MULTI-VALUED REGISTER SIMULATIONS

A Thesis

by

SAI KRISHNA ADITYA BIRADAVOLU

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

MASTER OF SCIENCE

Chair of Committee,	Jennifer Welch
Committee Members,	Andreas Klappenecker
	Krishna Narayanan
Head of Department,	Scott Schaefer

May 2021

Major Subject: Computer Engineering

ABSTRACT

Shared read-write registers help processes in a shared-memory system to communicate by performing read and write operations on them. In this thesis, we study the wait-free simulations of shared read-write registers using weaker shared registers, for two consistency conditions regularity and atomicity. We propose an algorithm which is a hybrid of the existing tree algorithm and the clique algorithm such that it gives a trade-off between the two algorithms in the number of read/write steps used and the number of base registers. We also explore if existing algorithms, particularly the tree algorithm, can be extended to simulate multiple writer registers, since register simulations in most prior works have been for single-writer registers.

ACKNOWLEDGMENTS

Firstly I would like to thank my supervisor, Dr. Jennifer Welch whose guidance and support throughout my Master's degree helped me transition comfortably to the field of Distributed Algorithms. I would also like to thank Dr. Dariusz Kowalski. My weekly discussions with him and Dr. Welch were critical in shaping the key ideas behind the algorithms proposed in this thesis. I am also grateful to Dr. Krishna Narayanan for his advice and guidance during the initial semesters of my graduate studies.

Finally I would like to thank my parents and my uncle, Ramesh Amancherla for being incredibly supportive throughout my education.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professor Jennifer Welch [advisor] and Professor Andreas Klappenecker of the Department of Computer Science and Engineering and Professor Krishna Narayanan of the Department of Electrical and Computer Engineering, Texas A&M University.

The thesis was completed by the student in collaboration with Dr. Jennifer Welch and Dr. Dariusz Kowalski, Department of Computer & Cyber Sciences, Augusta University .

Funding Sources

This work was made possible in part by NSF under Grant Number 1816922.

Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the NSF

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND RELATED WORK

A shared read-write register is a basic concurrent data structure that stores a value and supports two operations to read and to write new values into it. These data structures help processes in a shared-memory system to communicate by performing read and write operations on them. The problem of synchronizing concurrent processes that try to access the shared registers in a shared-memory system has been studied since the 1970s. This eventually led to Lamport ([1], [2]) formally introducing the notion of concurrent consistency conditions, namely safety, regularity and atomicity. Lamport ([1], [2]) also studied how registers with certain base properties can be used to implement stronger registers, and correspondingly proposed algorithms to do so. Recent works ([3], [4]) have again renewed the interest in register simulations. In this thesis, we study the wait-free simulation of shared read-write registers using a collection of weaker shared read-write registers. Wait-freedom means that every operation invoked by a process completes within a finite number of steps taken by that process, no matter how fast the other processes operate. Shared read-write registers can be classified using properties such as their concurrency consistency condition, number of read and write processes that are supported and whether they are binary or multi-valued.

There are many prior works studying the simulations of multi-valued registers from binary registers, atomic registers from regular registers, multi-reader registers from single-reader registers, etc. In this thesis, we look at simulations of multi-valued multi-reader registers from binary multi-reader registers. Lamport ([1], [2]) proposed the unary representation algorithm which simulates a k-valued regular/atomic single-writer register using $k-1$ binary regular/atomic single-writer registers. The tree algorithm in [5] also uses $k-1$ base registers to implement a multi-valued regular single-writer register from binary regular single-writer registers, but improves on the number of steps taken by the reader and writer. Later Chaudhuri et al. [6] improve on the number of write steps used by proposing a one-write algorithm called the clique algorithm which uses $k(k-1)/2$

base registers to implement a k-valued regular single-writer register from binary regular single-writer registers. Although the clique algorithm significantly improves on the number of write steps, it is worse off in the steps taken by the reader and the number of base registers used, when compared to the tree algorithm. We attempt to bridge this gap and propose an algorithm to simulate multi-valued regular/atomic single-writer register from binary regular/atomic single-writer registers, that gives a trade-off between the tree and clique algorithms in the number of read/write steps used and the number of base registers used.

The clique algorithm [6] can simulate an atomic multi-valued register from binary atomic base registers, but the tree algorithm cannot simulate an atomic register. Chen and Wei [7] proposed a modification to the tree algorithm to simulate an atomic register using binary atomic base registers. While all these algorithms work with invisible reads, i.e., when the reader doesn't write to any base registers, the number of base registers can be improved when visible reads are used [8]. We present an algorithm with invisible reads that is based on both the tree and clique algorithms and attempts to utilize the advantages of both algorithms to result in a hybrid that exhibits trade-off results in various metrics.

Most of the register simulations in these prior works have been for single-writer registers. This is because regularity was originally only defined for the single-writer scenario. Shao et al. [9] first defined a set of multi-writer consistency conditions for regularity. In this work, we also explore if existing algorithms, in particular the tree algorithm, can be extended to simulate multiple writer registers which satisfy any of the consistency conditions for regularity as in [9].

1.1 Related Work

As mentioned above Peterson [8] and Lamport ([1], [2]) have some of the first proposed algorithms for simulating multi-valued registers. Peterson's algorithm uses a buffer-based implementation to implement a multi-valued atomic register. The algorithm uses two global buffers, each with $\log k$ registers, and every reader has another buffer. The idea is that the writer alternatively writes

to the two global buffers and also writes the value to be returned into the reader's own buffer. Aghazadeh et al. [3] also use an improved buffer-based implementation and simulate a multi-writer atomic register.

Lamport's unary representation algorithm ([1], [2]) simulates a multi-valued regular register from binary regular registers. It uses a base register corresponding to each value that can be written, "arranged" in descending order. The WRITE protocol for a value v writes 1 into the corresponding base register and writes a 0 into every other base register to its right. The READ protocol works in the opposite direction, by reading the right-most register first, going to the left until a 1 is read, and returns the corresponding value associated with that register.

The tree algorithm that was described in [5] simulates a multi-valued single writer regular register from binary base single-writer regular registers. The READ operation in the simulated register is thought of as a binary search in a binary tree with registers as nodes of the tree. The leaves of the tree are the values that the simulated register can take. The READ operation takes the path from the root to a leaf based on what value is returned along each node of its path. If a value of 0(1) is read, it then proceeds to the left(right) child of the node. If the subsequent child is a leaf then that is the value returned by the simulated register, if not, the child node is also read and the READ proceeds further down the path. Similarly a WRITE operation goes from the leaf to the root of the tree, starting from the parent of the value at the leaf that is supposed to be written. A 0(1) is written in the parent node if the child is the left(right) child, and the operation ends when the root node is written.

The clique algorithm [6] implements a k-ary single writer regular/atomic register from binary base regular/atomic registers respectively. The algorithm uses a k-vertex clique with a binary register representing each edge of the clique. The idea is that the simulated write algorithm consists of flipping the base register on the edge between the old and new value of the write. The simulated read algorithm reads all the base registers, and infers the value to be returned, by applying a function to the values read.

Chen and Wei [7] have extended the tree algorithm and modified it to simulate a k-valued atomic register from binary atomic registers. Their algorithm uses a larger tree structure by including a layer of k^2 height 1 nodes, each of them being an atomic register, such that they represent the parents of all possible pairs of old-new values. This helps the algorithm in atomically changing from the current value to the new value. The tree on top of the bottom layer with k^2 nodes is a similar binary tree as used in the regular tree algorithm, and all the nodes apart from the bottom layer can be regular registers. The READ protocol of this modified tree algorithm is the same as before, whereas the WRITE protocol involves an additional step of flipping the parent node of the old-new leaf pair, after performing the write operation with the index of the old value.

Shao et al. [9] came up with formalization to define consistency conditions for regular multi-writer registers. They introduced four definitions of varying relative strengths. They also introduced the corresponding algorithms that satisfy each of these consistency conditions. They point out various ways in which the original definition of regularity ([1], [2]) can be extended and the costs associated with it. We will further discuss this in Section 4.2.

2. MODEL

Shared read-write registers store a value and support read and write operations. For a read-write register X, read operation has an invocation read(X) and a response of return(X,v). Similarly, to write a value v into the shared read-write register, the write invocation is write(X,v) with a response of ack(X). If in a sequence of non-overlapping operations, each read returns the value of the latest preceding write, we say that it is a legal sequence of operations.

Consider a sequence of operations on the shared register to be σ. We denote $\sigma|i$ to be the subsequence that contains the invocations and responses of a process p_i. A sequence of operations σ form a schedule if $\sigma|i$ starts with an invocation and consists of invocations and their corresponding responses alternately, with every invocation having a matching response.

We consider the asynchronous shared-memory model, with read and write processes. A shared register is said to be *single-writer* if only one process is allowed to perform write operations on it. A single-writer register can have any number of read processes (i.e., processes that perform read operations on the register). Processes and registers are modeled as state machines, and a *configuration* of the system contains the states of all the processes and registers in the system. An *execution* is a sequence of configurations and steps in an alternating manner, starting with the initial configuration of the system. Each step is an invocation or response of a read or write operation.

We say that a register is *regular* if, for any execution, every read operation returns the value written by an overlapping write operation or the latest preceding write operation. If there is no preceding write operation, the read operation returns the initial value.

We say that a read-write register is *atomic* if, for any execution there exists a total ordering of all the operations in the execution such that it respects the ordering of the non-overlapping read and write operations in the execution, and any read operation in the total order returns the value written by the latest preceding write operation in the total order. If there is no preceding write operation,

the read operation returns the initial value.

An implementation is said to be *wait-free* if, every operation by a process completes in a finite number of steps by that process irrespective of how the other processes in the system behave.

We consider implementing a multi-valued register using a set of binary physical registers, and read and write processes. All invocations and responses for READ and WRITE operations on the simulated logical registers are denoted in uppercase words as READ(X), RETURN(X,v), WRITE(X,v) and ACK, where X is the logical register. Similarly, lowercase words are used to denote operations on physical registers, like read(A), return(A,v), write(A,v) and ack, where A is the physical register.

3. HYBRID ALGORITHMS

In this section we present the hybrid algorithms for the cases of regular registers and atomic registers in Sections 3.1 and 3.2 respectively.

3.1 Hybrid Algorithm for Regular Registers

We consider the problem of simulating a k-ary single writer regular register using binary regular registers as building blocks, where $k > 2$. Here an algorithm is proposed, which is a hybrid of the tree algorithm [5] and the clique algorithm [6] as discussed in Section 1.1.

3.1.1 Tree Algorithm

The tree algorithm takes $\log k$ steps each by the reader and writer, whereas the clique algorithm takes a single write step and $k(k-1)/2$ steps by the reader. The proposed algorithm exhibits a trade-off between the number of steps taken by a simulated read and those taken by a simulated write, as a hybrid between both the tree and clique algorithms (Table 3.1). The key idea is to construct a tree with k leaves where the depth and the number of children of each internal node vary inversely. Each tree node is then implemented using an instance of the clique algorithm.

In the rest of this subsection we describe and analyze a small generalization of the tree algorithm [5], where the base registers are not necessarily binary. See [7] for related discussion. The intuition as mentioned above is that instead of restricting the algorithm to use a binary tree structure, we use a tree such that the number of children each node has is inversely proportional to the depth of the tree. We then use the clique algorithm for each internal node, since now each internal tree node doesn't necessarily hold only two values. Intuitively, the proposed algorithm is intended to use the best of both the existing algorithms, by reducing the tree depth (which causes the write steps to decrease compared to the original tree algorithm), and also reduce the number of registers the clique algorithm uses (and thus the read steps decrease compared to the clique algorithm). This results in the hybrid nature of performance for the proposed algorithm.

Let $T(k, d)$ be a rooted tree with k leaves such that every internal node has the same number of children and all the leaves have depth d, where k and d are such that $k^{1/d}$ is an integer.

Claim: $T(k, d)$ is a complete tree with branching factor $k^{1/d}$ and $(k-1)/(k^{1/d}-1)$ internal nodes.

Definition: A $k^{1/d}$*-ary string* is a (finite) string over $\{0, 1, \ldots, k^{1/d} - 1\}$.

The root node of $T(k, d)$ is labeled with the empty string ϵ. For each node of $T(k, d)$ labeled with $k^{1/d}$-ary string ℓ, the strings $\ell 0, \ell 1, \ldots, \ell x$, where $x = k^{1/d} - 1$, are the labels of its children going from left to right. Note that each of the k leaves of $T(k, d)$ is labeled with a unique $k^{1/d}$-ary string of length d, representing one of the values in V.

Example: $k = 16, d = 2, k^{1/d} = 16^{1/2} = 4$: branching factor is 4 and there are 5 internal nodes (the root and its 4 children). The root is labeled ϵ, its children are labeled 0, 1, 2, and 3, and the leaves are labeled 00 through 33 (in base 4). In base 10, the leaves correspond to the values 0 through 15.

We now describe the hybrid tree algorithm based on $T(k, d)$ to emulate a regular register over the set V of k values, using regular base registers each of which can hold one of $k^{1/d}$ values. For simplicity, assume $V = \{0, 1, \ldots, k-1\}$. Assign a base register to each internal node in $T(k, d)$.

Each base register can take on any value in the set $\{0, 1, ..., k^{1/d} - 1\}$. The simulated read and write algorithms are given in Algorithm 1. We assume that initially every base register contains the value 0 and that there is an initializing simulated write W_0 that writes the initial value and completes before any other operation begins.

Example: To write(9): since 9 in base 10 is 21 in base 4, write 1 into register labeled 2 (parent of leaf labeled 21), write 2 into register labeled ϵ (root).

To read: read 3 from register labeled ϵ (root), read 2 from register labeled 3 (parent of leaf labeled 32), return 14 in base 10 (which equals 32 in base 4).

The following lemma (rephrased from [5]) proves that the proposed hybrid algorithm for regular registers is correct even when the base registers aren't binary.

Algorithm 1 Code for Hybrid tree algorithm.

```
1: READ():
2: ℓ := ϵ
3: for p := d to 1 do
4:     v_p := read base register for node with label ℓ
5:     ℓ := ℓv_p // ℓ = v_d ... v_p
6: return v_d v_{d-1} ... v_1 (in desired base)

7: WRITE(v):
8: let v_d v_{d-1} ... v_1 be the base k^{1/d} representation of v
9: ℓ := v_d v_{d-1} ... v_1
10: for p := 1 to d do
11:     ℓ := ℓ minus last character // ℓ = v_d v_{d_1} ... v_{p+1}
12:     write v_p to base register for node with label ℓ
13: return ACK
```

Definition: Base read r *reflects* base write w if r and w access the same base register and either (i) w completely precedes r, or (ii) w and r overlap and r returns the value that w writes.

Definition: Simulated read R *notices* simulated write W if R contains a base read that reflects a base write contained in W.

Observation: Since every simulated read R reads the root and the initializing write W_0 writes the root, R notices W_0, i.e., every simulated read notices at least one simulated write.

Lemma 1. *(cf. Lemma 4.2 in [5]) Every simulated read R returns the value of the latest simulated write W that R notices.*

Proof. By the observation, W is well-defined. Let s be the last register read by R such that R's read r from s reflects W's write w to s. (There is at least one such register by the definition of "notice".) That is, either (i) w completely precedes r, or (ii) w and r overlap and r returns the value that w writes.

Claim 1: The value b read by r from s is the same as the value written by w into s.

Proof of Claim 1: First note that w either precedes or overlaps r. If w and r overlap, then by the definition of reflects, w writes b. Suppose w completely precedes r and, for contradiction, that w writes some value other than b. By the definition of regularity since r reads b, there is a base write w' that starts after w ends and before r ends that writes b into s. Since a simulated write never

writes to the same base register more than once, w' must be part of a simulated write $W' \neq W$. Since there is only one write of the simulated register, W' follows W. But then R notices W', contradicting the choice of W as the latest simulated write that R notices.

Claim 2: s is the last base register read by R.

Proof of Claim 2: Suppose in contradiction that s is not the last base register read by R. Then after r, R next reads a child t of s, since simulated reads go down the tree. Since simulated writes go up the tree, W writes a child of s before it writes s; by Claim 1, the child of s that W writes is t. Thus R's read of t reflects W's write to t. This contradicts the definition of s as the last register read by R such that R's read of this register reflects W's write to this register. □

Theorem 1. *(cf. Theorem 4.3 in [5]) The hybrid tree algorithm for regular registers is correct.*

Proof. Since clearly the algorithm is wait-free (operations terminate), we just have to show regularity. Let R be a simulated read and W be the latest simulated write that R notices. By Lemma 1, R returns the value that W writes. We must show that either W overlaps R or W is the latest simulated write that precedes R. Obviously R cannot precede W. Suppose in contradiction W precedes R but there is another simulated write W' such that W precedes W' and W' precedes R. Since W' writes the register corresponding to the root of $T(k, d)$ before R reads that register, R notices W', contradicting the choice of W as the latest simulated write that R notices. □

We next state some obvious facts about the complexity of the generalized tree algorithm based on $T(k, d)$.

Lemma 2. *The number of steps taken in the simulated write algorithm is d times the number of steps for writing one of the tree node registers. The number of steps taken in the simulated read algorithm is d times the number of steps for reading one of the tree node registers. The number of binary base registers is $\frac{k-1}{k^{1/d}-1}$ times the number of binary base registers used to implement each tree node register.*

3.1.2 Clique Algorithm

We use the clique algorithm [6] to implement each tree node. We briefly review that algorithm next. To implement an m-ary regular register using binary base registers, where $m > 3$, conceptually there is a clique on m nodes, one for each value, and a binary base register is associated with each (undirected) edge in the clique, resulting in $\frac{1}{2}(m-1)m$ base registers. The simulated write algorithm consists of flipping the value in the base register associated with the edge that connects the old simulated value to the new simulated value, resulting in one step per simulated write. The simulated read algorithm consists of reading all the base registers and applying a function to the values read, resulting in $\frac{1}{2}(m-1)m$ steps per simulated read. The function is the following: for each value v, calculate the number of edges incident on the corresponding node in the clique from whose associated register a 1 was read. If, for every node, the number of such edges is even, then return the initial value; otherwise, return the largest value for which the number of such edges is odd.

Theorem 2. *(Theorem 4.6 in [6]) The clique algorithm correctly implements a regular m-ary register using regular binary base registers.*

3.1.3 Combination

Corollary 1. *The algorithm resulting from using a copy of the clique algorithm, instantiated with $m = k^{1/d}$, for each internal node of $T(k, d)$ in the generalized tree algorithm results in an algorithm for simulating a k-ary regular register using binary base registers that is correct (termination, regularity) and uses*

- *d steps per simulated write ,*
- *$\frac{d}{2}(k^{1/d} - 1)k^{1/d}$ steps per simulated read, and*
- *$\frac{1}{2}(k-1)k^{1/d}$ binary base registers.*

Essentially we have an algorithm that is a hybrid between the clique algorithm ($d = 1$) and the original tree algorithm ($d = \log_2 k$), and gives a trade-off between the number of steps taken by the simulated read and the simulated write. Table 3.1 shows the trade-off.

	clique	2-hybrid	d-hybrid	tree
depth	1	2	d	$\log_2 k$
branching factor	k	$\sqrt{k}$	$k^{1/d}$	2
no. internal nodes	1	$\sqrt{k}+1$	$\frac{k-1}{k^{1/d}-1}$	$k-1$
no. regs/internal node	$\frac{1}{2}(k^2-k)$	$\frac{1}{2}(k-\sqrt{k})$	$\frac{1}{2}(k^{1/d}-1)k^{1/d}$	1
no. steps per write	1	2	d	$\log_2 k$
no. steps per read	$\frac{1}{2}(k^2-k)$	$k-\sqrt{k}$	$\frac{d}{2}(k^{1/d}-1)k^{1/d}$	$\log_2 k$
no. regs	$\frac{1}{2}(k^2-k)$	$\frac{1}{2}(k^{3/2}-\sqrt{k})$	$\frac{1}{2}(k-1)k^{1/d}$	$k-1$

Table 3.1: Behavior of hybrid regular algorithm for different values of d.

We can observe that when the number of write steps is an arbitrarily large constant d, the number of read steps is $\Theta\left(k^{2/d}\right)$, which is an arbitrarily small polynomial, and the number of registers is $\Theta\left(k^{1+1/d}\right)$, which is only slightly larger than the lower bound, as we discuss in the section below.

3.1.4 Lower Bounds

Now let's think about whether our hybrid algorithm can be improved to give better trade-offs.

A result in [5] (Theorem 4.8) states that if the simulated write takes d steps, then the simulated read must take at least $(d!k/2)^{1/d}$ steps, as long as $d \leq (\log_2 k)/3$. Since it can be shown that $\lim_{d\to\infty}(d!)^{1/d}/d = 1/e \approx .37$, this implies that the upper and lower bounds on the number of steps for a read differ by a factor of $\Theta(k^{1/d})$, which goes from $\Theta(k)$ to $\Theta(1)$ as d goes from 1 to $(\log_2 k)/3$.

A result by Wei [10] states that for algorithms with invisible reads (the read algorithm doesn't write to any base registers), the number of base registers must be at least $k-1$. This implies that

the upper and lower bounds on the number of registers also differ by a factor of $\Theta(k^{1/d})$, which goes from $\Theta(k)$ to $\Theta(1)$ as d goes from 1 to $(\log_2 k)/3$.

3.2 Hybrid Algorithm for Atomic Registers

We can generalize the tree algorithm for implementing an k-valued atomic register, as described in [7]. The algorithm also uses a balanced tree with k^2 height 1 nodes (the nodes in the tree that are parents of the leaves), and each node shares a binary register called a switch, to select between its two children. The registers in the tree can be regular except for the height 1 nodes which must be atomic. We try to extend the hybrid tree algorithm described above in section 3.1 for regular registers to simulating an atomic register as well, so that the nodes in the tree need not be binary. But the algorithm relies on the fact that each height 1 node has exactly two children that represent the old and new values that were written and also help to atomically 'switch' from old to new values. Therefore we require the height 1 nodes to be binary and atomic.

The idea is that we modify the tree structure used in [7] but leave the bottom layer (parents of the leaves) of binary atomic registers as it is. The top layer (the tree on top of the binary atomic nodes) is modified to be similar to the tree structure we use in Section 3.1, but with k^2 leaves (since there are k^2 binary atomic nodes underneath it). This will help us tune the depth of the tree (and therefore the branching factor) as desired, and to obtain the necessary trade-off between the number of read/write steps.

We can generalize this tree into two layers. The top layered tree is similar to the hybrid tree described in the previous section for regular registers, with its leaves as the height 1 nodes. Let us refer to the height 1 nodes as $w_0, w_1, \ldots, w_{k^2-1}$. Then in the second layer, each height 1 node w_i has 2 children; the left child has a value of $\alpha = \lfloor i/k \rfloor$, and a right child with a value of $\beta = (i \ mod \ k)$. So, each pair of values (α, β) have a common parent $w_{\alpha * k + \beta}$.

Similar to the hybrid tree algorithm for regular registers, to implement a k-valued atomic register, for the top layer, we use a $T(k^2, d)$ rooted tree, with k^2 leaves (which would become the height

1 nodes of the overall tree) such that every internal node has the same number of children and all the height 1 nodes of the overall tree are at depth d, where k and d are such that $k^{2/d}$ is an integer. Just as in case of hybrid algorithm for regular registers, each internal node of the tree (apart from the height 1 nodes) is implemented using the clique algorithm.

The root node of $T(k^2, d)$ is labeled with the empty string ϵ. For each node of $T(k^2, d)$ labeled with $k^{1/d}$-ary string ℓ, the strings $\ell 0, \ell 1, \ldots, \ell x$, where $x = k^{2/d} - 1$, are the labels of its children going from left to right. Note that each of the k^2 leaves of $T(k^2, d)$ is labeled with a unique $k^{2/d}$-ary string of length d. Assign a base register to each internal node in $T(k^2, d)$. Each base register can take on any value in the set $\{0, 1, ..., k^{2/d} - 1\}$.

The simulated read and write algorithms are given in Algorithm 2. We assume that initially every base register contains the value 0 and that there is an initializing simulated write W_0 that writes the initial value and completes before any other operation begins.

Similar to [7] and in [5], the read algorithm starts from the root and arrives to a height 1 node and then returns the corresponding value of either the left or the right child of that height 1 node, as discussed before. The write algorithm starts by writing a zero at the height 1 node whose left child has the current value and right child has the new value to be written. It then proceeds to the write values to nodes in the tree along the path to the root just like in the regular tree algorithm. This is analogous to performing a WRITE operation of the current value as per Algorithm 1. Now a value of 1 is written to that height 1 node. This is equivalent to performing a WRITE operation of the new value as per Algorithm 1, since all other values to be written along the path to the root will be the same. The WRITE operation is linearized immediately after the final step. This atomic version of the tree algorithm requires the height 1 nodes to be atomic, since the last write operation at the height 1 node (switching its value from 0 to 1) needs to be atomic for the logical WRITE operation to be linearized after this.

Algorithm 2 Code for hybrid tree algorithm for atomic registers.

```
1:  READ():
2:  ℓ := ϵ
3:  for p := d to 1 do
4:      v_p := read base register for node with label ℓ
5:      ℓ := ℓ v_p // ℓ = v_d ... v_p
6:  v = read base register in the leaf of T(k^2, d) with
    label ℓ
7:  i = ℓ (in base 10)
8:  if v==0 then
9:      return ⌊i/k⌋
10: else
11:     return (i mod k)
12: WRITE(v):
13: v_parent := oldval * k + v
14: let v_d v_{d-1} ... v_1 be the base k^{2/d} representation
    of v_parent
15: let parent be the node with label v_d v_{d-1} ... v_1
16: write(parent, 0)
17: ℓ := v_d v_{d-1} ... v_1
18: for p := 1 to d do
19:     ℓ := ℓ minus last character // ℓ =
        v_d v_{d_1} ... v_{p+1}
20:     write v_p to base register for node with label ℓ
21: write(parent, 1)
22: oldval := v
23: return ACK
```

3.2.1 Correctness of Algorithm 2

From Theorem 1 we know that the hybrid tree algorithm for regular registers is correct. The write protocol in Algorithm 2 uses the write protocol in Algorithm 1 as subroutine with $v = v_{parent}$ (from lines 17-20), and the read protocol of Alg. 2 uses the read protocol of Alg. 1 as subroutine (until line 5). We can see that both the read and write operations terminate and Algorithm 2 is wait-free. We now have to show atomicity. To do this we can use the correctness proof in [7] (and also from their prior conference version of the same in 2016 [11]).

Theorem 3. *(cf. Section 4.2 in [7]) The hybrid tree algorithm for implementing atomic registers is correct.*

Proof. Similar to the hybrid tree algorithm for regular registers, we assume that there is an initializing simulated write W_0 that writes the initial value and completes before any other operation begins.

Consider an execution of READ and WRITE operations, and let R be a READ operation in the execution such that it returns a value x. If a WRITE of x is linearized during the execution of

R, then we also linearize R immediately after the first such WRITE. So, R returns the value of the last WRITE operation before it. If there is no such WRITE linearized during the execution of R, then R is linearized immediately after its first step.

So in the case that there is no linearized WRITE during R, it should be proven that a WRITE of x is the last linearized WRITE before the first step of R. Let W be the last such linearized WRITE. Since W is linearized after it's last step, and before the first step of R, this means that W finishes before R starts. If there is no other concurrent WRITE until after R finishes, then R returns the value written by W, by the correctness of Algorithm 1.

Let us assume that there is another WRITE W_1 following W, before R finishes, and say W writes $y \neq x$. This means W_1 must be linearized after the first step of R, since W is supposed to be the last linearized WRITE before the first step of R. This means the last completed write operation before the first step of R can either be the last write at line 21 of W, or the first write before line 21 of W_1 (from Algorithm 2). From the Algorithm 2, it is evident that either of these writes would write y.

For R to return a value of x, this should mean that there is some other write W_2 of x during the execution interval R (from the correctness of Algorithm 1 which is a subroutine in Algorithm 2). This contradicts our assumption that no WRITE is linearized during the execution interval of R. Hence W must have written x. □

	clique	d-hybrid-atomic	atomic-tree [7]
depth	1	$d+1$	$2\log_2 k$ +1
branching factor	k	$k^{2/d}$	2
no. internal nodes	1	$k^2 + \frac{k^2-1}{k^{2/d}-1}$	$2k^2-1$
no. steps per write	1	$d+2$	$2\log_2 k + 2$
no. steps per read	$\frac{1}{2}(k^2-k)$	$1+\frac{d}{2}(k^{2/d}-1)k^{2/d}$	$2\log_2 k + 1$
no. regs	$\frac{1}{2}(k^2-k)$	$k^2+\frac{1}{2}(k^2-1)k^{2/d}$	$2k^2-1$

Table 3.2: Behavior of the hybrid-atomic algorithm.

3.2.2 Algorithm 2 performance analysis

Table 3.2 shows the trade off between the number of steps taken by the read protocol and the write protocol, as a hybrid between the clique algorithm and the modified tree algorithm for atomic registers. The expressions for the number of steps per read, number of steps per write and number of registers can be carried over from *Corollary 1*, but for a $T(k^2, d)$ tree and taking into account the additional layer of k^2 binary atomic registers. This means that the tree has a depth of $d+1$, and so the simulated read and write algorithms follow a path of length $d + 1$. The additional atomic write step (from line 21 in Alg. 2) results in a total of $d + 2$ write steps. We can also notice that the additional k^2 atomic registers will result in the total number of registers to be $\Theta\left(k^{2+2/d}\right)$ which is worse off than that of the regular register case.

The upper extreme reaches the modified tree algorithm [7] for $d = \log_2 k$. But, unlike the hybrid tree algorithm of Section 3.1, we observe here that at the lower extreme this version of the hybrid tree algorithm doesn't reach the clique algorithm. This is also because of the additional layer of binary atomic registers that are added to the tree at the bottom. The number of registers of this hybrid atomic algorithm is always greater than that of the clique algorithm.

It is interesting to check how the number of steps per read would vary between extremes in the hybrid algorithm. When $k = 1$ or 2 (which isn't very interesting), we observe that the hybrid algorithm takes more steps per read than the clique algorithm, no matter what d is. By plotting and comparing the functions, we can observe that for $k \geq 3$, the hybrid algorithm takes fewer steps per read than the clique algorithm if $d \geq 6$.

4. MULTI-WRITER REGISTER SIMULATIONS

4.1 Background

Shared register simulations have been studied in various works ([6], [5], [1], [2]), where a set of shared read-write registers are used to simulate stronger shared read-write registers in a wait-free manner. In particular, simulations of multi-valued safe, regular and atomic registers using binary base registers that are also safe, regular and atomic respectively, have been studied in prior works like [6] and [5]. But notably, most prior research on these multi-valued regular register simulations only focused on simulating single-writer registers and having single-writer base registers. One reason for that was the consistency conditions for safe and regular registers had only been defined for single writer registers, until Shao et al. [9] defined new multi-writer consistency conditions for regularity. We focus on attempting to check if these prior simulations can be extended to multiple writers.

It is intriguing to extend the elegant tree algorithm mentioned before, to multiple writers without making too many modifications by just using multi-writer base registers. However, we'll show through some counterexamples that this doesn't work. In particular, we show that the tree algorithm cannot be extended to satisfy the MWRegWeak multi-writer consistency condition [9], when atomic registers are used (Section 4.2). We further continue to show that the tree algorithm wouldn't satisfy an even weaker multi-writer consistency condition, called MWRegPM, when atomic base registers are used (Section 4.3).

4.2 MWRegWeak Register from Binary Atomic Registers

Consider the set of all logical writes in schedule S to be L_w and define the partial order (L_w, $<_S$), such that $W_1 <_S W_2$ if and only if W_2 is invoked after W_1's response. An element W of L_w is a **maximal** element if there is no other write $W_1 \in L_w$ such that $W <_S W_1$.

Shao et al. [9] describe the MWRegWeak consistency condition as "*A schedule satisfies*

MWRegWeak if each read r returns the value of some write w that either overlaps or precedes r, as long as no other write falls completely between w and r."

Consider the partial order of all logical reads and writes in S, $\{$WRITES(S) $\cup$ READS(S), $<_S\}$. For a READ r in S, let L_{w_r} be the set of all logical writes such that $w <_S r, \forall\, w \in L_{w_r}$.

We can rephrase MWRegWeak using the definition of maximal writes by saying that "A schedule S satisfies MWRegWeak if each read r returns either an overlapping write or a maximal write in (L_{w_r}, $<_S$)".

Atomicity is a stronger consistency condition which requires that all operations in a schedule have a total ordering that respects the partial order of the executions of operations, and also the semantics of the objects ([1], [2], [9]). Therefore, if a schedule satisfies atomicity then it also satisfies the MWRegWeak consistency condition. If the tree algorithm fails to simulate an MWRegWeak logical register using the stronger atomic registers as building blocks, then it implies the tree algorithm also can't simulate an MWRegWeak logical register using weaker MWRegWeak building blocks, since atomicity implies MWRegWeak. Using this, a counterexample is presented as follows:

Consider the tree below in Figure 4.1 that represents a 4-ary logical register X, with the base registers A, B and C all being atomic registers:

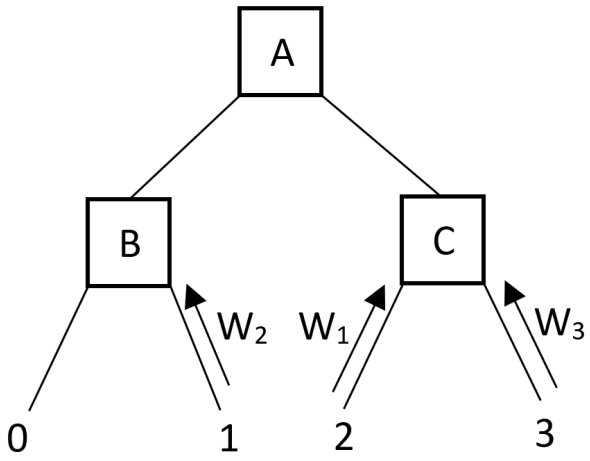

Figure 4.1: Example 4-ary logical register X with base atomic registers.

Consider the schedule for the logical register X as shown below in Figure 4.2:

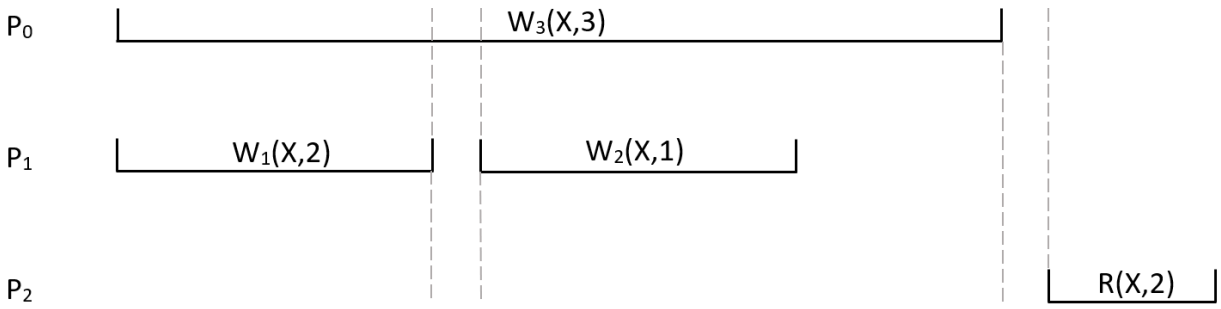

Figure 4.2: Example schedule at X.

Elaborating this further to include the physical registers A, B & C, as shown in Figure 4.3:

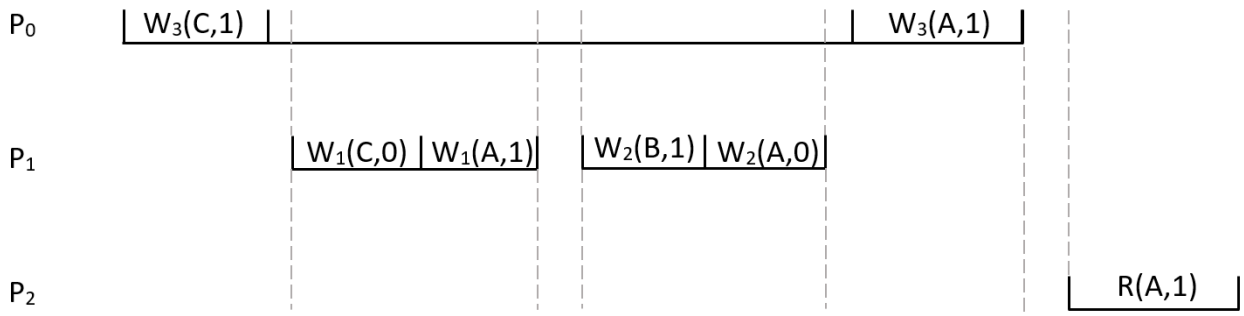

Figure 4.3: Detailed schedule considered in this counterexample.

In this scenario, when R starts reading A, it returns the value of 1 consistent with A being atomic and then proceeds down the tree to read C, as shown below in Figures 4.4 & 4.5.

At A:

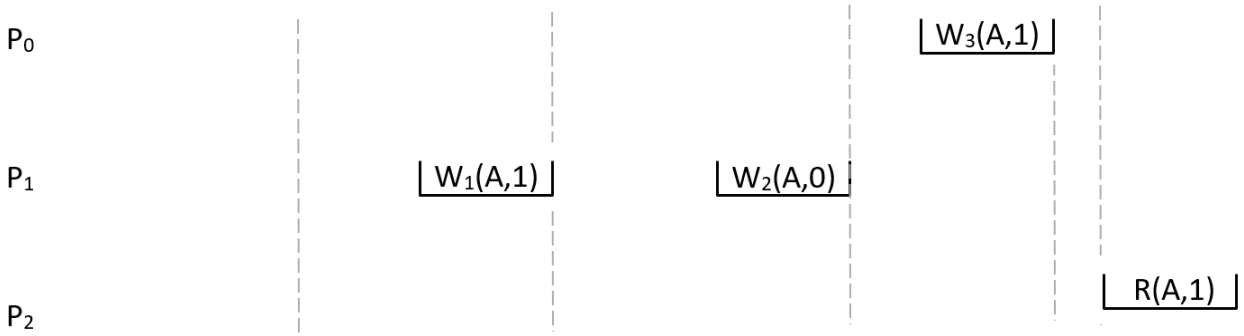

Figure 4.4: R reads 1 at register A.

At C:

Figure 4.5: R reads 0 at register C.

Since R reads a zero here at C, it proceeds to return the left leaf (the left child of C), i.e., 2. But there is a write of 1 by W_2 at X which falls completely between the write of 2 by W_1 and R. This violates the definition of MWRegWeak consistency in this schedule for X.

4.3 MWRegPM Register from Binary MWRegPM Registers

In the previous section, we have proved using a counterexample that an MWRegWeak register cannot be simulated using the tree algorithm. This opens up the question that the tree algorithm might satisfy a weaker multiple writer consistency condition. We have previously mapped the definition of MWRegWeak consistency condition to a maximal element set preceding a READ. To further weaken the consistency condition, the idea is to broaden the set of WRITEs preceding the READ, whose values could be returned by the READ. We consider a "pseudo-maximal" set of elements that also include elements that overlap at least one maximal element.

Now consider a poset X, an element t in X is said to be called a **pseudo-maximal** element if there exists a maximal element m of X such that $t \not< m$.

We can describe a weaker consistency condition named MWRegPM as: "A schedule S satisfies MWRegPM if each read r returns either an overlapping write or a pseudo-maximal write in (L_{w_r}, $<_S$)" (with S and L_{w_r} as defined in section 4.2)

As discussed in Section 4.2, atomicity is a stronger consistency condition than MWRegPM. So,

if the tree algorithm could simulate an MWRegPM logical register using MWRegPM registers as building blocks, then it can do so using the stronger atomic registers as building blocks as well. On the other hand if the tree algorithm can't simulate an MWRegPM logical register with MWRegPM base registers, then it's an open question to check if the use of stronger building blocks like atomic registers might help the case. So we first investigate the case with MWRegPM registers as building blocks using a counterexample, and then proceed to demonstrate a counterexample when the base registers are atomic as well in the next section.

Consider a 4-ary logical register X as shown in Figure 4.6, and when base registers are all satisfying MWRegPM consistency condition.

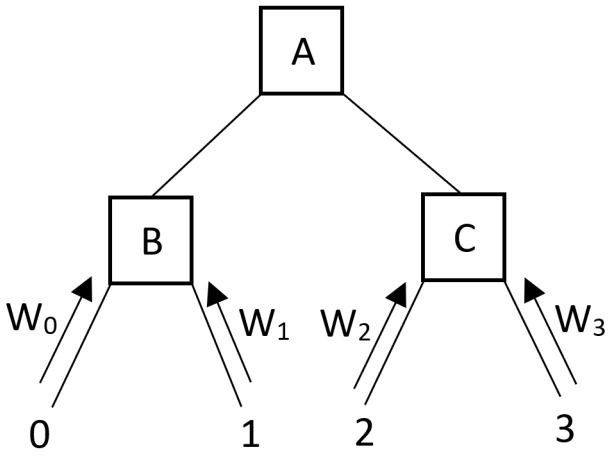

Figure 4.6: Example 4-ary logical register X with base MWRegPM registers.

Consider the following schedule at the logical register X.

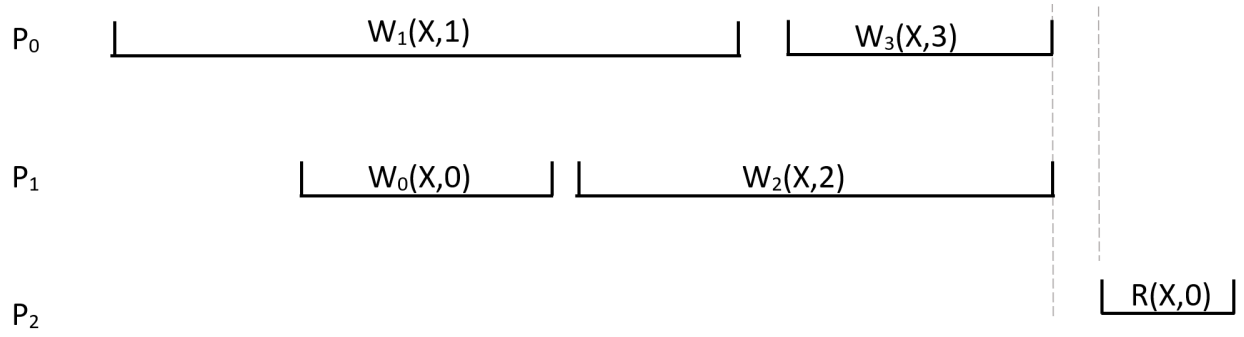

Figure 4.7: Example schedule for this counter example.

As seen above in Figure 4.7, W_2 and W_3 are maximal writes to R. W_1 is pseudo-maximal and W_0 is not. In more detail at each of the physical registers A, B and C, the schedule looks as follows in Figure 4.8:

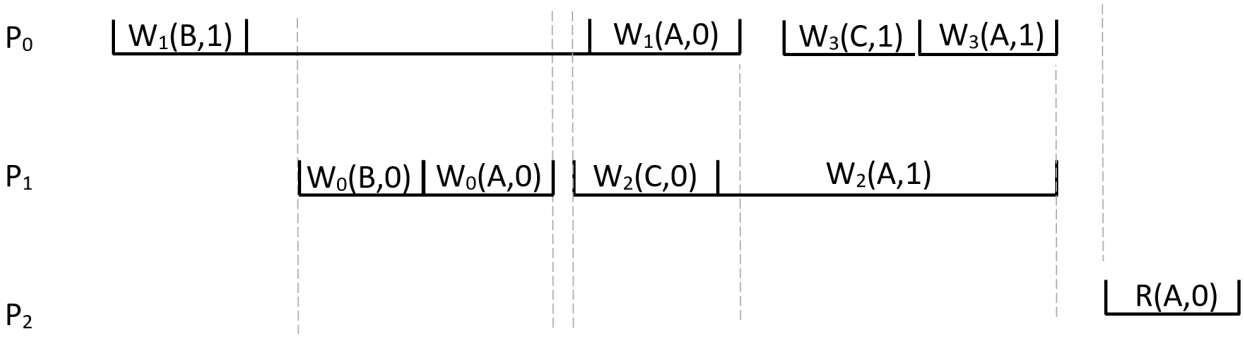

Figure 4.8: Detailed schedule at X.

As shown below in Figures 4.9 & 4.10, R starts reading at A after all four writes are completed, and reads W_1 which is valid since it is pseudo-maximal; and then proceeds to return the value written by W_0, which is not pseudo-maximal. Therefore X violates MWRegPM consistency condition.

At A:

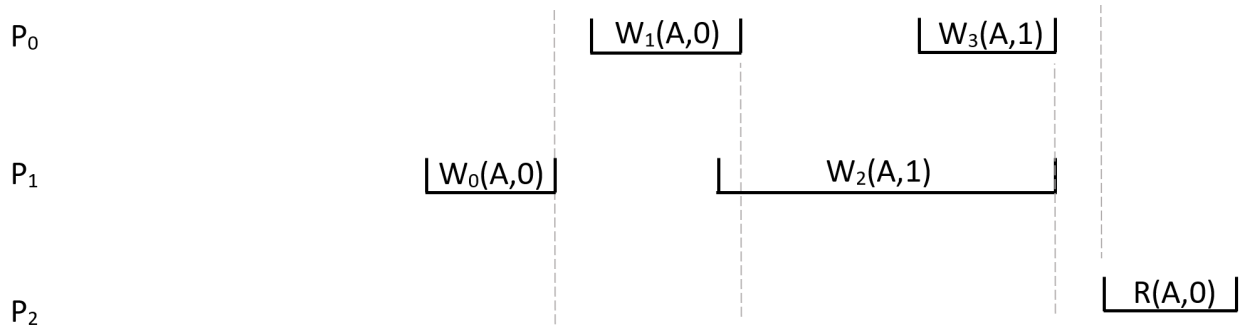

Figure 4.9: At register A, R reads the value written by W_1.

At B:

Figure 4.10: At register B, R reads the value written by W_0.

4.4 MWRegPM register from binary atomic registers

Now consider the scenario when we are trying to simulate a multi-valued MWRegPM register using the tree algorithm and all the base registers are atomic. This time let us consider the tree as shown in Figure 4.11, which represents an 8-ary logical register X:

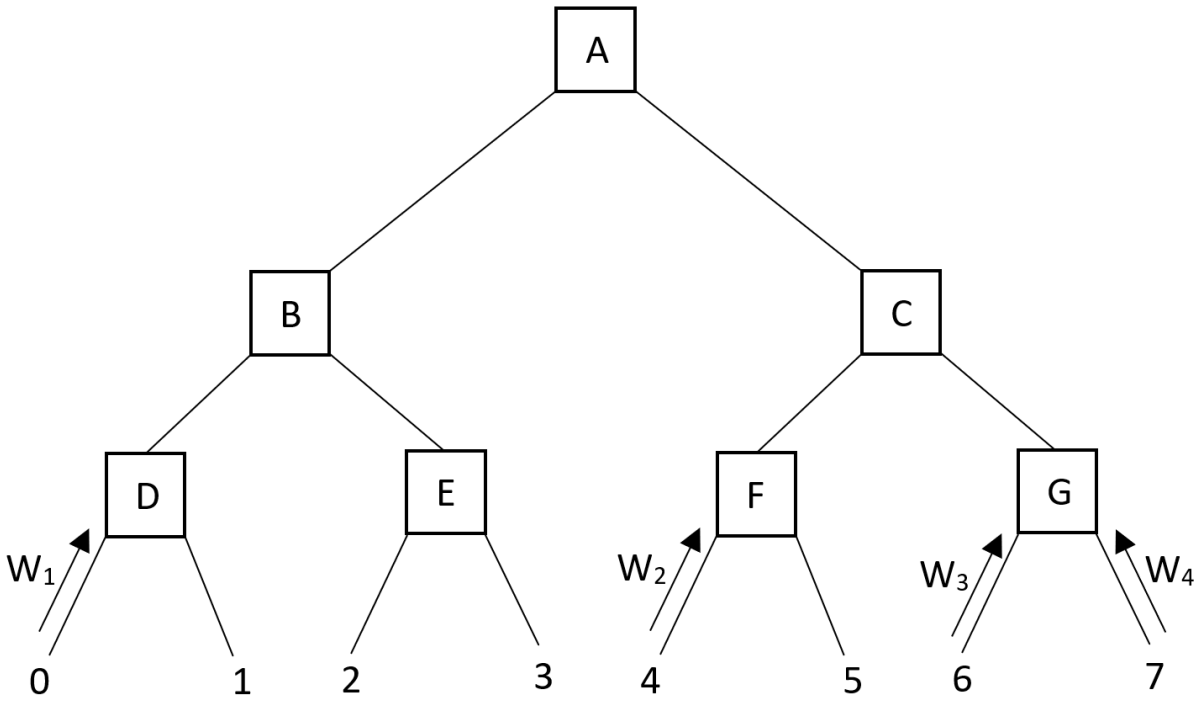

Figure 4.11: Example 8-ary logical register X with base atomic registers.

Consider the following schedule of X shown in Figure 4.12:

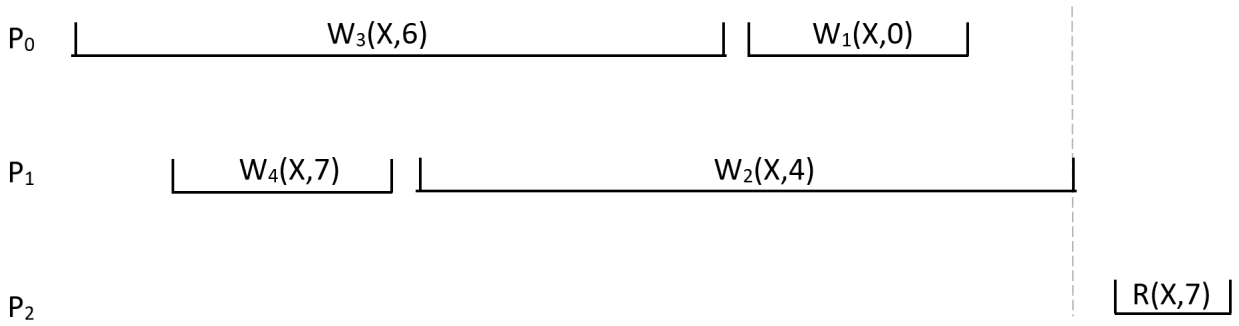

Figure 4.12: Schedule at X considered for this counterexample.

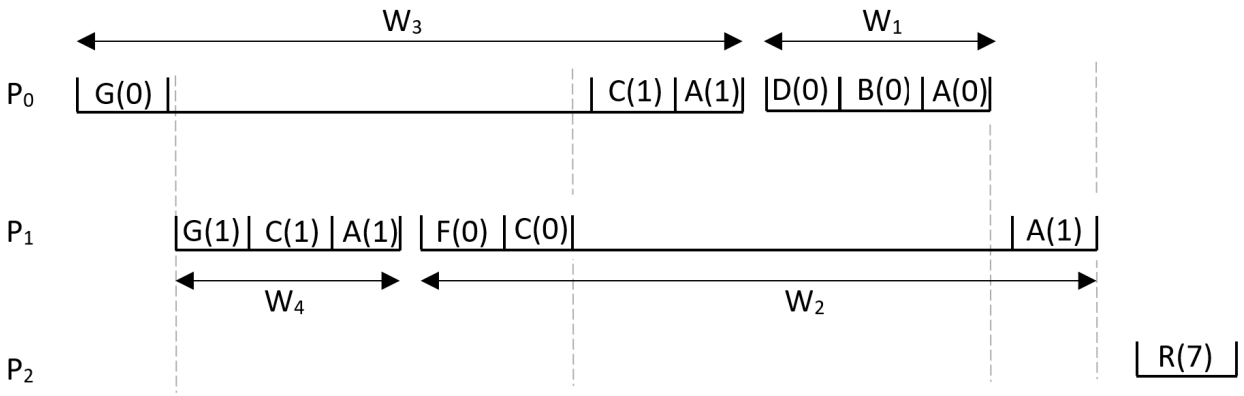

Figure 4.13: Detailed schedule at X considered for this counterexample.

From Figures 4.12 & 4.13, we can see that W_1 and W_2 are maximal, W_3 is pseudo-maximal and W_4 is not. We will proceed to show how R can return the value of W_4 which isn't valid according to $MWRegPM$ consistency condition.

At A:

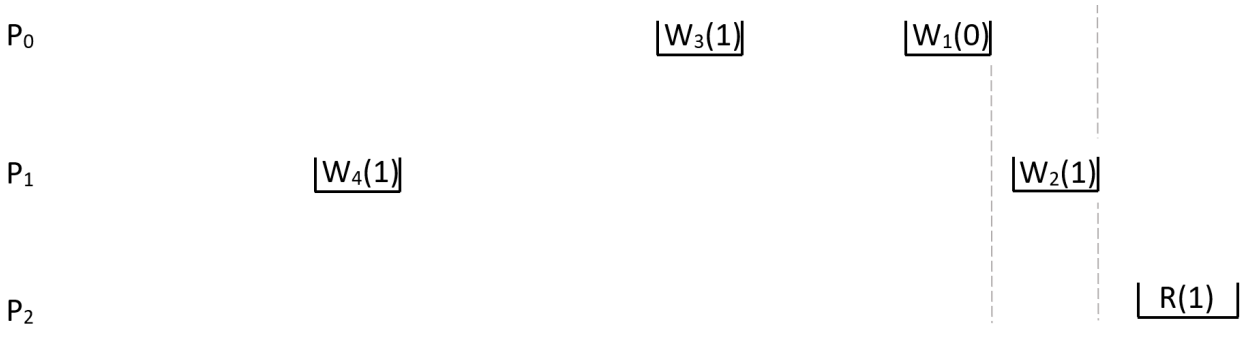

Figure 4.14: At register A, R reads the value written by W_2.

READ R of X starts reading at A after the four WRITEs have completed, and reads a 1 from A,

following atomicity as shown above in Figure 4.14. After proceeding further to C, as seen below in Figure 4.15, it reads a 1 again as per how the schedule is constructed. This leads the reader to G (see Figure 4.16 below), where it again reads a 1, and thus returning 7 which was written by W_4. Thus X violates MWRegPM consistency condition.

At C:

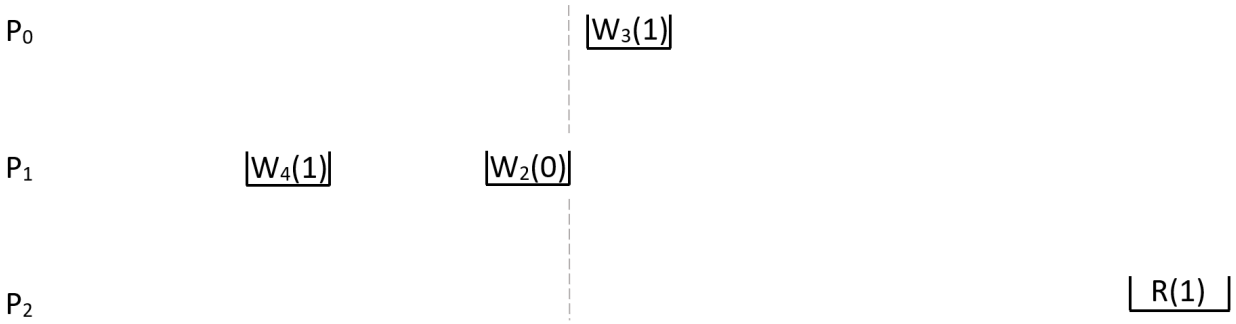

Figure 4.15: At register C, R reads the value written by W_3.

At G:

Figure 4.16: At register G, R reads the value written by W_4.

5. CONCLUSIONS AND FUTURE WORK

In this thesis, we have first studied the wait-free simulation of a single writer regular register with binary regular registers as building blocks. We proposed an algorithm that is based on the existing ideas of tree and clique algorithms ([6], [5]). The algorithm uses a modified tree structure with each internal node of the tree implemented using the clique algorithm. The proposed hybrid algorithm exhibits a trade-off between the number of read and write steps. At one extreme the algorithm behaves exactly like the clique algorithm, and towards the other extreme it simulates the tree algorithm. The depth of the modified tree structure can be adjusted to obtain the desired trade-off in read/write steps. We further extend the idea of this hybrid algorithm to simulate single writer atomic registers, based on the prior work from Chen and Wei [7]. But unlike the hybrid algorithm for regular registers, an extreme of the hybrid atomic algorithm doesn't imitate the clique algorithm and performs much worse off in terms of the number of registers. This is due to an additional layer of atomic registers in the tree structure of the atomic algorithm [7]. It will be an interesting attempt to further modify the hybrid atomic algorithm to improve the performance metrics.

We also explore if existing algorithms, in particular the tree algorithm, can be extended to simulate multiple writer registers, since register simulations in most prior works have been for single-writer registers. We consider the multi-writer consistency condition for regular registers, MWRegWeak, introduced by Shao et. al. [9], and prove using a counterexample that the tree algorithm does not satisfy the MWRegWeak consistency condition. We further introduce a weaker consistency condition named MWRegPM and show that the tree algorithm does not satisfy this consistency condition as well. This direction of work can be further investigated to check if the elegant tree or clique algorithms can be modified to satisfy any of the multi-writer consistency conditions for regularity. That can then be further extended to study the simulation of a multi-

writer regular register using binary single-writer regular registers as building blocks.

REFERENCES

[1] L. Lamport, "On interprocess communication. part I: basic formalism," *Distributed Comput.*, vol. 1, no. 2, pp. 77–85, 1986.

[2] L. Lamport, "On interprocess communication. part II: algorithms," *Distributed Comput.*, vol. 1, no. 2, pp. 86–101, 1986.

[3] Z. Aghazadeh, W. M. Golab, and P. Woelfel, "Making objects writable," in *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pp. 385–395, ACM, 2014.

[4] A. Berger, I. Keidar, and A. Spiegelman, "Integrated bounds for disintegrated storage," in *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, vol. 121 of *LIPIcs*, pp. 11:1–11:18, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[5] S. Chaudhuri and J. L. Welch, "Bounds on the costs of multivalued register implementations," *SIAM J. Comput.*, vol. 23, no. 2, pp. 335–354, 1994.

[6] S. Chaudhuri, M. J. Kosa, and J. L. Welch, "One-write algorithms for multivalued regular and atomic registers," *Acta Inf.*, vol. 37, p. 161–192, Nov. 2000.

[7] T. Z. Chen and Y. Wei, "Step-optimal implementations of large single-writer registers," *Theor. Comput. Sci.*, vol. 826-827, pp. 40–50, 2020.

[8] G. L. Peterson, "Concurrent reading while writing," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 46–55, 1983.

[9] C. Shao, J. L. Welch, E. Pierce, and H. Lee, "Multiwriter consistency conditions for shared memory registers," *SIAM J. Comput.*, vol. 40, no. 1, pp. 28–62, 2011.

[10] Y. Wei, "Space complexity of implementing large shared registers," *CoRR*, vol. abs/1808.00481, 2018.

[11] T. Z. Chen and Y. Wei, "Step optimal implementations of large single-writer registers," in *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, vol. 70 of *LIPIcs*, pp. 32:1–32:16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.