(x,4), R_5(x,4), W(x,3) \\
R_6\text{:} & W(x,1), W(x,2), W(x,4), W(x,3), R_6(x,3)
\end{array}
$$

2. Implementing MWWeakReg+

The algorithm we use to implement MWWeakReg+, which we name *Alg_WB*, uses the second building block introduced in Section A, that is, to add a write-back phase in the read operation.

The following lemma shows how the write-back building block changes the relationship between the operations and their timestamps.

Lemma 5 *For any schedule σ on a shared register implemented by Alg_WB, there exists the following relationship between the operations and their timestamps:*

(a) For any read operation r and any write operation w, if $w <_\sigma r$, then $ts(w) \leq ts(r)$.

(b) For any two write operations w_1 and w_2, if $w_1 <_\sigma w_2$ in σ, then $ts(w_1) < ts(w_2)$.

(c) For any read operation r and any write operation w, if $r <_\sigma w$ in σ, then $ts(r) < ts(w)$.

(d) For any two read operation r_1 and r_2, if $r_1 <_\sigma r_2$ in σ, then $ts(r_1) \leq ts(r_2)$.

Proof. The key is the non-empty intersection of quorums used in queries by read and write operations and quorums used in updates by write operations. The proof of Lemma 2 applies here to prove (a) and (b). Essentially the same argument as in (b) is used to prove (c), and essentially the same argument as in (a) is used to prove (d).

■

Theorem 3 *Algorithm Alg_WB implements MWWeakReg+.*

Proof. For any schedule σ resulting from an execution of Alg_WB, we construct π_r for a given read operation r by the following method. We divide all the operations in

$writes(\sigma) \cup \{r\}$ into two groups: $G_1 = \{w | ts(w) \leq ts(r)\} \cup \{r\}$, and $G_2 = \{w | ts(w) > ts(r)\}$.

All the operations of G_1 are followed by all the operations of G_2 in π_r. Within each group, we have the following rules:

- All the write operations in G_1 are arranged by their timestamp order. The write operation that r reads from is put at the end, followed immediately by r.
- All the operations in G_2 are ordered by their timestamp value. If two operations have the same timestamp value, arrange them with arbitrary order.

π_r is legal by our construction. Now we show that π_r is co-consistent. For the first bullet of our causal order definition, if $op_1 <_\sigma op_2$, then from Lemma 5 and our construction, op_1 appears before op_2 in π_r. The second bullet is also satisfied by our construction. For the third bullet, if op_1 appears before op_3, which appears before op_2, then op_1 appears before op_2.

Thus Alg_WB implements a MWWeakReg+ shared variable. ■

Figure 8 gives an execution of algorithm Alg_WB, which will generate the schedule shown in Figure 7.

E. CohReg

1. Specifying CohReg

As discussed earlier, MWWeakReg is very weak in the sense that even from a single process's view, the write operations can be observed in different orders by different reads. As yet another alternative to strengthen MWWeakReg, CohReg adds an additional restriction that any two read operations by the same process must observe

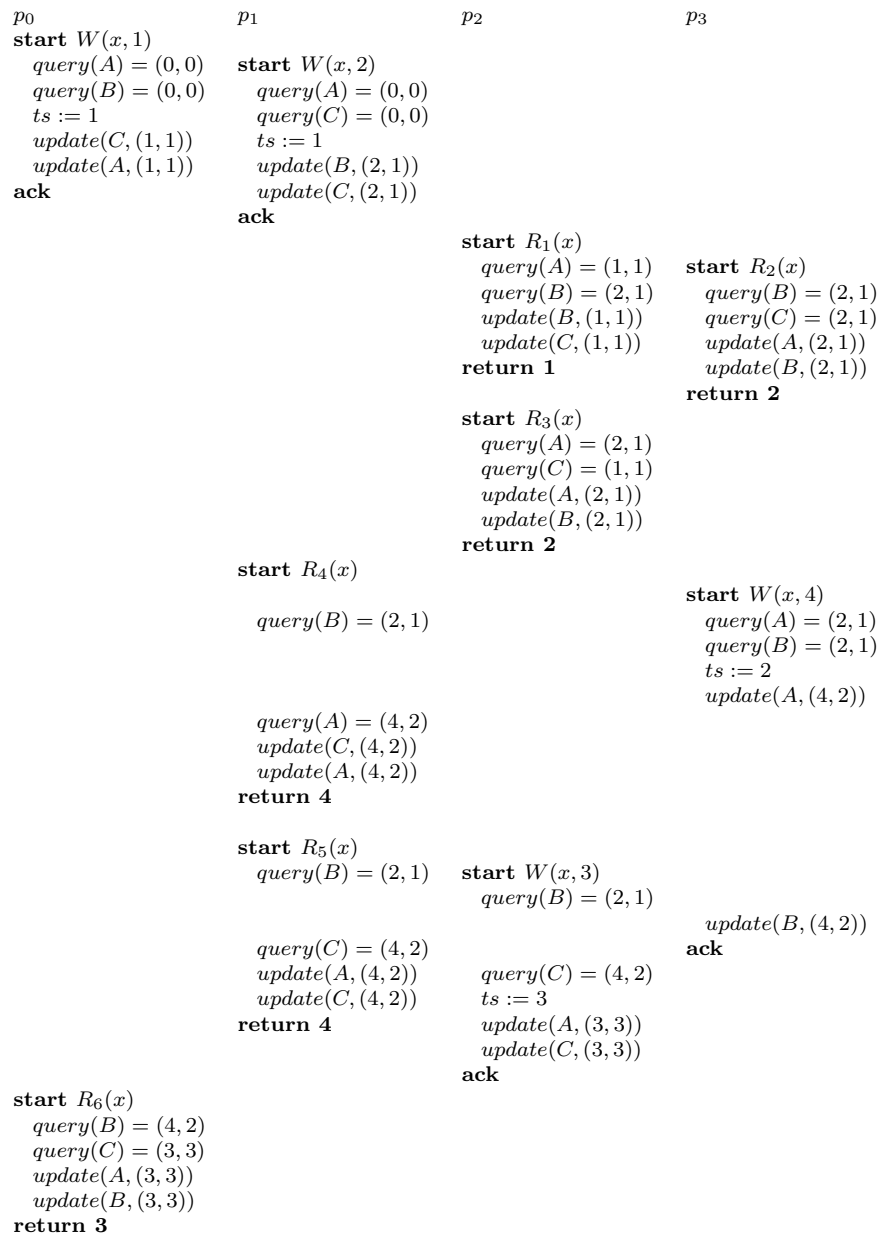

Fig. 8. An execution of Alg_WB that generates the schedule in Figure 7.

the write operations in the same order. Since this condition implies the previously proposed condition known as Coherence ([10]), we name it CohReg.

Definition 10 (CohReg) *A schedule* σ satisfies CohReg *if there exists a reads-from function on* σ *such that for each process* i*, there exists a permutation* π_i *of* $writes(\sigma) \cup ops(\sigma|i)$ *such that:*

- π_i *is legal,*
- $\pi_i|i$ *is* σ*-consistent,*
- *for any read operation* r *by* i*,* $\pi_i|(writes_{\leftarrow r}(\sigma) \cup \{r\})$ *is* σ*-consistent,*
- *for any read operation* r *and any write operation* w *by process* i*, if* $w <_\sigma r$ *and* r *reads from another write* w'*, then* w *appears before* w' *in* π_j *for all* j*; if* $r <_\sigma w$*, then* w *appears after* w' *in* π_j *for all* j*, and*
- *for any two read operations* r_1 *and* r_2 *by process* i*, if* $r_1 <_\sigma r_2$ *and they read from different writes,* w_1 *and* w_2 *respectively, then* w_1 *appears before* w_2 *in* π_j *for all* j*.*

where $\pi_i|i$ *denotes the subsequence of* π_i *consisting just of operations by process* i*, and* $\pi_i|(writes_{\leftarrow r}(\sigma) \cup \{r\})$ *denotes the subsequence of* π_i *consisting of the write operations that start before* r *finishes. A shared memory object satisfies CohReg if all schedules on that object satisfy CohReg.*

The schedule shown in Figure 9 satisfies CohReg. The subsequence of operations for each process is given below, which satisfies all the conditions of CohReg:

p_0: $W(x,1), W(x,2), W(x,3), W(x,4), R_6(x,4)$
p_1: $W(x,1), W(x,2), W(x,3), W(x,4), R_4(x,4), R_5(x,4)$
p_2: $W(x,2), W(x,1), R_1(x,1), R_3(x,1), W(x,3), W(x,4)$
p_3: $W(x,1), W(x,2), R_2(x,2), W(x,4), W(x,3)$

Figure 3 does not satisfy CohReg since it is impossible to construct a permutation for process p_2 that satisfies all the conditions of CohReg.

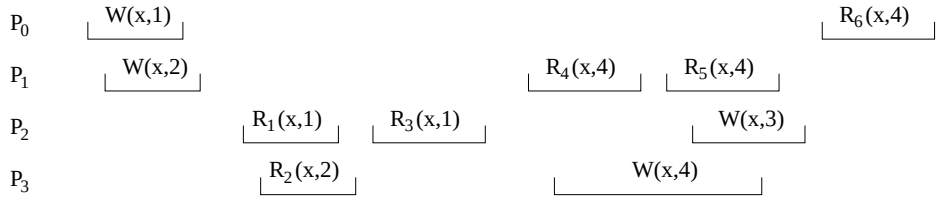

Fig. 9. Schedule that satisfies CohReg.

2. Implementing CohReg

The algorithm we use to implement CohReg uses the third building block introduced in Section A. Let us call this algorithm *Alg_LC*. The following lemma shows how the local cache affects the relationships between two operations and their timestamp order.

Lemma 6 *For any schedule σ on a shared register implemented by Alg_LC, there exists the following relationship between the operations and their timestamps:*

(a) For any two write operations w_1 and w_2: if $w_1 <_\sigma w_2$, then $ts(w_1) < ts(w_2)$.

(b) For any read operation r and any write operation w: if $w <_\sigma r$, then $ts(w) \leq ts(r)$.

(c) For any read operation r and any write operation w by the same process: if $r <_\sigma w$, then $ts(r) < ts(w)$.

(d) For any two read operations r_1 and r_2 by the same process: if $r_1 <_\sigma r_2$, then $ts(r_1) \leq ts(r_2)$. Furthermore, if $ts(r_1) = ts(r_2)$, then they read from the same write.

Proof. The proof of Lemma 2 applies here to prove (a) and first part of (b).

(b) Second part. According to *Alg_LC*, the local cache on each process keeps the latest value-timestamp pair that it knows about, and the timestamp in the cache

never decreases. When the write w finishes, the local cache should store the value and timestamp of w. According to the read procedure, if the largest timestamp from a read's query is no more than what is in the local cache, the read will choose the value in the local cache. Therefore, since r reads the same timestamp as in the local cache, it must read the value in the local cache, which is the value of w. Thus r reads from w.

(c) Since the timestamp never decreases in the local cache as we argued in (b), and the write procedure will increase the timestamp, therefore w's timestamp will be larger than r.

(d) The first part can be proved by using the same argument as in (c). For the second part, according to the read procedure, after r_1 finishes, the local cache should hold the value and timestamp that r_1 returns. Since r_2 reads the same timestamp as r_1, it must read what is in the local cache, which is what r_1 reads. Therefore, r_1 and r_2 read from the same write. ■

Theorem 4 *Algorithm Alg_LC implements CohReg.*

Proof. Given any schedule σ resulting from an execution of Alg_LC, we construct, for each process i, a permutation π_i of operations in $writes(\sigma) \cup ops(\sigma|i)$ as follows.

First we put all the write operations into a sequence according to the increasing order of their timestamp value. Write operations with the same timestamp value are arranged arbitrarily for now. Let π_w be the resulting sequence. The reader can easily verify that π_w is σ-consistent.

Next, we put all the read operations by process i into π_w. If a read operation r reads from a write w and w is not the last one among all the writes that have the same timestamp value as w in π_w, then make it so first. After that, we insert r into π_w between w and the next write in π_w. The read operations between any two writes

should be arranged in order of occurrence. Now we name the final sequence π_i after inserting all the reads of p_i.

For the first condition of CohReg, according to the construction method above, for any read r and the write that r reads from, say w, there is no other write operations between w and r in π. Thus π is legal.

For the second condition, assume there are two operations op_1 and op_2 by process i and $op_1 <_\sigma op_2$. There are four cases:

- Both are writes. Then according to Lemma 6, $ts(op_1) < ts(op_2)$ and op_1 will appear before op_2 in π_i.
- Both are reads. According to Lemma 6, if they read from the same write, they will be arranged after the write and op_1 will be placed before op_2 in π_i; if they read from different write, then $ts(op_1) < ts(op_2)$, thus op_1 will be placed before op_2 in π_i as well.
- op_1 is a write and op_2 is a read. Then according to Lemma 6, $ts(op_1) \leq ts(op_2)$. Thus op_1 will appear before op_2 in π_i by our construction.
- op_1 is a read and op_2 is a write. According to item (c) of Lemma 6, op_2 will appear after op_1.

Thus $\pi_i|i$ is σ-consistent.

For the third condition, since we have already known the order among writes is σ-consistent in π_i, we only need to prove that for a write w, if it precedes r in σ, it will appear before r in π_i. By item (b) of Lemma 6 and by our construction method, this is true.

For the fourth condition, according to algorithm Alg_LC, $ts(w) < ts(w')$. Thus w will appear before w' in π_i for all i by our construction.

Similarly for the last condition, $ts(w_1) < ts(w_2)$ according to algorithm Alg_LC. Therefore, w_1 will appear before w_2 in π_i for all i by our construction.

Therefore, *Alg_LC* implements a CohReg shared variable.

■

Figure 10 shows an execution of algorithm Alg_LC that will generate the schedule shown in Figure 9. The denotation $query(local) = (v, t)$ indicates that we fetch the value-timestamp pair stored in the local cache, and the denotation $local := (v, t)$ indicates that the local cache is updated with new value-timestamp pair (v, t).

F. MWReg+

1. Specifying MWReg+

Informally, MWReg+ strengthens MWReg by placing the following additional constraint on the read operations: any two read operations performed by the same process must be observed in π in the order in which they occur at that process.

This is equivalent to the requirement that once a process reads from a given write, it never reads from an "earlier" write in the order of writes perceived by that process, i.e., individual processes read from writes in nondecreasing order. In [19], variables with this property are called *monotone variables*. Following is the formal definition of MWReg+.

Definition 11 (MWReg+) *A schedule* σ satisfies MWReg+ *if there exists a permutation* π *of* $ops(\sigma)$ *such that, for all read operations* r *in* $ops(\sigma)$*, the projection* π_r *of* π *onto* $writes_{\leftarrow r}(\sigma) \cup \{r\}$ *satisfies:*

- π_r *is legal,*
- π_r *is* σ*-consistent,*

p_0	p_1	p_2	p_3
start $W(x,1)$			
$query(A) = (0,0)$	**start** $W(x,2)$		
$query(B) = (0,0)$	$query(A) = (0,0)$		
$query(local) = (0,0)$	$query(C) = (0,0)$		
$ts := 1$	$query(local) = (0,0)$		
$update(C,(1,1))$	$ts := 1$		
$update(A,(1,1))$	$update(B,(2,1))$		
$local := (1,1)$	$update(C,(2,1))$		
ack	$local := (2,1)$		
	ack		
		start $R_1(x)$	
		$query(A) = (1,1)$	**start** $R_2(x)$
		$query(B) = (2,1)$	$query(B) = (2,1)$
		$query(local) = (0,0)$	$query(C) = (2,1)$
		$local := (1,1)$	$query(local) = (0,0)$
		return 1	$local := (2,1)$
			return 2
		start $R_3(x)$	
		$query(A) = (1,1)$	
		$query(C) = (2,1)$	
		$query(local) = (1,1)$	
		return 1	
	start $R_4(x)$		
			start $W(x,4)$
	$query(B) = (2,1)$		$query(A) = (1,1)$
			$query(B) = (2,1)$
			$query(local) = (2,1)$
			$ts := 2$
			$update(A,(4,2))$
	$query(A) = (4,2)$		
	$query(local) = (2,1)$		
	$local := (4,2)$		
	return 4		
	start $R_5(x)$		
	$query(B) = (2,1)$	**start** $W(x,3)$	
		$query(B) = (2,1)$	
			$update(B,(4,2))$
			$local := (4,2)$
	$query(C) = (2,1)$		**ack**
	$query(local) = (4,2)$		
	return 4	$query(C) = (2,1)$	
		$query(local) = (1,1)$	
		$ts := 2$	
		$update(A,(3,2))$	
		$update(C,(3,2))$	
		$local := (3,2)$	
		ack	
start $R_6(x)$			
$query(B) = (4,2)$			
$query(C) = (3,2)$			
$query(local) = (1,1)$			
$local := (4,2)$			
return 4			

Fig. 10. An execution of Alg_LC that generates the schedule in Figure 9.

- *if r' is a read operation by the same process as r and $r <_\sigma r'$, then for any write w, if w appears before r in π_r, then w appears before r' in π_r.*

A shared memory object satisfies MWReg+ if all schedules on that object satisfy MWReg+.

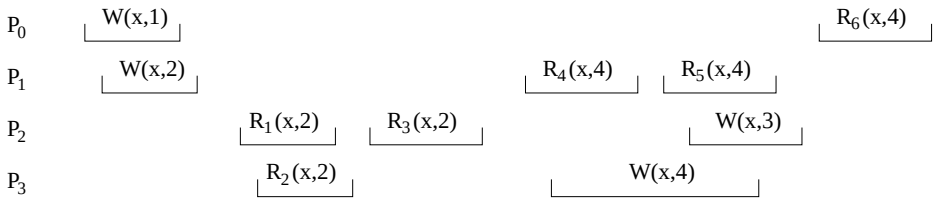

Fig. 11. Schedule that satisfies MWReg+.

Figure 11 shows a schedule that satisfies MWReg+. Compared with the schedule in Figure 5, which satisfies MWReg, the only difference is that read operation R_5 now returns the same value as the read preceding it, which is more recent than $W(x, 2)$. Now the permutation π is $W(x, 1)$, $W(x, 2)$, $R_1(x, 2)$, $R_2(x, 2)$, $R_3(x, 2)$, $W(x, 3)$, $W(x, 4)$, $R_4(x, 4)$, $R_5(x, 4)$, $R_6(x, 4)$,and the projection for the read operations are:

$$
\begin{array}{ll}
R_1\text{:} & W(x,1), W(x,2), R_1(x,2) \\
R_2\text{:} & W(x,1), W(x,2), R_2(x,2) \\
R_3\text{:} & W(x,1), W(x,2), R_3(x,2) \\
R_4\text{:} & W(x,1), W(x,2), W(x,4), R_4(x,4) \\
R_5\text{:} & W(x,1), W(x,2), W(x,3), W(x,4), R_5(x,4) \\
R_6\text{:} & W(x,1), W(x,2), W(x,3), W(x,4), R_6(x,4)
\end{array}
$$

It can be easily verified that all the projections satisfy the first two bullets of MWReg+. In addition, for the two read operations of p_1 (or p_2), their projections also satisfy the third bullet.

2. Implementing MWReg+

We name the implementation algorithm *Alg_ID_LC*. It uses both the first and the third building blocks in Section A. Following is the proof of the correctness of the algorithm.

Theorem 5 *Algorithm Alg_ID_LC implements MWReg+.*

Proof. We construct π as in the proof of Theorem 2, except that read operations with identical timestamps are ordered consistently with their partial order in σ. Now we prove that π satisfies the conditions in Definition 11.

The first two conditions can be proved using the same arguments as in the proof of Theorem 2.

As for the third condition, consider two read operations r_1 and r_2 of the same process, where r_1 completes before r_2 begins. Because π_{r_1} and π_{r_2} are projected from the same sequence π, it is sufficient to prove that (1) r_1 appears before r_2 in π, and (2) all writes that appear in π_{r_1} also appear in π_{r_2}.

The first claim follows from the fact that, by *Alg_ID_LC*, the timestamp of r_2 is at least that of r_1, so our construction method places them in π in the order indicated. The second claim follows from the fact that $writes_{\leftarrow r_1} \subseteq writes_{\leftarrow r_2}$, which is clear by definition of $writes_{\leftarrow r}$ (see Section C). Thus all writes that appear in π_{r_1} also appear in π_{r_2}. ■

Figure 12 shows an execution of algorithm Alg_ID_LC that will generate the schedule shown in Figure 11.

p_0	p_1	p_2	p_3
start $W(x,1)$			
$query(A) = (0,[0,0])$	**start** $W(x,2)$		
$query(B) = (0,[0,0])$	$query(A) = (0,[0,0])$		
$query(local) = (0,[0,0])$	$query(C) = (0,[0,0])$		
$ts := [1,0]$	$query(local) = (0,[0,0])$		
$update(C,(1,[1,0]))$	$ts := [1,1]$		
$update(A,(1,[1,0]))$	$update(B,(2,[1,1]))$		
$local := (1,[1,0])$	$update(C,(2,[1,1]))$		
ack	$local := (2,[1,1])$		
	ack		
		start $R_1(x)$	
		$query(A) = (1,[1,0])$	**start** $R_2(x)$
		$query(B) = (2,[1,1])$	$query(B) = (2,[1,1])$
		$query(local) = (0,[0,0])$	$query(C) = (2,[1,1])$
		$local := (2,[1,1])$	$query(local) = (0,[0,0])$
		return 2	$local := (2,[1,1])$
			return 2
		start $R_3(x)$	
		$query(A) = (1,[1,0])$	
		$query(C) = (2,[1,1])$	
		$query(local) = (2,[1,1])$	
		return 2	
	start $R_4(x)$		
			start $W(x,4)$
	$query(B) = (2,[1,1])$		$query(A) = (1,[1,0])$
			$query(B) = (2,[1,1])$
			$query(local) = (2,[1,1])$
			$ts := [2,3]$
			$update(A,(4,[2,3]))$
	$query(A) = (4,[2,3])$		
	$query(local) = (2,[1,1])$		
	$local := (4,[2,3])$		
	return 4		
	start $R_5(x)$		
	$query(B) = (2,[1,1])$	**start** $W(x,3)$	
		$query(B) = (2,[1,1])$	
			$update(B,(4,[2,3]))$
			$local := (4,[2,3])$
	$query(C) = (2,[1,1])$		**ack**
	$query(local) = (4,[2,3])$		
	return 4	$query(C) = (2,[1,1])$	
		$query(local) = (2,[1,1])$	
		$ts := [2,2]$	
		$update(A,(3,[2,2]))$	
		$update(C,(3,[2,2]))$	
		$local := (3,[2,2])$	
		ack	
start $R_6(x)$			
$query(B) = (4,[2,3])$			
$query(C) = (3,[2,2])$			
$query(local) = (1,[1,0])$			
$local := (4,[2,3])$			
return 4			

Fig. 12. An execution of Alg_ID_LC that generates the schedule in Figure 11.

G. PCGLin

1. Specifying PCGLin

PCGLin is a consistency condition that strengthens both MWWeakReg+ and CohReg. In addition to CohReg, it requires the order of writes observed by each process to extend $<_{co}$ instead of just $<_{\sigma}$. Compared to MWWeakReg+, it requires that the reads by the same process observe the write operations in the same order[3]. The formal definition is given below.

Definition 12 (PCGLin) *A schedule* σ satisfies PCGLin *if there exists a reads-from function on* σ *such that for all processes* i*, there exists a permutation* π_i *of* $writes(\sigma) \cup ops(\sigma|i)$ *such that:*

- π_i *is legal,*
- π_i *is co-consistent,*
- *for any read operation* r *and any write operation* w *by process* i*, if* $w <_{\sigma} r$ *and* r *reads from another write* w'*, then* w *appears before* w' *in* π_j *for all* j*,*
- *for any two read operations* r_1 *and* r_2 *by process* i*, if* $r_1 <_{\sigma} r_2$ *and they read from different writes,* w_1 *and* w_2 *respectively, then* w_1 *appears before* w_2 *in* π_j *for all* j*, and*
- *for any two read operations* r *by process* i *and* r' *by process* j*, if* $r <_{\sigma} r'$ *and* r *reads from write operation* w*, then* w *appears before* r' *in* π_j*.*

A shared memory object satisfies PCGLin if all schedules on that object satisfy PCGLin.

---

[3]As is shown later in Section 3 of Chapter IV, PCGLin also strengthens the previously proposed condition PCG by requiring the order of all the writes being observed be co-consistent.

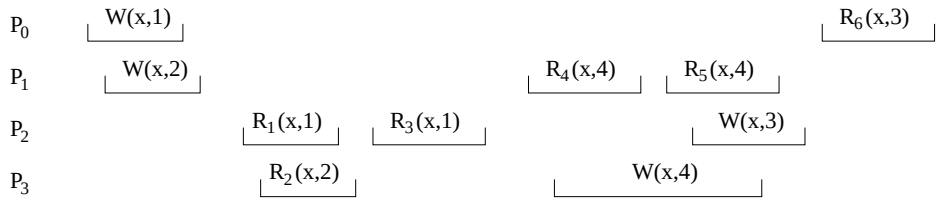

Fig. 13. Schedule that satisfies PCGLin

Figure 13 shows a schedule that satisfies PCGLin. The permutation for each process is listed below, each of which satisfies all the conditions of PCGLin:

p_0: $W(x,1), W(x,2), W(x,4), W(x,3), R_6(x,3)$
p_1: $W(x,1), W(x,2), W(x,4), R_4(x,4), R_5(x,4), W(x,3)$
p_2: $W(x,2), W(x,1), R_1(x,1), R_3(x,1), W(x,4), W(x,3)$
p_3: $W(x,1), W(x,2), R_2(x,2), W(x,4), W(x,3)$

The schedule in Figure 7 does not satisfy PCGLin because for process p_2, there is no way to construct a permutation that is legal and co-consistent. The schedule in Figure 9 does not satisfy PCGLin since p_0's permutation would violate the co-consistency among write operations.

2. Implementing PCGLin

The implementation algorithm, which we call *Alg_WB_LC*, uses the second and the third building blocks. Both Lemma 5 and Lemma 6 will apply to *Alg_WB_LC* as well.

Theorem 6 *Algorithm Alg_WB_LC implements a PCGLin shared variable.*

Proof. For each process i, we construct a permutation π_i of $writes(\sigma) \cup ops(\sigma|i)$ using the following method.

We first order the write operations into a sequence π_i^w according to the increasing order of their timestamp value. Writes with the same timestamp value are arranged arbitrarily with the exception that the write by process i itself is placed as the last

one among those writes in π_w. We then insert the read operations by process i into π_i^w. Each read operation follows the write operation it reads from and precedes the next write in π_i^w. The read operations between any two write operations are arranged in order of the occurrence. We name the final sequence π_i.

First, according to our construction, a read follows the write operation it reads from and there is no other write between them. Therefore, π_i is legal.

For the second condition, we need to prove that for any two operations op_1 and op_2 in $writes(\sigma) \cup ops(\sigma|i)$, if $op_1 <_{co} op_2$, then op_1 appears before op_2 in π_i. From the definition of $<_{co}$, there are two possible cases:

- $op_1 <_\sigma op_2$. There are four different cases:
  - Both are write operations. Then according to Lemma 5, $ts(op_1) < ts(op_2)$. By our construction, op_1 will appears before op_2 in π_i.
  - Both are read operations. According to item (d) of Lemma 6, either $ts(op_1) = ts(op_2)$ or $ts(op_1) < ts(op_2)$. For the first case, op_1 and op_2 will read from the same write and op_2 will be arranged after op_1; for the second case, the write op_1 reads from will appear before the write op_2 reads from, thus op_1 will appear before op_2 as well.
  - op_1 is a write operation and op_2 is a read operation. Then according to item (a) of Lemma 5, $ts(op_1) \leq ts(op_2)$. By our construction, op_1 will appear before op_2 in π_i.
  - op_1 is a read operation and op_2 is a write operation. According to item (c) of Lemma 5, $ts(op_1) < ts(op_2)$. By our construction, op_1 will appear before op_2 in π_i.
- op_2 reads from op_1. According to our construction, op_2 will follow op_1 in π_i.

Overall, π_i is co-consistent.

For the third condition, according to Alg_WB_LC, we have $ts(w) < ts(w')$. Then for the permutation of any process j, w will appear before w' by our construction.

For the fourth condition, we have $ts(w_1) < ts(w_2)$ according to Alg_WB_LC. Thus w_1 will appear before w_2 in the permutation of any process j by our construction.

For the last condition, according to Lemma 5 we have $ts(r) \leq ts(r')$. Since $ts(w) = ts(r)$, thus $ts(w) \leq ts(r')$. By our construction, w will be placed before r' in π_j.

Therefore, Algorithm Alg_WB_LC implements a PCGLin shared avariable.

■

Figure 14 shows an execution of algorithm Alg_WB_LC that will result in the schedule shown in Figure 13.

H. Atomicity

For the completeness of our lattice walk, it is worth mentioning that the remaining two algorithms, one using the first and the second building blocks and the other using all three, implement atomicity. The following theorem shows that Algorithm *Alg_ID_WB*, which uses process ID in the timestamp and includes a write-back phase in the read procedure, implements atomicity.

Theorem 7 *Algorithm Alg_ID_WB implements an atomic shared variable.*

Proof. We construct the permutation π of all the operations in schedule σ as in the proof of Theorem 5 that Alg_ID_LC implements MWReg+. π is legal by the construction. Furthermore, Lemma 5 and our construction ensures that for any two operations op_1 and op_2 in σ, if $op_1 <_\sigma op_2$, then op_1 precedes op_2 in π.

Thus π is legal and σ-consistent and the schedule is atomic. ■

p_0	p_1	p_2	p_3
start $W(x,1)$			
$query(A) = (0,0)$	**start** $W(x,2)$		
$query(B) = (0,0)$	$query(A) = (0,0)$		
$query(local) = (0,0)$	$query(C) = (0,0)$		
$ts := 1$	$query(local) = (0,0)$		
$update(C,(1,1))$	$ts := 1$		
$update(A,(1,1))$	$update(B,(2,1))$		
$local := (1,1)$	$update(C,(2,1))$		
ack	$local := (2,1))$		
	ack		
		start $R_1(x)$	
		$query(A) = (1,1)$	**start** $R_2(x)$
		$query(B) = (2,1)$	$query(B) = (2,1)$
		$query(local) = (0,0)$	$query(C) = (2,1)$
		$update(B,(1,1))$	$query(local) = (0,0)$
		$update(C,(1,1))$	$update(A,(2,1))$
		$local := (1,1)$	$update(B,(2,1))$
		return 1	$local := (2,1))$
			return 2
		start $R_3(x)$	
		$query(A) = (2,1)$	
		$query(C) = (1,1)$	
		$query(local) = (1,1)$	
		$update(A,(1,1))$	
		$update(B,(1,1))$	
		return 1	
	start $R_4(x)$		
			start $W(x,4)$
	$query(B) = (1,1)$		$query(A) = (1,1)$
			$query(B) = (1,1)$
			$query(local) = (2,1)$
			$ts := 2$
			$update(A,(4,2))$
	$query(A) = (4,2)$		
	$query(local) = (2,1)$		
	$update(C,(4,2))$		
	$update(A,(4,2))$		
	$local := (4,2)$		
	return 4		
	start $R_5(x)$		
	$query(B) = (1,1)$	**start** $W(x,3)$	
		$query(B) = (1,1)$	
			$update(B,(4,2))$
			$local := (4,2)$
	$query(C) = (4,2)$		**ack**
	$query(local) = (4,2)$		
	$update(A,(4,2))$	$query(C) = (4,2)$	
		$query(local) = (1,1)$	
	$update(C,(4,2))$	$ts := 3$	
	return 4	$update(A,(3,3))$	
		$update(C,(3,3))$	
		$local := (3,3)$	
		ack	
start $R_6(x)$			
$query(B) = (4,2)$			
$query(C) = (3,3)$			
$query(local) = (1,1)$			
$update(A,(3,3))$			
$update(B,(3,3))$			
$local := (3,3)$			
return 3			

Fig. 14. An execution of Alg_WB_LC that generates the schedule in Figure 13.

The argument in the proof of Theorem 7 also applies to algorithm Alg_ID_WB_LC to prove that it implements an atomic shared variable.

I. Relation to the Original Single-Writer Definition

The following lemma emphasizes the relationship between our proposed definitions and SWReg, the single-writer definition of Lamport.

Lemma 7 *Suppose there is only a single writer. Then the following are true:*

(a) MWWeakReg, MWReg, and MWWeakReg+ are equivalent to SWReg.

(b) CohReg and MWReg+ are strictly stronger than SWReg but remain weaker than atomicity. Furthermore, if there is only a single reader, then CohReg and MWReg+ are equivalent to atomicity.

(c) PCGLin is equivalent to atomicity.

Proof. **(a)** We will prove that each consistency condition is equivalent to SWReg in the presence of a single writer:

- **MWWeakReg.** If there is only a single writer, then Definition 7 is exactly the same as that of SWReg. So MWWeakReg and SWReg are equivalent.
- **MWReg.** The definition of MWReg implies SWReg for a single writer. On the other hand, for any schedule σ that satisfies SWReg, every read has a permutation of all the writes and itself that is legal and σ-consistent. Since there is only one writer, σ-consistency implies that all the writes appear in the same order in each read's permutation. Therefore, we can construct a permutation of all the operations in $ops(\sigma)$ by inserting each read into the total order of writes right after the write it reads from. This permutation satisfies MWReg. Therefore we have SWReg $\subseteq$ MWReg. Thus SWReg = MWReg.

- **MWWeakReg+.** For MWWeakReg+, since the causal order extends $<_\sigma$, the permutation for each read also satisfies the conditions for SWReg. Thus MWWeakReg+ $\subseteq$ SWReg. Now let us show that SWReg $\subseteq$ MWWeakReg+. If a schedule satisfies SWReg, then every read has a permutation of all the writes and itself that is legal and σ-consistent. Since there is only one writer, we have $<_\sigma$=$<_{co}$, which means the permutation of each read above is also co-consistent. Thus the schedule satisfies MWWeakReg+. Therefore, SWReg $\subseteq$ MWWeakReg+.

(b) For any single-writer schedule that satisfies MWReg+, the permutation for each read also satisfies the conditions of SWReg. Thus MWReg+ $\subseteq$ SWReg. Meanwhile, MWReg+ requires that a read operation cannot return something that is "older" than what has been previously observed by the process. SWReg, on the other hand, does not have this restriction. Consider the schedule in Figure 15. This schedule satisfies SWReg. However, since the second read of p_1 returns something earlier than what has been observed by the first read, it does not satisfy MWReg+. Therefore, MWReg+ is strictly stronger than SWReg.

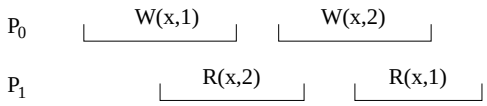

Fig. 15. Single-writer schedule that satisfies SWReg but neither MWReg+ nor CohReg.

Similarly, the first and third conditions of CohReg imply the two conditions of SWReg, thus CohReg $\subseteq$ SWReg. On the other hand, the schedule shown in Figure 15 satisfies SWReg. However, it is impossible to construct a permutation of all the operations that satisfies all the conditions of CohReg. Thus CohReg is strictly

stronger than SWReg.

Now consider the schedule shown in Figure 16. This schedule satisfies both MWReg+ and CohReg. However it is not atomic since we cannot create a sequence of all the operations that is legal and σ-consistent.

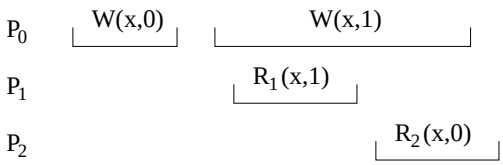

Fig. 16. Single-writer schedule that satisfies MWReg+ and CohReg but not atomicity.

Now we show that if there are only two processes, one being a writer and the other a reader, then MWReg+ = Atomicity. For any schedule that satisfies MWReg+, there is a permutation π of all the operations in $ops(\sigma)$ that satisfies the conditions in Definition 11. π is legal, otherwise we could find some read r such that the projection π_r of π onto $writes_{\leftarrow r}(\sigma) \cup \{r\}$ is not legal. π is also σ-consistent, otherwise there are two reads r_1 and r_2 by the same process, with r_1 preceding r_2 in σ, that appear in π in the reverse order. If so, then the projections will violate the third condition of MWReg+. Therefore, MWReg+ $\subseteq$ Atomicity. On the other hand, since Atomicity is the strongest consistency condition, we have Atomicity $\subseteq$ MWReg+. Thus MWReg+ = Atomicity.

Finally we show that CohReg = Atomicity if there are a single writer and a single reader. Assume a schedule σ satisfies CohReg. The permutation π_{p_r} of the reader process consists of all the operations in σ. From Definition 10, π_{p_r} is legal and σ-consistent. Therefore σ satisfies Atomicity. Thus we have CohReg $\subseteq$ Atomicity. On the other hand, since Atomicity is stronger than any other consistency conditions, we have Atomicity $\subseteq$ CohReg. Therefore, CohReg = Atomicity.

(c) Assume a schedule σ satisfies PCGLin. Since there is a single writer, all the writes appear in the same order in π_i for all i. Denote this total order as π_w. π_w is σ-consistent. Now we insert all the read operations into π_w. Each read immediately follows the write it reads from and if there are multiple reads that follow the same write, arrange them in an order extending $<_\sigma$. We name the resulting sequence π. For each process i, $\pi|writes(\sigma) \cup ops(\sigma|i)$ is a permutation that satisfies PCGLin. π is legal by our construction. Now we show π is σ-consistent. Consider any two operations op_1 and op_2 such that $op_1 <_\sigma op_2$.

- If both are write operations, op_1 will appear before op_2 since π_w is σ-consistent.
- If both are read operations, let us assume that op_1 reads from write operation w. According to the last condition of PCGLin, w appears before op_2 in π_j where j is the process that performs op_2. Then by our construction op_2 will be placed after op_1 in π.
- If op_1 is a write and op_2 is a read, then op_1 will have to appear before op_2. Otherwise, the projection of $\pi|writes(\sigma) \cup ops(\sigma|i)$ will violate PCGLin.
- If op_1 is a read and op_2 is a write, then by the same argument as the previous bullet, op_1 will appear before op_2.

Thus π is σ-consistent and we have $\sigma \in$ Atomicity. Therefore PCGLin $\subseteq$ Atomicity. On the other hand, since Atomicity is stronger than any other consistency conditions, Atomicity $\subseteq$ PCGLin. Thus we have PCGLin = Atomicity.

■

CHAPTER IV

PROPERTIES OF THE DEFINITIONS

So far, all our proposed definitions and their implementation algorithms have focused on the single-variable context. In this chapter, we will extend to the multi-variable scenarios, in which a shared memory system consists of multiple read/write shared variables. For each of our proposed consistency conditions, we first state the definition for the multi-variable situation, and then prove that each one has the desirable property of being "local". For most of the definitions, the multi-variable version is actually the same as the single-variable version; in a couple of cases, we clarify that certain aspects of the definition apply only on a per-variable basis.

We then compare the relative strength among our definitions, as well as with some well-known consistency conditions already appearing in the literature.

A. Locality

We say a consistency condition C is *local* if a schedule σ satisfies C if and only if the projection of σ on each shared variable satisfies C. Locality is a desirable property of consistency conditions: as mentioned in [12], locality enhances modularity and concurrency.

In our case, if all our proposed definitions are proved to be local, which implies that the shared variables can be implemented independently of each other, then our implementation algorithms can be used to implement a multi-variable shared memory with the same consistency condition without additional costs.

In the following, we first give the multi-variable definition of our proposed consistency conditions and then we prove their locality.

1. Locality of MWWeakReg

Definition 13 (Multi-Variable MWWeakReg) *A schedule* σ satisfies MWWeakReg *if, for all read operations* r *in* $ops(\sigma)$*, there exists a permutation* π_r *of* $writes(\sigma) \cup \{r\}$ *such that:*

- π_r *is legal, and*
- π_r *is* σ*-consistent.*

A shared memory object satisfies MWWeakReg if all schedules on that object satisfy MWWeakReg.

Theorem 8 *MWWeakReg is local.*

Proof. We need to show that for any schedule σ on a set of shared variables, σ satisfies MWWeakReg iff $\sigma|x$ satisfies MWWeakReg for every shared variable x. Since the "only if" part in the definition of locality is obvious, we will only focus on the "if" part. (Similar approaches apply to the other locality proofs later in this section.) In other words, we want to show that we can construct for each read operation r, a permutation π_r of all the writes in $ops(\sigma)$ and r (not just the writes in $ops(\sigma|x)$) such that π_r is legal and σ-consistent.

Consider any read operation r in $ops(\sigma)$. Suppose the read operation is on shared variable x. That $\sigma|x$ satisfies MWWeakReg implies that there is a permutation π'_r of $writes(\sigma|x) \cup \{r\}$ that is legal and σ-consistent. For all the other writes, which are not on x, we insert them into π'_r according to the σ-consistent order $<_\sigma$. Since the final sequence is an extension from the partial order of $<_\sigma$, thus this sequence is σ-consistent. Furthermore, as we did not change the order of operations in π'_r and there are no reads from variables other than x, the final sequence is legal. ∎

2. Locality of MWReg

Definition 14 (Multi-Variable MWReg) *A schedule* σ satisfies MWReg *if there exists a permutation* π *of* $ops(\sigma)$ *such that, for all read operations* r *in* $ops(\sigma)$*, the projection* π_r *of* π *onto* $writes_{\leftarrow r}(\sigma) \cup \{r\}$ *satisfies:*

- π_r *is legal, and*
- π_r *is* σ*-consistent.*[1]

A shared memory object satisfies MWReg if all schedules on that object satisfy MWReg.

To prove the locality of MWReg (as well as MWWeakReg+, MWReg+ and PCGLin later in this chapter), we follow the example of proving the locality of linearizability in [12]. The only difference between the proofs is the way to define a partial order, which will be extended into a total order to construct the permutation.

Theorem 9 *MWReg is local.*

Proof. Let σ be a schedule on a set of shared variables such that $\sigma|x$ satisfies MWReg for each shared variable x. Let π^x be the permutation of $ops(\sigma|x)$ that witnesses MWReg for $\sigma|x$, and denote the total order derived from π^x as $<_x$. Note that $<_x$ orders all writes consistently with σ, and, for any individual read, orders that read consistently with all writes, but may switch the relative order of reads.

Now we define a new relation $<_{po}$ according to the following rules:

1. If $op_1 <_x op_2$ for any shared variable x, then $op_1 <_{po} op_2$.

[1] Note that if there are only a finite number of reads in a given execution, the writes after the last read are not constrained by MWReg to appear in any particular order. We consider this to be acceptable, as such writes are never observed.

2. If op_1 and op_2 are on different variables, op_1 is a write operation, and $op_1 <_\sigma op_2$, then $op_1 <_{po} op_2$.

3. If $op_1 <_{po} op_2$ and $op_2 <_{po} op_3$, then $op_1 <_{po} op_3$.

Basically, $<_{po}$ extends the union of the $<_x$ relations to include ordering constraints involving two operations on different variables, when the first one is a write.

Suppose $<_{po}$ is a partial order. Then we can choose any permutation π of all the operations in $ops(\sigma)$ that extends $<_{po}$. It is easy to verify that for each read r, the projection of π onto $writes_{\leftarrow r}(\sigma) \cup \{r\}$ is legal and σ-consistent. (Legality for shared variable x is ensured by $<_x$, and σ-consistency is ensured by $<_x$ and by condition 2.)

Now we prove that $<_{po}$ is a partial order. Suppose in contradiction it is not. Let C be a shortest cycle in $<_{po}$. C must involve more than one variable, since each relation $<_x$ is a partial order and has no cycles.

Claim: C consists of at most one operation on any given variable. Why? Suppose in contradiction there are at least two operations on some shared variable x in C. Since there is at least one other variable occurring in C, we can represent the operations in C as the sequence

$$op_1, s_1, s_2, \ldots, s_k, op_i, \ldots op_m,$$

where

- op_1 and op_i are both on variable x,
- each s_j is a sequence of operations on some variable $y_j \neq x$, where $y_j \neq y_{j+1}$,
- each operation in the entire sequence is $<_{po}$ the next operation in the sequence, and
- $op_m <_{po} op_1$.

The edges between consecutive operations in each s_j are due to $<_{y_j}$. The edges from op_1 to the first operation in s_1, from the last operation in s_j to the first operation in s_{j+1}, and from the last operation in s_k to op_i are due to $<_\sigma$. Thus op_1 and the last operation in each s_j are all writes.

For each s_j, the first operation in s_j starts before the last operation in s_j (a write) ends, since otherwise π^{y_j}, on which $<_{y_j}$ is based, would violate σ-consistency for those two operations, one of which is a write.

Thus op_1 ends before op_i begins. Since op_1 is a write and op_1 and op_i are both operations on variable x, $op_1 <_x op_i$.

Thus we can get a shorter cycle than C by skipping directly from op_1 to op_i, a contradiction.

(End of Claim)

Thus every operation in C involves a different variable, and every edge in C is due to $<_\sigma$. But it is not possible for $<_\sigma$ to have a cycle, a contradiction.

■

3. Locality of MWWeakReg+

Definition 15 (Multi-Variable MWWeakReg+) *A schedule σ* satisfies MWWeakReg+ *if there exists a reads-from function on σ such that for all read operations r in $ops(\sigma)$, there exists a permutation π_r of $writes(\sigma) \cup \{r\}$ such that:*

- *π_r is legal, and*
- *π_r is co-consistent.*

A shared memory object satisfies MWWeakReg+ if all schedules on that object satisfy MWWeakReg+.

Theorem 10 *MWWeakReg+ is local.*

Proof. Consider a schedule σ and a reads-from function for σ such that, for each shared variable x, $\sigma|x$ satisfies MWWeakReg+. In other words, for each x, the following holds: For each read r in $\sigma|x$, there exists a permutation π_r^x of writes$(\sigma|x) \cup \{r\}$ that is legal and co-consistent (note that co-consistency here refers just to the operations in ops$(\sigma|x)$.)

We must show that the original schedule σ satisfies MWWeakReg+. That is, we must show that for each read r in σ, there exists a permutation π_r of writes$(\sigma) \cup \{r\}$ that is legal and co-consistent (with respect to all the operations).

Consider an arbitrary read r in σ. Let x be the variable that r reads. We will construct the desired permutation π_r of writes$(\sigma) \cup \{r\}$ by first constructing a partial order and then letting π_r be any total order that is consistent with the partial order. The desired properties of π_r will follow from properties of the partial order.

Recall that for each shared variable x', the assumed MWWeakReg+ property of $\sigma|x'$ gives us a permutation of all the writes on x'. Informally, the relation will consist of interleaving these permutations in a particular way that respects real-time ordering and reads-from relationships. More formally, for variable x (the one read by the chosen read r), consider the permutation π_r^x, and for each variable $x' \neq x$, choose an arbitrary read r' on x' and consider the permutation $\pi_{r'}^{x'}$. (If there is no read on x', then simply order the write operations into a sequence that extends $<_\sigma$.) Let $<_{x'}$ denote the total ordering induced by $\pi_{r'}^{x'}$.

Let $<_{po}$ be the relation on ops$(\sigma) \cup \{r\}$ defined as follows:

- if $op_1 <_{x'} op_2$, then $op_1 <_{po} op_2$;
- if $op_1 <_{co} op_2$, then $op_1 <_{po} op_2$; and
- if $op_1 <_{po} op_2$ and $op_2 <_{po} op_3$ then $op_1 <_{po} op_3$.

We now show that $<_{po}$ is a partial order. Suppose in contradiction it is not, and let

$C = op_0, op_2, \ldots, op_{m-1}, op_0$ be a shortest cycle in $<_{po}$.

Observation 1: Since C is shortest, it has the form of alternating $<_{x'}$ and $<_{co}$ edges.

Observation 2: Also, for any two operations op_i and $op_{(i+1) \bmod m}$, if there is a $<_{x'}$ edge between them for some variable x', then $op_i \not<_\sigma op_{(i+1) \bmod m}$. Otherwise, we have $op_i <_\sigma op_{(i+1) \bmod m} <_\sigma op_{(i+2) \bmod m}$ or $op_i <_\sigma op_{(i+1) \bmod m} \not<_\sigma r' <_\sigma op_{(i+2) \bmod m}$, where r' reads from $op_{(i+1) \bmod m}$. Either case implies $op_i <_{co} op_{(i+2) \bmod m}$, which yields a smaller cycle $op_0, ..., op_i, op_{(i+2) \bmod m}, ..., op_{m-1}, op_0$. Contradiction.

Case 1: C consists of all writes, i.e., $C = w_0, w_1, \ldots, w_{m-1}, w_0$. Without loss of generality, assume that $w_1 <_{x'} w_2$ for some variable x'. By the observations above, we have

$$w_0 \not<_\sigma w_1 \quad \begin{matrix} \not<_\sigma r <_\sigma \\ or \\ <_\sigma \end{matrix} \quad w_2 \not<_\sigma w_3 \ldots w_{m-2} \not<_\sigma w_{m-1} \quad \begin{matrix} \not<_\sigma r <_\sigma \\ or \\ <_\sigma \end{matrix} \quad w_0$$

As a result, either $w_1 <_\sigma w_0$ or $w_1 <_{co} w_0$. Either case contradicts the assumption that $w_0 <_{x'} w_1$.

Case 2: C contains the unique read, r.

Case 2.1: r is at the head of a $<_x$ edge in C. Then C can be written as $r, w_1, w_2, \ldots, w_{m-1}$, where

- $r <_x w_1$,
- $w_{2i-1} <_{co} w_{2i}$,
- $w_{2i} <_{x_i} w_{2i+1}$ for some variable x_i, and
- $w_{m-1} <_{co} r$.

By a similar argument as in Case 1, we have:

$$\not<_\sigma\ r <_\sigma \qquad\qquad\qquad\qquad \not<_\sigma\ r <_\sigma$$
$$r \not<_\sigma w_1 \quad or \quad w_2 \not<_\sigma w_3 \ldots w_{m-2} \not<_\sigma w_{m-1} \quad or \quad w_1$$
$$<_\sigma \qquad\qquad\qquad\qquad <_\sigma$$

This implies either $w_1 <_\sigma r$ or $w_1 <_{co} r$, contradicting the fact that $r <_{x'} w_1$.

Case 2.2: r is at the head of a $<_{co}$ edge in C. A similar argument as in Case 2.1 gives a contradiction here as well.

Now we know that $<_{po}$ is a partial order. Let π_r be any total order on writes$(\sigma)U\{r\}$ that is consistent with $<_{po}$.

This permutation is legal, since it extends π_r^x, which is legal, and no further operations on x are added. This permutation is co-consistent because $<_{po}$ is.

■

4. Locality of CohReg

For CohReg, we say a multi-variable shared memory satisfies CohReg if CohReg is satisfied on per-variable basis. Thus the locality is inherent in its definition (just like the locality of Coherence ([10])).

5. Locality of MWReg+

To extend MWReg+ into multi-variable context, we need to clarify that the third bullet in Definition 11 will only apply on a per-variable basis. The formal definition is given below.

Definition 16 (Multi-Variable MWReg+) *A schedule* σ satisfies MWReg+ *if there exists a permutation* π *of* $ops(\sigma)$ *such that, for all read operations* r *in* $ops(\sigma)$*, the projection* π_r *of* π *onto* $writes_{\leftarrow r}(\sigma) \cup \{r\}$ *satisfies:*

- π_r *is legal,*
- π_r *is* σ*-consistent, and*
- *if* r' *is a read operation by the same process and on the same variable as* r *and* r' *is performed after* r*, then for any write* w*, if* w *appears before* r *in* π_r*,* w *appears before* r' *in* $\pi_{r'}$*.*

A multi-variable shared memory system satisfies MWReg+ if all schedules on the shared memory satisfy MWReg+.

Theorem 11 *MWReg+ is local.*

Proof. We use the same partial order $<_{po}$ as in Theorem 9 and we then extend this partial order into a total order to construct a permutation π of all the operations in $ops(\sigma)$.

The same argument in Theorem 9 applies here to prove that $<_{po}$ is indeed a partial order and that π satisfies the first two conditions of MWReg+.

Now the only thing left is to prove that the third condition of MWReg+ is also satisfied. Suppose there are two read operations r_1 and r_2 on the same variable and they are performed by the same process, with r_1 being performed before r_2, we need to show that for any two write operations w_1 and w_2, if w_1 appears before w_2 in π_{r_1}, then w_1 appears before w_2 in π_{r_2}. Assume that r_1 and r_2 are on variable x, then $r_1 <_x r_2$. Otherwise, MWReg+ on x will be violated. Since we will not change the relative order of the two reads in π, therefore, any two write operations that are in the projection of r_1 will appear in the projection of r_2 as well, and since the projection will not alter the relative order of the operations, the two writes should be in the same order in both projections.

■

6. Locality of PCGLin

When extending PCGLin to multi-variable version, the last three conditions remain on a per-variable basis.

Definition 17 (PCGLin) *A schedule* σ satisfies PCGLin *if there exists a reads-from function on* σ *such that for all processes* i*, there exists a permutation* π_i *of* $writes(\sigma) \cup ops(\sigma|i)$ *such that:*

- π_i *is legal,*
- π_i *is co-consistent,*
- *for a read operation* r *and a write operation* w *on shared variable* x *by process* i*, if* $w <_\sigma r$ *and* r *reads from another write* w'*, then* w *appears before* w' *in* π_i *for all* i*,*
- *for two read operations* r_1 *and* r_2 *on shared variable* x*, by process* i*, if* $r_1 <_\sigma r_2$ *and they read from different writes,* w_1 *and* w_2 *respectively, then* w_1 *appears before* w_2 *in* π_j *for all* j*, and*
- *for two read operations* r *by process* i *and* r' *by process* j *on the same shared variable* x*, if* $r <_\sigma r'$ *and* r *reads from write operation* w*, then* w *appears before* r' *in* π_j*.*

A shared memory object satisfies PCGLin if all schedules on that object satisfy PCGLin.

Theorem 12 *PCGLin is local.*

We skip the proof for Theorem 12 since it is very similar to that of Theorem 10. Moreover, both proofs use the same partial order definition.

B. Comparison

In this section, we first compare the strength among our proposed consistency conditions; we then explore the conjunction relationship among them; we finally compare our definitions with other existing consistency conditions.

1. Comparison Between Proposed Definitions

Theorem 13 *For any two local consistency conditions C_s and C_w, if C_s is stronger than C_w on a single shared variable, then C_s is stronger than C_w.*

Proof. We first need to prove that given any schedule $\sigma \in C_s$, σ also belongs to C_w. Given any shared variable x in σ, $\sigma|x \in C_s$. Since C_s is stronger than C_w on a single shared variable, we thus have $\sigma|x \in C_w$. By the locality property of C_w, $\sigma \in C_w$.

Now we need to prove that there exist a schedule that is in C_w but is not in C_s. Since C_s is stronger than C_w on a single shared variable, say x, there is a schedule σ_x that is in C_w but not in C_s. ■

A direct application of Theorem 13 is that the comparison among our proposed definitions can be done on a single variable basis since all of them are local. In the following proofs, all the schedules we discuss access only a single shared variable.

Lemma 8 *MWReg+ $\subset$ MWReg $\subset$ MWWeakReg.*

Proof. The definition of MWReg+ is identical to the definition of MWReg except for the imposition of an additional constraint. The fact that this constraint strengthens the definition can be seen from the fact that the schedule in Figure 5 satisfies MWReg but not MWReg+. Therefore MWReg+ is stronger than MWReg.

MWReg can be seen to imply MWWeakReg because the projection π_r for each read operation r under MWReg is also a permutation that satisfies the conditions in

the definition of MWWeakReg. As we have already noted, however, the schedule in Figure 3 does not satisfy MWReg. Thus MWReg is stronger than MWWeakReg. ■

Lemma 9 *MWReg+ ⊂ CohReg*

Proof. For a schedule σ, let π be a permutation of $ops(\sigma)$ that validates MWReg+. The write operations in π preserve the partial order of writes in σ. Otherwise, there would be at least one projection π_r that did not satisfy the second condition of MWReg+.

For each process i, we project π onto $writes(\sigma) \cup ops(\sigma|i)$ and we name the resulting subsequence π_i. It is easy to verify that π_i satisfies the first and third conditions of CohReg. The third condition of MWReg+ implies that all the read operations from the same process appear in π in the same order as in σ. Therefore, $\pi|i$ is σ-consistent. For the fourth condition of CohReg, assume there is a write w and a read r by process i and r reads from another write w'. If $w <_\sigma r$, the operations would appear in π as $..., w, ..., w', ..., r....$ Thus w will appear before w' for all π_i's. If $r <_\sigma w$, then the operations will appear in π as $..., w'..., r, ..., w, ...$ Therefore, w appears after w' for all π_i's. For the last condition, the operations would appear in π as $..., w_1, ..., r_1, ..., w_2, ..., r_2, ...$ Thus w_1 will appear before w_2 in all π_i's. So we have MWReg+ $\subseteq$ CohReg. To see why MWReg+ is stronger than CohReg, consider the example shown in Figure 9, in which the schedule satisfies CohReg but not MWReg+.

■

Lemma 10 *PCGLin ⊂ MWWeakReg+ ⊂ MWWeakReg.*

Proof. It is easy to verify from their definitions that for any schedule $\sigma \in$ MWWeakReg+, $\sigma \in$ MWWeakReg. On the other hand, there exists some schedule that belongs to MWWeakReg but does not satisfy MWWeakReg+. Figure 3 shows such an example.

The definition of PCGLin implies MWWeakReg+ since for any read operation r performed by process i, we can obtain the sequence π_r that satisfies MWWeakReg+ by projecting the sequence π_i that satisfies PCGLin to $writes(\sigma) \cup \{r\}$. On the other hand, Figure 7 shows a schedule that satisfies MWWeakReg+ but not PCGLin. Thus PCGLin $\subset$ MWWeakReg+. ■

Lemma 11 *PCGLin $\subset$ CohReg $\subset$ MWWeakReg.*

Proof. For any read operation r, assume it is performed by process i, we obtain a subsequence π_r^0 by projecting π_i onto $writes(\sigma) \cup \{r\}$, where π_i is the subsequence that satisfies all the conditions of CohReg. π_r^0 is legal since π_i is legal. According to the definition of CohReg, if π_r^0 is not σ-consistent, it must be that there exists a write w such that $r <_\sigma w$ while w appears before r in π_r^0. Assume r reads from write w_r. The operations will appear in π_r^0 as $..., w, ..., w_r, r, ...$ w overlaps all the writes between w and w_r (including w_r) in σ. Otherwise, if w precedes some write, then it is impossible that $r <_\sigma w$; if w succeeds some write, then π_i will violate the third condition of CohReg for some read operation. Thus we can move w after r in π_r^0 and we name the resulting sequence π_r. π_r satisfies both conditions of MWWeakReg.

Therefore, CohReg $\subseteq$ MWWeakReg. Figure 3 shows a schedule that satisfies MWWeakReg. However, it does not meet all the requirement of CohReg. Thus CohReg $\subset$ MWWeakReg.

PCGLin implies CohReg according to their definitions. On the other hand, Figure 9 gives a schedule that is CohReg but does not satisfy PCGLin. Thus PCGLin $\subset$ CohReg.

■

2. Conjunctions

We discover the following conjunction relationships among our proposed definitions.

Lemma 12 *MWReg+ = MWReg ∩ CohReg.*

Proof. From Lemma 8 and Lemma 9, we know that MWReg+ ⊆ MWReg ∩ CohReg. Now we prove that MWReg ∩ CohReg ⊆ MWReg+. We need to show that given any schedule σ that satisfies both MWReg and CohReg, we can obtain a sequence π of $ops(\sigma)$ that satisfies MWReg+.

Since $\sigma \in$ MWReg, we can obtain a sequence π of $ops(\sigma)$ such that π satisfies the two constraints of MWReg. Assume π does not satisfy MWReg+, then there must exist two read operations r and r' by the same process p_i such that r gets performed first, but r appears after r' in π. r and r' read from different write operations, otherwise, π is illegal. Assume r reads from w and r' reads from w'. w' precedes w in π. On the other hand, since schedule σ also satisfies CohReg, w will be observed as preceding w' by all read operations in σ. Assume here is a read operation r'' that starts after both w and w' finish, then r'' will read from w' and they will appear in π as $..., w', r', ...w..$, which will violate MWReg. Contradiction. Thus π satisfies MWReg+ and $\sigma \in$ MWReg+.

Therefore we have MWReg+ = MWReg ∩ CohReg.

■

The following lemma shows PCGLin is the conjunction of MWWeakReg+ and CohReg.

Lemma 13 *PCGLin = MWWeakReg+ ∩ CohReg.*

Proof. From Lemma 10 and Lemma 11, we have PCGLin ⊆ MWWeakReg+ ∩ CohReg.

On the other hand, assume a schedule σ satisfies both MWWeakReg+ and CohReg. We now prove that it also satisfies PCGLin.

For each process p_i, we start from the sequence that satisfies CohReg and we name it π_i^0. Assume there are two operations op_1 and op_2 such that $op_1 <_{co} op_2$ but op_1 appears after op_2 in π_i^0. According to the definition of $<_{co}$ and the definition of CohReg, this could only happen when op_1 is a read and op_2 is a write. Pick the pair with the smallest distance in π_i^0. Assume op_1 reads from w, then they would appear in π_i^0 as ...,op_2,...,w,op_1,other reads that read from w,....

First, op_2 overlaps w and all the writes between op_2 and w in σ. Otherwise, if $op_2 <_\sigma w$, then it is impossible that $op_1 <_\sigma op_2$; if $w <_\sigma op_2$, π_i^0 will not satisfy σ-consistency for some read after op_2.

Second, there should be no reads that read from op_2. Assume not, then pick one from those reads. If the read occurs before op_1 in σ, then it reads from a future write. It is impossible; if the read occurs after op_1, then π_i^0 violates the per-process order. Contradiction.

Third, op_2 overlaps all the read operations between op_2 and w. Assume not, then pick one read r'. If $op_2 <_\sigma r'$, then we have $r <_\sigma op_2 <_\sigma r'$, which means π_i^0 violates the per-process order. If $r' <_\sigma op_2$, then op_2 and op_1 is not the closest pair anymore. Both cases lead to a contradiction.

Fourth, for other reads that read from w, op_2 either overlaps or follows them in σ. Assume not, then suppose op_2 proceeds r' and r' reads from w. Since $w <_{co} op_2$ and $op_2 <_{co} r'$, then there is no way to construct a permutation for r' that satisfies MWWeakReg+. Contradiction.

Therefore, we can move op_2 right after all the reads that read from w and the resulting sequence still satisfies CohReg. We continue on with other pairs that violate co-consistency and we arrange their order the same way as above. We name the final

sequence π_i. π_i satisfies all the three conditions of PCGLin. Therefore, $\sigma \in$ PCGLin. Thus MWWeakReg+ $\cap$ CohReg $\subseteq$ PCGLin.

Overall we have PCGLin = MWWeakReg+ $\cap$ CohReg.

■

The last conjunction relationship is given in the next Lemma.

Lemma 14 *MWReg* $\cap$ *MWWeakReg+* = *Atomicity.*

Proof. We know that Atomicity $\subseteq$ MWReg $\cap$ MWWeakReg+ for sure. So we only need to prove that for any schedule σ, if $\sigma \in$ MWReg $\cap$ MWWeakReg+, then $\sigma \in$ Atomicity.

According to the definition of MWReg, we can obtain a sequence π of $ops(\sigma)$ such that for each read operations r, the projection of π onto $writes_{\leftarrow r}(\sigma) \cup \{r\}$ is legal and σ-consistent.

If the above statement is valid for the projection on $writes(\sigma) \cup \{r\}$, then it implies that π itself is legal and σ-consistent, which means $\pi \in$ Atomicity. So we will have to prove that for each read operation r, the projection of π onto $writes_{\rightarrow r}(\sigma) \cup \{r\}$ is σ-consistent, where $writes_{\rightarrow r}(\sigma) = \{w | w \in writes(\sigma)$ and w begins after r ends in $\sigma\}$. Assume not. Then there are two possible cases: (1) there are two writes w_1 and w_2 such that $w_1 <_\sigma w_2$ and w_2 proceeds w_1 in π. This is not possible since for any read operation r' who starts after w_2 finishes, the projection $\pi_{r'}$ will violate MWReg. (2) it is read r and a write w that follows r in σ and w proceeds r in π. Suppose r reads from write operation w'. Then the three operations would appear in π as $w, ..., w', r, ...$ On the other hand, according to *co*-consistency, $w' <_{co} w$. We say we can move w after r and all the reads that read from w'. Otherwise, there must exist some read operation such that w has to appear before w' in its permutation, which

violates MWWeakReg+. Thus we can re-order sequence π such that the projection for each read is legal and σ-consistent. Therefore, $\sigma \in$ Atomicity.

■

The overall conjunction relationship can be illustrated using an Venn diagram, as is shown in Figure 17.

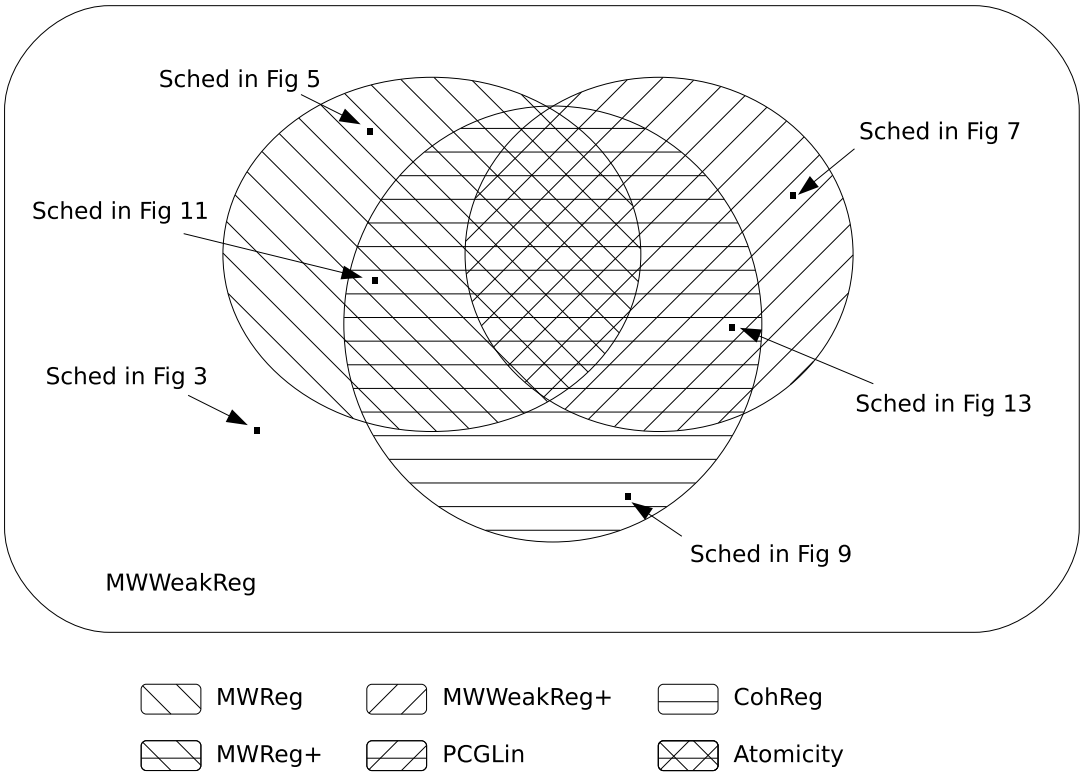

Fig. 17. Venn diagram of the proposed definitions

3. Comparison with Existing Consistency Conditions

Lemma 15 *CohReg* $\subset$ *Coherence.*

Proof. According to the definition of CohReg, for each process i, there is a permutation π_i of $writes(\sigma) \cup ops(\sigma|i)$ that satisfies all the conditions of CohReg. We now

construct a permutation π of all the operations in $ops(\sigma)$ by inserting into π_0, the permutation of process p_0, all the read operations performed by processes other than p_0. Each read follows the write it reads from and precedes the next write in π_0. If there are multiple reads of the same process that follow the same write, arrange them according to the order of occurrence.

First, π is legal as all the reads follow the write it read from and there is no other writes between a read and the write it reads from.

Second, $\pi|i$ is σ-consistent for each process i. To see this, consider the following possible cases:

- for two write operations w_1 and w_2 by i, if $w_1 <_\sigma w_2$, then w_1 appears before w_2 in π. Otherwise, π_0 will violate the third condition of CohReg;
- for two read operations r_1 and r_2 by i, assume they read from w_1 and w_2 respectively. If $r_1 <_\sigma r_2$, then according to the fifth condition of CohReg, w_1 will appear before w_2 in π_0. Therefore, r_1 will be placed before r_2 in π;
- for a write operation w and a read operation r by i and $w <_\sigma r$, if r reads from w, then w will appear before r in π; otherwise, assume r reads from w', then according to the fourth condition of CohReg, w will appear before w' in π_0, thus r will be placed after w in π;
- for a write operation w and a read operation r by i and $r <_\sigma w$, assume r reads from w'. Then the fourth condition of CohReg requires that w appear after w' in π_0. Therefore, r will be placed before w in π.

Therefore, $\sigma \in$ Coherence and we have CohReg $\subseteq$ Coherence.

Figure 18 gives an example schedule that does not satisfy CohReg but does satisfy Coherence. Therefore CohReg $\subset$ Coherence.

■

Lemma 16 *PCGLin* $\subset$ *PCG.*

Proof. The definition of PCGLin implies PRAM. Furthermore, we have PCGLin $\subset$ CohReg $\subset$ Coherence. Therefore PCGLin $\subset$ PRAM $\cap$ Coherence = PCG. ■

All the consistency conditions given in Chapter III have been proved to be local consistency conditions. On the other hand, sequential consistency, PCG and PRAM are known to be not local. Our proposed conditions, except CohReg and PCGLin, do not have constraints on the order of operations from a process's point of view, which is required by sequential consistency, PCG and PRAM. Thus our proposed definitions of regularity cannot be compared with these consistency conditions in terms of strength, as we now show.

The schedule in Figure 18 is sequentially consistent, and thus satisfies PCG and PRAM. It does not, however, satisfy any of the proposed regularity definitions in Chapter III.

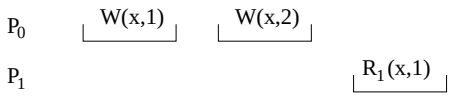

Fig. 18. Schedule that is sequentially consistent but not MWWeakReg.

Now consider the schedule in Figure 19. For variable x, the permutation $W(x,2)$, $W(x,1)$, $R_1(x,1)$, $R_4(x,1)$ is legal. For variable y, the permutation $R_2(y,0)$, $W(y,1)$, $W(y,2)$, $R_3(y,2)$ is legal. Thus the schedule satisfies MWReg+ (as well as MWReg, CohReg and MWWeakReg). However, in order to construct a permutation on p_1's operations and all the other processes' write operations, it is necessary to place $W(y,1)$ after $R(y,0)$ and place $W(y,2)$ before $R(y,2)$. However, the resulting sequence is not

legal because $R(x, 1)$ should return 2. Therefore the schedule is not PRAM and thus it is not PCG or sequentially consistent.

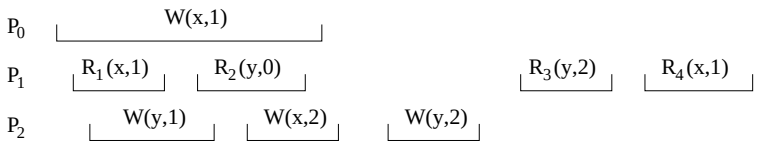

Fig. 19. Schedule that satisfies MWReg+ but not PRAM

The relationships between all the consistency conditions discussed above are shown as a partial order in Figure 20.

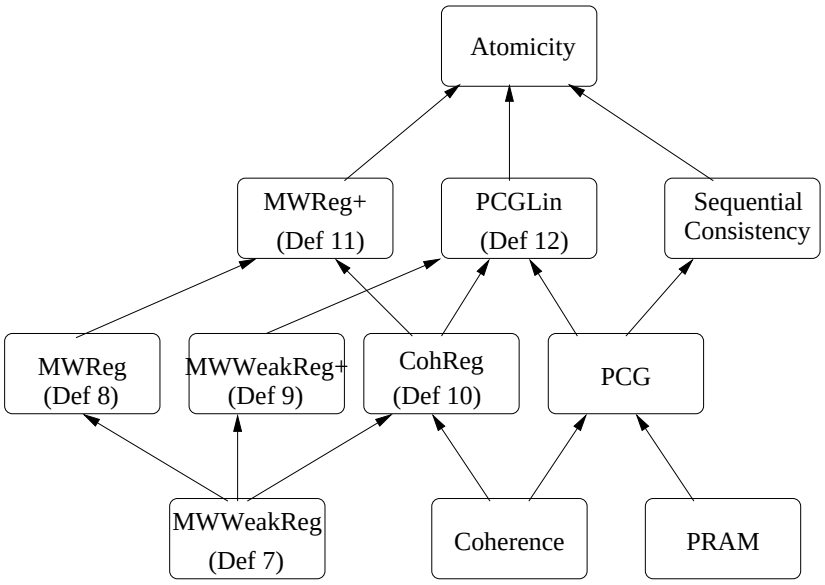

Fig. 20. Partial order among existing consistency conditions

CHAPTER V

MUTUAL EXCLUSION USING MULTI-WRITER REGULAR SHARED VARIABLES

In this chapter, we use the mutual exclusion problem as a practical context to evaluate the strength of our new specifications for multi-writer regular shared variables. Specifically, we study the correctness of two well-known algorithms for mutual exclusion when the variables are implemented according to the consistency conditions we have proposed. The algorithms we examine are Peterson's algorithm for 2 processes ([23]) and Dijkstra's algorithm for n processes ([24]). The algorithms are shown in Figure 21.[1]

Algorithms for solving mutual exclusion are assumed to have four sections: *entry*, *critical*, *exit* and *remainder*. The *critical* section is code that must be protected from concurrent execution. The *entry* section is the code executed in preparation for entering the critical section. The *exit* section is executed to release the critical section. The rest of the code is in the *remainder* section.

A *run of an algorithm* (not to be confused with an execution on a shared object) is defined as an interleaving of local operations and shared-memory operation invocations and responses performed by the participating processes, such that the following are satisfied:

- the projection of the algorithm run onto (the actions performed by) each individual process is consistent with the order of operations imposed by the local algorithm for that process, and

[1] Although Lamport's Bakery algorithm ([14]) and Peterson-Fischer's algorithm ([22]) are often studied in this context, they are not of interest to us here since these algorithms use only single-writer shared variables.

1. Peterson's Algorithm for 2 Processors
Code for process p_i, $i \in \{0, 1\}$:

```
shared variables:
    Flag[0..1] : integer     /* initially 0 */
    Turn : integer     /* initially 0 */

/* entry section */
1    repeat
2        Flag[i] := 0;
3        wait until (Flag[1 − i] = 0 or Turn = i);
4        Flag[i] := 1;
5    until (Turn = i or Flag[1 − i] = 0)

6    if (Turn = i) then wait until (Flag[1 − i] = 0);

Critical Section

/* exit section */
7    Turn := 1 − i;
8    Flag[i] := 0;

Remainder Section
```

2. Dijkstra's Algorithm for n Processors
Code for process p_i, $0 \leq i \leq n-1$:

```
shared variables:
    Flag[0..n − 1] : idle, requesting, in-cs   /* Initially, idle */
    Turn : integer     /* Initially 0 */

/* entry section */
1    repeat
2        Flag[i] := requesting;
3        while (Turn ≠ i) do
4            if (Flag[Turn] = idle) then Turn := i;
5        end while
6        Flag[i] := in-cs;
7    until (∀j ≠ i, Flag[j] ≠ in-cs)

Critical Section

/* exit section */
8    Flag[i] := idle;

Remainder Section
```

Fig. 21. Algorithms for mutual exclusion

- the projection of the algorithm run onto the shared-memory operations on each variable is a schedule on that variable.

(In this context, we consider a shared-memory object "request" to be the invocation of a request by a process , and a shared-memory object "response" to be the receipt of a response by a process. They are thus process actions, but can nevertheless be meaningfully projected onto the object also.) We say that an algorithm *runs under consistency condition* C if its projection onto each shared variable satisfies C.

We say that an algorithm A *solves mutual exclusion under consistency condition* C if, for each run of A under C, the following constraints hold:

- **mutual exclusion (ME):** there is at most one process in the critical section at any point in the execution.

- **eventual progress (EP):**[2] if there is some process waiting to enter the critical

[2]We use this term, rather than the more traditional ND ("no deadlock") in order

Table I. Correctness of mutual exclusion algorithms using multi-writer regular variables.

	Peterson's Algorithm	Dijkstra's Algorithm
MWWeakReg	ME, EP, NL	ME
MWReg	ME, EP, NL	ME, EP
MWWeakReg+	ME, EP, NL	ME
CohReg	ME, EP, NL	ME
MWReg+	ME, EP, NL	ME, EP
PCGLin	ME, EP, NL	ME
Atomicity	ME, EP, NL	ME, EP

section, then eventually some process enters the critical section.

- **no lockout (NL):** if some process is waiting to enter the critical section, then eventually *that* process enters the critical section.[3]

We now examine the two mutual exclusion algorithms shown in Figure 21. Table I shows which of the conditions of mutual exclusion described above are met by each algorithm when implemented with variables satisfying each of our consistency conditions. As a comparison, we also list the conditions that are guaranteed by these algorithms when the shared variables are atomic.

---

to avoid ambiguity: the term "deadlock" sometimes includes "livelock" (in which processes continue taking steps but keep one another trapped in a loop due to timing issues) and sometimes does not. The definition of "eventual progress" explicitly precludes either situation.

[3]Although NL implies EP, we include both requirements, partly for historical reasons (e.g., [13]) but primarily because it gives us a finer gauge of the effectiveness of various consistency conditions, *viz.* Dijkstra's algorithm, which solves EP but not NL under MWReg and MWReg+.

We first consider Peterson's algorithm for two processors ([23]). This algorithm uses two single-writer shared variables and one multi-writer shared variable. The proof of the next theorem is very similar to the proofs of Theorem 4.10 through Theorem 4.12 in [6]. Although [6] assumes that all the variables are atomic, the arguments hold unchanged for variables even with a consistency condition as weak as MWWeakReg, and therefore all the other proposed regularity definitions.

Theorem 14 *Peterson's Algorithm solves mutual exclusion (ME, EP, and NL) under all the proposed definitions of regularity.*

Dijkstra's algorithm for n processors uses n single-writer shared variables and one multi-writer shared variable ([24]). Under both MWReg and MWReg+ it behaves the same way as under atomicity: ME and EP are guaranteed, but not NL. Under MWWeakReg, MWWeakReg+, CohReg and PCGLin, only ME is guaranteed. Intuitively, if a consistency condition requires a total order of write to be observed by all processes and that total order extends σ-consistency, then it will satisfy EP of Dijkstra's algorithm. The proof of the corresponding theorem is shown below.

Theorem 15 *Dijkstra's Algorithm satisfies ME under all the proposed definitions and satisfies ME and EP under both MWReg and MWReg+, but does not satisfy NL under any of the conditions.*

Proof. (Theorem 15) We first show that the algorithm satisfies ME under MWWeakReg. Assume that two processes p_0 and p_1 enter the critical section simultaneously. It follows that both perform write operation $W(Flag[i], \textit{in-cs})$ (Line 6 of Dijkstra's algorithm in Figure 21), and that therefore neither of the read operations $R(Flag[j])$ (Line 7) return *in-cs*. Consider $R_0(Flag[1])$ and $W_1(Flag[1], \textit{in-cs})$. As $R_0(Flag[1])$ does not return *in-cs* by the argument above, it follows that $R_0(Flag[1])$ begins before $W_1(Flag[1], \textit{in-cs})$ ends, i.e., $R_0(Flag[1]) \not\succ_\sigma W_1(Flag[1], \textit{in-cs})$. Therefore, as

each process performs only one operation at a time, we have $W_0(Flag[0], \textit{in-cs}) <_\sigma R_0(Flag[1]) \not>_\sigma W_1(Flag[1], \textit{in-cs}) <_\sigma R_1(Flag[0])$. It follows from the rule above that $W_0(Flag[0], \textit{in-cs}) <_\sigma R_1(Flag[0])$. As there are no other writes to $Flag[0]$, it follows from the definition of MWWeakReg that $R_1(Flag[0])$ returns *in-cs*; therefore p_1 does *not* enter the critical section, and we have a contradiction. Since MWWeakReg is the weakest among our proposed definitions, thus ME is satisfied under all the other definitions.

Eventual Progress (EP) may be violated under PCGLin. To see this, consider the following execution:

$$w_0(Flag[0], \textit{requesting}), r_0(Turn, 1), r_0(Flag[1], \textit{requesting}), r_0(Turn, 1), ...$$

$$w_1(Flag[1], \textit{requesting}), r_1(Turn, 0), r_1(Flag[0], \textit{idle}), w_1(Turn, 1), r_1(Turn, 2), r_1(Flag[2], \textit{requesting}), r_1(Turn, 2), ...$$

$$w_2(Flag[2], \textit{requesting}), r_2(Turn, 0), r_2(Flag[0], \textit{idle}), w_2(Turn, 2), r_2(Turn, 3), r_2(Flag[3], \textit{requesting}), r_2(Turn, 3), ...$$

.....

Initially, $Turn$ is set to 0 and $Flag[i]$ is set to *idle* for all i. In this execution, p_0 is slow at the beginning, so all the processes except p_0 enter the repeat loop and update their $Flag$ entries to *requesting*. Next, all the processes except p_0 enter the while loop and read $Turn = 0$ and $Flag[0] = \textit{idle}$; thus they all write their ids to $Turn$. Suppose that these writes are performed concurrently.

Once p_0 updates its $Flag$ to *requesting*, each process continues by repeatedly reading $Turn$ in line 3 until it receives its own id as the result of some read. However, under PCGLin, the order of concurrent writes may be observed differently by subsequent reads of different processes; thus any of these reads may return any of the concurrently written values of $Turn$, so there is no guarantee that any process will ever read its own id. If none does so, none will pass the while loop, and EP is violated. Since PCGLin is stronger than MWWeakReg, MWWeakReg+ and CohReg, it follows that EP may be violated under those conditions as well.

Next we show that EP is guaranteed when the algorithm executes under MWReg.

Note that each process writes to $Turn$ at most once before some process enters the critical section. By the definition of MWReg, there exists a sequence of all the write operations on $Turn$ such that all read operations that begin after the last write to $Turn$ return the id of that write. Thus, the process whose write is last in that sequence will pass the write loop and enter the critical section. Therefore progress occurs. As MWReg+ is stronger than MWReg, MWReg+ also guarantees EP.

Since Dijkstra's Algorithm does not guarantee NL even with atomicity, it does not guarantee NL under any of the proposed conditions.

■

From the proof above, we notice that Dijkstra's algorithm cannot make progress if the underlying consistency condition does not require a common view of the order of write operations.

CHAPTER VI

CONCLUSION

If Lamport's consistency conditions continue to be of interest in the area of distributed shared memory, as seems likely, it is essential that these conditions be formally extended into the multi-writer model. While this extension is simple in the case of atomicity, it is more difficult and potentially ambiguous for the weaker condition of regularity.

In this thesis, we attempt to obtain a formal extension of Lamport's definition of regularity from the single-writer model ([16]) to the more general multi-writer model. We have shown that the extension is not trivial. While there exist various ways to extend the single-writer definition, the resulting definitions will have different strengths.

We started from a generic algorithm, which is a generalization of several existing protocols that use quorum systems to implement read/write register. We then identified three building blocks from the algorithm. By applying different combination of the building blocks, we were able to formalize the consistency conditions the algorithm can yield and to identify possible candidates for multi-writer regularity. Our results showed that six of the consistency conditions yielded are possible definitions of multi-writer regularity. For each of the six extended consistency conditions, we presented the formal definition, provided the implementation algorithm and proved the correctness of the implementation algorithms.

The definitions form a lattice as respect to their strength, and the implementations have varying costs with respect to number of messages, size of messages, time delay, and local memory requirements. Taken together, the set of definitions point out the ambiguity of the informal notion of regularity and the algorithms suggest that

different costs may be associated with different choices for disambiguating.

Locality is a desirable property of consistency conditions, which enhances modularity and concurrency. In our study, we show that all the proposed definitions satisfy locality.

We have also analyzed the relationships between these extended consistency conditions and a number of other well-known consistency conditions. As part of this analysis, we gave a partial order describing the relative strengths of these consistency conditions.

Finally, we provide a practical context for our results by studying the correctness of two well-known algorithms for mutual exclusion when the variables are implemented under our proposed consistency conditions. We find that Peterson's algorithm is fully correct under all the conditions. Dijkstra's algorithm satisfies only some of the constraints of the mutual exclusion problem under any of the conditions.

In our work, we do not take into consideration any failures that may occur at either server processes or client processes in the quorum system that is used to implement the shared memory objects. Therefore, one direction of our future work is to explore the fault-tolerant versions of our proposed definitions and their implementations.

Our implementation algorithms exhibits differences in cost. However, we do not know if the differences are actually necessary. Thus another interesting topic to explore would be to show some complexity separation between our proposed conditions, i.e., if we can prove some lower bound on the cost of any algorithm for some consistency condition C.

The still weaker condition of *safeness* [16] can also be extended to the multi-writer model by means of similar techniques to those we have used here; this is one possible avenue of future work. It might also be worthwhile to explore ways of formalizing

the multi-writer version of consistency conditions met by the probabilistic quorum systems of [21], which operate more efficiently than strict quorum systems at the expense of occasionally providing outdated information.

In addition to seeking the formal specifications of those weaker consistency conditions, it is worthwhile to identify certain problems that cannot be solved under those consistency conditions, instead of just showing that some algorithm is incorrect.

Finally, exploring the semantics and consistency model of other data structures, which is built on top of quorum systems, might also be of great interest.

REFERENCES

[1] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. *ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany, June 1993, pp.251-260.

[2] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, Vol.42, No.1, pp.124-142, 1995.

[3] H. Attiya and R. Friedman. A Correctness Condition for High Performance Multiprocessors. *SIAM Journal on Computing*, Vol.27, No.6, pp.1637-1670, 1998.

[4] M. Ahamad, P. W. Hutto, G. Neiger, J. E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. TR GIT-CC-93/55, Georgia Institute of Technology, July 1994.

[5] F. Anger. On Lamport's Interprocess Communication Model. *ACM Transactions on Programming Languages and Systems*, Vol.11, No.3, pp.404-417, 1989.

[6] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd Ed.*. Hoboken, New Jersey, USA, John Wiley and Sons, 2004.

[7] R. A. Bazzi. Synchronous Byzantine Quorum Systems. *Distributed Computing*, Vol 13, No.1, pp.45-52, 2000.

[8] S. Ben-David. The Global Time Assumption and Semantics for Concurrent Systems. *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, Toronto, Ontario, Canada, August 1988, pp.223-231.

[9] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, Vol.43, No.2, pp.225-267, 1996.

[10] J. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[11] V. K.Garg and M. Raynal. Normality: A Consistency Condition for Concurrent Objects. *Parallel Processing Letters*, Vol.9, No.1, pp.123-134, 1999.

[12] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, Vol.12, No.3, pp.463-492, 1990.

[13] L. Higham and J. Kawash. Tight Bounds for Critical Sections in Processor Consistent Platforms. *IEEE Transactions on Parallel and Distributed Systems*. Vol.17, No.10, pp.1072-1083, 2006.

[14] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, Vol.17, No.8, pp.453-455, 1974

[15] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, Vol. C-28, No.9, pp. 690-691, 1979.

[16] L. Lamport. On Interprocessor Communication. Part I and II. *Distributed Computing*, Vol.1, No.2, pp. 77-101, 1986.

[17] R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report 180-88, Department of Computer Science, Princeton University, 1988.

[18] N. Lynch and A. Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. *Proceedings of the 16th International Symposium on Distributed Computing*, Toulouse, France, October 2002, pp.173-190.

[19] H. Lee and J. Welch. Randomized Registers and Iterative Algorithms. *Distributed Computing*, Vol. 17, No.3, pp.209-221, 2005.

[20] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, Vol.11, No.4, pp.203-213, 1998.

[21] D. Malkhi, M. Reiter, A. Wool, and R. Wright. Probabilistic Quorum Systems. *Information and Computation*, Vol.170, No.2, pp.184-206, 2001.

[22] G. L. Peterson and M. J. Fischer. Economical Solutions for the Critical Section Problem in a Distributed System. *Proceedings of the 9th ACM Symposium on Theory of Computing*, Boulder, Colorado, USA, May 1977, pp.91-97.

[23] G. L. Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, Vol.12, No.3, pp. 115-116, June 1981.

[24] M. Raynal. *Algorithms for Mutual Exclusion*. Cambridge, Massachusetts, USA, MIT Press, 1986.

[25] M. Raynal and A. Schiper. From Causal Consistency to Sequential Consistency in Shared Memory Systems. *Proceedings of the 15th Conference on Foundations of Software Technologies and Theoretical Computer Science*, Bangalore, India, December 1995, pp.180-194.

[26] M. Raynal and A. Schiper. A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories. *Proceedings of the 9th International Conference*

on Parallel and Distributed Computing Systems, Dijon, France, September 1996, pp.125-131.

[27] R. C. Steinke and G. J. Nutt. A Unified Theory of Shared Memory Consistency. *Journal of the ACM*, Vol.51, No.5, pp.800-849, 2004.

[28] C. Shao, Evelyn Pierce and J. L. Welch. Multi-Writer Consistency Conditions for the Shared Memory Objects. *Proc. 17th International Conference on Distributed Computing (DISC)*, Sorrento, Italy, October 2003, pp. 92-105.

APPENDIX A

EXISTING CONSISTENCY MODELS

We give the formal definitions of several existing consistency conditions using the model we defined in Chapter II.

Atomicity, also called Linearizability, is the strongest consistency condition ([12]). It requires that there exist a total ordering of all the operations in a schedule that respects both the semantics of the objects and the partial order of executions of the operations. The formal definition is given below.

Definition 18 (Atomicity) *There exists a permutation π of $ops(\sigma)$ such that:*

- *π is legal,*
- *π is σ-consistent.*

Sequential consistency([15]) requires that there exist a total order of all the operations in a schedule that respects the semantics of the objects and is consistent with the order of operations executed by each process.

Definition 19 (Sequential Consistency) *There exists a permutation π of $ops(\sigma)$ such that:*

- *π is legal,*
- *$\pi|i$ is σ-consistent for all processes i.*

PRAM was introduced in [17]. This consistency condition requires that the write operations of a process be observed by other processes in the order in which they are performed. Formally speaking, a memory consistency condition is PRAM if it satisfies the following:

Definition 20 (PRAM) *For all processes i, there exists a permutation π_i of $(ops(\sigma|i) \cup writes(\sigma))$ such that:*

- *π_i is legal,*
- *$\pi_i|j$ is σ-consistent for all processes j.*

Coherence ([10]) requires sequential consistency on a per-object basis, which means that the operations on different objects executed by the same process may be observed in an order other than that in which they are invoked.

Definition 21 (Coherence) *For all variable x, there exists a permutation π_x of $ops(\sigma|x)$ such that:*

- *π_x is legal,*
- *$\pi_x|i$ is σ-consistent for all processes i.*

Goodman's Processor Consistency (PCG) is rigorously defined in [1]. It is a combination of coherence and PRAM.

Definition 22 (PCG) *For all processes i, there exists a permutation π_i of $ops(\sigma|i) \cup writes(\sigma)$ such that:*

- *π_i is legal,*
- *$\pi_i|j$ is σ-consistent for all processes j,*
- *$\pi_i|writes(\sigma, x) = \pi_j|writes(\sigma, x)$ for all processes j and all variable x,*

where $writes(\sigma, x)$ is the subset of $writes(\sigma)$ that access variable x.

VITA

Cheng Shao received his B.E. degree from Tianjin University in 1997 and M.E. degree from Institute of Computing Technology, Chinese Academy of Sciences in 2000. He started his research work with Dr. Jennifer L. Welch since 2001. His work is focused on distributed data structures, specifically the formal specifications and implementations of the shared data structures. His permanent address is: 3508 Dripping Springs Dr., Plano, TX 75025.