Serializability is unnecessarily strict for real-time systems because most transactions in such systems occur periodically and changes among data values over a few consecutive periods are often insignificant. Hence, data values produced within a short interval can be treated as if they are "similar" and interchangeable. This notion of similarity allows higher concurrency than serializability, and the increased concurrency may help more transactions to meet their deadlines. The similarity stack protocol (SSP) proposed in [25, 26] utilizes the concept of similarity. The rules of SSP are constructed based on prior knowledge of worst-case execution time (WCET) and data requirements of transactions. As a result, SSP rules need to be re-constructed each time a real-time application is changed. Moreover, if WCET and data requirements of transactions are over-estimated, the benefits provided by similarity can be quickly overshadowed, causing feasible schedules to be rejected.

The advantages of similarity and the drawbacks of SSP motivate us to design other similarity-based protocols that can better utilize similarity without relying on any prior information. Since optimistic approaches usually do not require prior information of transactions, we explore the ideas of integrating optimistic approaches with similarity in this thesis. We develop three different protocols based on either the forward-validation or backward-validation mechanisms. We then compare implemen-
tation overheads, number of transaction restarts, length of transaction blocking time, and predictabilities of these protocols. One important characteristic of our design is that, when similarity is not applicable, our protocols can still accept serializable histories. We also study how to extend our protocols to handle aperiodic transactions and data freshness in this thesis. Finally, a set of simulation experiments is conducted to compare the deadline miss rates between SSP and one of our protocol.
Similarity-Based Real-Time
Concurrency Control Protocols

by

Chih Lai

A Thesis Submitted
to
Oregon State University

In Partial Fulfillment of
the requirements for the
degree of
Doctor of Philosophy

Presented January 29, 1999
Commencement June 1999
Doctor of Philosophy thesis of Chih Lai presented on January 29 1999

APPROVED:

Major Professor, representing Computer Science

Chair of the Department of Computer Science

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Chih Lai, Author
Acknowledgements

I wish to express my gratitude to Professor Walter Rudd, my major advisor, for his suggestions and constructive criticisms that have been so helpful to me. His encouragements during my last two-year study make this work possible.

I want to thank Dr. H. Rebecca Callison, my minor advisor, who introduced me to the real-time systems and showed me the art of research. Her contributions are invaluable to this research.

I would like to thank my committee members, Dr. Margaret Burnett, Dr. Shih-lien Lu and Dr. Ed Schmisseur for their suggestions and comments. I also want to thank Dr. Tei-Wei Kuo for his time to answer questions regarding similarity over the e-mail.

The constant support and encouragement from my parents through this long endeavor have made this thesis possible. My special thanks to my wife, Shu-Hui, who encouraged me throughout this long study. Her positive thinking always leads me to find amazing solutions. Finally thanks to my daughter, Emma, to be a very good listener while I was rehearsing my final oral exam, although she is only four months old.
TABLE OF CONTENTS

Chapter 1: Introduction .......................... 1
  1.1 An Overview ................................ 1
  1.2 Motivation and Thesis Outline ............ 3

Chapter 2: Related Research ..................... 5
  2.1 Real-Time Scheduling Protocols ............ 5
    2.1.1 Rate Monotone and Earliest Deadline First Protocols .... 5
    2.1.2 Skip-Over Scheduling .................... 6
    2.1.3 Value-Based Scheduling .................. 7
  2.2 Serializability-Based Real-Time Concurrency Control Protocols .. 9
    2.2.1 Concurrency Control and Serializability ............ 9
    2.2.2 Priority Inheritance and Priority Ceiling Protocols .... 10
    2.2.3 Pessimistic Real-Time Concurrency Control Protocols .... 13
    2.2.4 Optimistic Real-Time Concurrency Control Protocols .... 14
    2.2.5 Speculative Real-Time Concurrency Control Protocol .... 15
  2.3 Epsilon Serializability .................... 16
  2.4 Similarity .................................. 16
    2.4.1 Definition of Similarity .................. 17
    2.4.2 View Similarity and View Δ-Serializability .......... 18
    2.4.3 Conflict Similarity and Conflict Δ-Serializability .... 21
    2.4.4 Similarity Stack Protocol (SSP) .................. 22
    2.4.5 Disadvantages of SSP ..................... 27
  2.5 Real-time Simulator ......................... 30
# TABLE OF CONTENTS (CONTINUED)

## Chapter 3: The Models

- 3.1 Transaction Model ........................................... 32
- 3.2 Object Model .................................................. 34
- 3.3 System Model .................................................. 35

## Chapter 4: New Similarity-Based Protocols

- 4.1 Optimistic Protocols: Forward and Backward Validations .......... 36
- 4.2 Applying Optimistic Approaches in Real-Time Systems .............. 37
- 4.3 Similarity-based Optimistic Concurrency Control- Forward Validation (SOCC-FV) ........................................... 39
  - 4.3.1 Overview of SOCC-FV ........................................ 40
  - 4.3.2 Correctness of SOCC-FV ..................................... 45
- 4.4 Similarity-based Optimistic Concurrency Control- Backward Validation (SOCC-BV) ........................................... 47
  - 4.4.1 Improving the Backward Validation Mechanism ............... 48
  - 4.4.2 Overview of SOCC-BV ........................................ 50
  - 4.4.3 Correctness of SOCC-BV ..................................... 54
- 4.5 Similarity-based Optimistic-then-Pessimistic Protocol (SOPP) .... 56
  - 4.5.1 The Starvation Problem ...................................... 57
  - 4.5.2 The Problem of Chain Blocking ................................ 58
  - 4.5.3 SOPP Overview ............................................... 62
  - 4.5.4 Properties of SOPP .......................................... 65
  - 4.5.5 Correctness of SOPP .......................................... 69
  - 4.5.6 Schedulability of SOPP ...................................... 71
<table>
<thead>
<tr>
<th>Chapter 5: Other Considerations in Real-Time Systems</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Aperiodic Transactions</td>
<td>75</td>
</tr>
<tr>
<td>5.2 Data Temporal Constraints</td>
<td>79</td>
</tr>
<tr>
<td>5.2.1 Extended Object Model</td>
<td>79</td>
</tr>
<tr>
<td>5.2.2 Relationships between Data Similarity and Data Freshness</td>
<td>80</td>
</tr>
<tr>
<td>5.2.3 Related Work - The Force-Wait Protocol</td>
<td>83</td>
</tr>
<tr>
<td>5.2.4 Enforcing Data-Freshness in Real-Time Systems</td>
<td>84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6: Simulation Experiments</th>
<th>88</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Implementation of the Simulator</td>
<td>88</td>
</tr>
<tr>
<td>6.2 Simulation Settings</td>
<td>91</td>
</tr>
<tr>
<td>6.2.1 System Configuration</td>
<td>91</td>
</tr>
<tr>
<td>6.2.2 Transaction Configuration</td>
<td>92</td>
</tr>
<tr>
<td>6.2.3 Performance Metric</td>
<td>94</td>
</tr>
<tr>
<td>6.3 Baseline Model</td>
<td>95</td>
</tr>
<tr>
<td>6.4 Comparison of SSP and SOPP</td>
<td>96</td>
</tr>
<tr>
<td>6.4.1 Effect of Similarity Bounds</td>
<td>97</td>
</tr>
<tr>
<td>6.4.2 Effect of Number of CPUs</td>
<td>100</td>
</tr>
<tr>
<td>6.4.3 Effect of Increasing Conflicts</td>
<td>104</td>
</tr>
<tr>
<td>6.4.4 Effect of Different Workloads</td>
<td>108</td>
</tr>
<tr>
<td>6.4.5 Effect of Overestimating Transaction Execution Time</td>
<td>114</td>
</tr>
<tr>
<td>6.5 Summary</td>
<td>116</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7: Conclusion</th>
<th>117</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>120</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Examples of view-similarity.</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Illustration of $\delta_x \leq sb_x - 2p_x$.</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>Calculation of recency bound on a multi-processor system.</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>System organization of Example 2.4.2.</td>
<td>28</td>
</tr>
<tr>
<td>2.5</td>
<td>Time-lines for executing Example 2.4.2.</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>The validation procedure performed in the SOCC-FV protocol.</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>The write phase procedure that avoids the data regression problem.</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>The validation procedure performed in the SOCC-BV protocol.</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Validation lock phase.</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>The similarity test procedure.</td>
<td>53</td>
</tr>
<tr>
<td>4.6</td>
<td>Integrating the optimistic-then-pessimistic approach with SOCC-BV.</td>
<td>59</td>
</tr>
<tr>
<td>4.7</td>
<td>Blocking chain in a single processor system.</td>
<td>59</td>
</tr>
<tr>
<td>4.8</td>
<td>Blocking chain in a multiprocessor system.</td>
<td>62</td>
</tr>
<tr>
<td>4.9</td>
<td>The validation procedure performed in the SOPP protocol.</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>The similarity test procedure that considers aperiodic transactions.</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>The write phase procedure that considers aperiodic transactions.</td>
<td>78</td>
</tr>
<tr>
<td>5.3</td>
<td>The data freshness test procedure.</td>
<td>85</td>
</tr>
<tr>
<td>5.4</td>
<td>A SOPP validation procedure that considers data freshness.</td>
<td>86</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview of the simulation framework.</td>
<td>89</td>
</tr>
<tr>
<td>6.2</td>
<td>Effect of similarity bounds.</td>
<td>98</td>
</tr>
<tr>
<td>6.3</td>
<td>Effect of increasing number of processors to 4.</td>
<td>101</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES (CONTINUED)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Effect of decreasing number of processors to 1.</td>
</tr>
<tr>
<td>6.5</td>
<td>Effect of increasing conflicts under RM.</td>
</tr>
<tr>
<td>6.6</td>
<td>Effect of increasing conflicts under EDF.</td>
</tr>
<tr>
<td>6.7</td>
<td>Different workloads under RM.</td>
</tr>
<tr>
<td>6.8</td>
<td>Different workloads under EDF.</td>
</tr>
<tr>
<td>6.9</td>
<td>Effect of increasing number of processors under different workloads.</td>
</tr>
<tr>
<td>6.10</td>
<td>Effect of over-estimating transaction execution time.</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Transaction attributes of Example 2.4.2</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>System parameters</td>
<td>92</td>
</tr>
<tr>
<td>6.2</td>
<td>Transaction parameters</td>
<td>93</td>
</tr>
<tr>
<td>6.3</td>
<td>Baseline parameters</td>
<td>95</td>
</tr>
<tr>
<td>6.4</td>
<td>Deadline miss rates between SOCC-FV, SOCC-BV, and SOPP</td>
<td>96</td>
</tr>
<tr>
<td>6.5</td>
<td>Percentage of transactions being restarted under SOPP</td>
<td>99</td>
</tr>
<tr>
<td>6.6</td>
<td>Inference results of different similarity bounds</td>
<td>100</td>
</tr>
<tr>
<td>6.7</td>
<td>Inference results of increasing number of processors to 4.</td>
<td>103</td>
</tr>
<tr>
<td>6.8</td>
<td>Inference results of decreasing number of processor to 1.</td>
<td>104</td>
</tr>
<tr>
<td>6.9</td>
<td>Inference results of increasing conflicts</td>
<td>107</td>
</tr>
<tr>
<td>6.10</td>
<td>Inference results of different workloads</td>
<td>113</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

1.1 An Overview

In real-time systems, each transaction has temporal constraints such as a period and a deadline. A period specifies the interval between occurrences of a transaction. Each transaction must complete before its deadline. Transactions with the same or overlapping periods may run concurrently, competing for shared resources. We need a concurrency control protocol to regulate interleaved transactions and maintain system correctness. On the other hand, we also need a real-time scheduling protocol to help transactions meet their temporal constraints. Since maintaining system correctness and temporal constraints are both important for real-time systems, protocols at these two levels must be carefully integrated so that satisfying constraints at one level does not sacrifice the constraints at another level, and vice versa [7, 32, 42, 45].

However, early real-time research has considered concurrency control and transaction scheduling separately. Either concurrency control protocols are not sensitive to transaction deadlines [14] or transaction scheduling protocols consider only independent transaction sets [33]. Several protocols [1, 39, 15] have been proposed to narrow this gap. They can be roughly divided into pessimistic lock-based protocols [1, 39] and optimistic restart-based protocols [15]. Lock-based protocols may block
higher priority transactions when a lower priority transaction seizes a lock requested by higher priority transactions, creating priority inversion problems. The restarting characteristic of optimistic protocols may cause unacceptably late restarts and waste system resources on transactions that must eventually abort, reducing the chance of meeting transaction deadlines.

All these protocols preserve *serializability* [5, 12, 35, 36] as their correctness criterion. Unfortunately, serializability is unnecessarily strict for real-time systems. This is because in many real-time applications like avionics, manufacturing control, and multimedia systems, the changes among sensor inputs over a few consecutive periods are often insignificant. Hence, data values produced within a short interval, called a *similarity bound* (sb) [25], can be treated as if they are "similar" and interchangeable. In other words, updates and queries should be allowed to execute in violation of the usual serializability criterion as long as they are similar. This notion of *similarity* as an extension of traditional serializability was formally introduced by Kuo and Mok [24]. Similarity allows more concurrency than serializability because it allows a transaction to produce its result using older values without being blocked to wait for the newest values to be produced. Such increased concurrency may help more transactions to meet their deadlines.

Kuo and Mok also developed the *similarity stack protocol* (SSP) [25, 26] to utilize the concept of similarity. SSP takes a pessimistic approach: a transaction is allowed to start and overlap with other transactions only if the SSP rules predict the execution of the transaction will not cause violation of similarity. The SSP rules rely on many worst case assumptions such as the estimated worst case execution time (WCET) and the estimated data access requirements of transactions. Since this estimated usage may be considerably greater than that expected due to data-dependent loops and branches, transactions may be unnecessarily blocked and the system may be under-
utilized. As a consequence, SSP may reject feasible schedules regardless of the fact that SSP is based on similarity which promises more concurrency than serializability.

1.2 Motivation and Thesis Outline

The advantages of similarity and the drawbacks of SSP motivate us to design other similarity-based protocols that can better utilize similarity without relying on any prior information. We start this dissertation by reviewing related research of real-time concurrency control in Chapter 2. The models and terminologies used throughout this dissertation are then defined in Chapter 3.

In Chapter 4, we present our similarity-based real-time concurrency control protocols. As opposed to SSP, our protocols are constructed based on optimistic approaches: a high priority transaction is allowed to start when its period begins. Possible violation of similarity is then validated when transactions are about to commit. Only the transactions that are involved in conflicts which cannot be resolved by similarity are restarted.

The major concern of adopting optimistic approaches in real-time systems is that optimistic protocols have high risk of restarts that can increase the likelihood of transactions missing deadlines. However, under the concept of similarity, fewer transactions may need to be restarted because more conflicts can be resolved by similarity when similarity bounds are increased. This feature of similarity should reduce the impact of roll-backs and make optimistic approaches more attractive and practical for real-time systems. Moreover, our protocols can still accept serializable histories even when similarity is not applicable.

Another important property that needs to be considered in designing real-time system concurrency control is predictability. That is the functional and the timing behavior of a protocol should be as deterministic as necessary to satisfy system specifi-
cation [45]. Hence, we also develop an optimistic-then-pessimistic approach [30] that produces bounded transaction blocking time and guarantees that each transaction can be restarted at most once. A schedulability analysis for the optimistic-then-pessimistic approach is then derived.

In chapter 5, we study how to extend our protocols to handle aperiodic transactions and maintain data freshness. We need to consider aperiodic transactions because aperiodic transactions may interrupt the expected evolution of an object which is the basis of similarity. In other words, if the expected evolution of an object is interrupted by an aperiodic transaction, the data values produced close in time may not be similar anymore. Hence, our protocols need to be extended to resolve conflicts between periodic and aperiodic transactions. Keeping data fresh is also important because if a data value used by a transaction is not sufficiently recent in reflecting the real-world situation (possibly caused by the effects of transaction scheduling or concurrency control [42]), the transaction may deliver meaningless result. This constraint is referred to as data temporal constraint [42, 41, 3, 32, 8, 9, 48]. Hence, in chapter 5, we also extend our protocols to handle data temporal constraints.

In Chapter 6, we study the real-time performance of protocols proposed in this thesis. We first discuss a simulation framework used in this study. This framework is implemented based on our design published in [29]. We then explain the simulation setup and the performance metrics measured in our simulation experiments. Finally, several sets of simulation results and their statistical analyses are discussed. Conclusions appear in Chapter 7.
Chapter 2

RELATED RESEARCH

In this chapter, we review related work in real-time scheduling and concurrency control. Chapter 2.1 discusses real-time scheduling protocols that consider only independent transaction sets. In Chapter 2.2 to 2.4, we review several real-time concurrency control protocols that resolve conflicts based on different correctness criterion. Finally, several existing real-time simulators used in performance studies are examined in Chapter 2.5.

2.1 Real-Time Scheduling Protocols

In this subsection, we review several scheduling protocols that involve only independent periodic transactions. We first discuss rate monotone (RM) and earliest deadline first (EDF) protocols. We then review the Skip-Over protocols. Finally, we discuss a value-based scheduling protocol that later might be used to enhance our work.

2.1.1 Rate Monotone and Earliest Deadline First Protocols

For scheduling real-time transactions, Liu and Layland proposed the rate monotonic priority assignment (RM) and earliest deadline first (EDF) scheduling protocols [33]. Both of them are priority driven and preemptive: a higher priority transaction can interrupt and preempt any active lower priority transaction. RM protocol is a fixed priority scheduling protocol because it assigns priorities to transactions accord-
ing to their static request rates. Hence, one transaction has higher priority than another if it occurs more frequently than the other. EDF, on the other hand, dynamically assigns priorities to transactions based on the deadlines of their current requests. A transaction has higher priority than another if the deadline of its current request is closer than the deadline of the other.

Liu and Layland also derived schedulability conditions for both RM and EDF protocols [33]. They showed that

**Theorem 2.1.1** Under RM protocol, a transaction set with m transactions is schedulable if the processor utilization factor of the transaction set is less than a least upper bound:

\[ U = \sum_{i=1}^{m} (c_i/p_i) \leq m(2^{1/m}) - 1 \]

The \(c_i\) and \(p_i\) represent the estimated worst case computation time and the period of each transaction, respectively.

A transaction set is schedulable by EDF protocol if and only if:

\[ U = \sum_{i=1}^{m} (c_i/p_i) \leq 1 \]

### 2.1.2 Skip-Over Scheduling

In some real-time systems, such as communication systems or aircraft control systems, skipping some periodic instances of transactions is usually acceptable [2, 21, 25]. Koren and Shasha characterized the tolerable skips of a transaction by its skip parameter \(s\) (\(s \geq 2\)), indicating that there must be at least \(s\) periods between two consecutive skips [21]. That is, after missing a deadline, at least \(s - 1\) transaction instances must meet their deadlines. A transaction instance is “red” if the instance
must complete before its deadline; an instance is “blue” (or skippable) if the instance can be aborted at any time. Koren and Shasha also proposed two scheduling protocols, Red Tasks Only (RTO), and Blue When Possible (BWP), in order to schedule red and blue transaction instances [21]. RTO protocol never attempts to schedule a blue transaction instance. BWP is more flexible and schedules blue transaction instances whenever no red transaction instances will be prevented from completing, thus putting idle time to good use.

Unfortunately, these protocols and the RM and EDF protocols consider only independent transactions. In addition, these protocols assume that the skip parameter of a transaction never changes during system execution. In other words, the distance between skippable transaction instances is always fixed. This limitation also excludes the possibility of skipping consecutive transaction instances. However, if transactions run concurrently and access shared data, red and blue transaction instances should be determined dynamically in order to maintain system logical and temporal constraints. Thus, as long as all system constraints are satisfied, skipping consecutive transaction instances should also be allowed.

2.1.3 Value-Based Scheduling

In some real-time systems, transactions with a closer deadline do not necessarily have higher criticality. For example, in a stock market system, if a query transaction and a trade transaction have the same deadline, the trade transaction should be considered more critical than the query transaction. Thus, real-time researchers have argued that a scheduling protocol should consider not only the deadline of a transaction, but also the criticality of a transaction [17, 18, 41].
One way to combine the effects of a transaction's deadline and criticality is to assign each transaction a value $d_i/C_i$, where $d_i$ and $C_i$ are deadline and criticality of transaction $T_i$, respectively. A transaction with smaller $d_i/C_i$ value has higher priority than a transaction with larger value. Several scheduling protocols have been proposed to maximize the total value of completed transactions [17, 18, 34].

However, obtaining maximum system value is not the goal of this research. Instead, our primary objective is to maintain system logical and temporal constraints in concurrent real-time systems, and reduce the amount of wasted system resources. With this concern in mind, the protocol CDLF proposed in [18] attracts our attention. The CDLF protocol assigns transaction priority based on the following value function:

$$V(t) = w * (d_i/C_i) - (1 - w) * L_i \quad 0 \leq w \leq 1$$

where $t$ is current time, $L_i$ is the number of data values that transaction $T_i$ has processed by time $t$, and $w$ is a weighting factor specified by users. The transaction with the smallest value has the highest priority.

We note that CDLF takes into account the amount of work each transaction has done as well as its deadline and criticality. This consideration can be integrated with a conflict resolution mechanism to avoid aborting transactions that have received lengthy service time from a system. For instance, if two transactions have close $L_i$ values, the one with smaller $d_i/C_i$ (more critical transaction) has higher priority. However, if two transactions have close $d_i/C_i$ values (equal importance), then the one which has processed the most data has higher priority.
2.2 Serializability-Based Real-Time Concurrency Control Protocols

In this subsection, we review several real-time concurrency control protocols that attempt to integrate with real-time scheduling protocols. We start our discussion with the *serializability theorem*, the most commonly adopted correctness criterion for concurrency control protocols. Next, we survey several pessimistic and optimistic real-time concurrency control protocols.

2.2.1 Concurrency Control and Serializability

In a process control system no tank can hold more water than its capacity, and in a bank system a service fee may be charged if an account balance is less than a certain amount. Such restrictions are called the *integrity constraints* (or constraints) of a system. A system state that satisfies all integrity constraints is called a *consistent state*, and we say that *system consistency* is maintained [36].

Since a transaction that updates system states cannot finish in a single atomic step, it may transfer a system through a sequence of inconsistent states before consistency is restored in the end. If transactions run concurrently, interleaved transactions may access intermediate inconsistent states and produce results that violate system constraints. Hence, we need a concurrency control protocol to regulate concurrent transactions so that all integrity constraints of a system are maintained.

The question facing by a concurrency control protocol is whether a history produced by the protocol maintains system consistency constraints. All we know about correctness is that a *serial* history is correct since we assume each transaction preserves system consistency when it is executed alone. Hence, any concurrent history that can be shown to be *equivalent* to a serial history is also correct and is referred to as a *serializable* history. This conclusion is the *serializability* theorem [5, 12, 35, 36].
There are several different ways to define equivalence between histories. The most commonly adopted equivalence is conflict equivalence \[5, 36\]. We say two operations conflict if they access the same data and at least one of them is a write operation. Two histories are conflict equivalent if (1) they contain the same set of transactions, and (2) any pair of conflicting operations is ordered the same way in both. A history is conflict serializable if it is conflict equivalent to a serial history. To test conflict serializability of a history, we first construct a conflict graph for the history. The conflict graph has transactions of the history as its nodes, and it contains an edge \( A \rightarrow B \), where \( A \) and \( B \) are distinct transactions of the history, whenever there is an operation of \( A \) which conflicts with a subsequent operation of \( B \). The history is conflict serializable if and only if the conflict graph of the history is acyclic \[5, 36\].

### 2.2.2 Priority Inheritance and Priority Ceiling Protocols

The transaction scheduling protocols proposed by Liu and Layland assume transactions are all independent; that is, transactions do not compete for shared resources \[33\]. Although applying synchronization primitives such as semaphores or monitors to their protocols will relieve this limitation to independent transactions, such mechanisms may lead to the problem of uncontrolled priority inversion \[39\].

Priority inversion occurs when a higher priority transaction is blocked by lower priority transactions. This situation arises when a higher priority transaction attempts to lock a resource that is currently held by a lower priority transaction. To maintain consistency, the higher priority transaction is blocked until the lower priority transaction releases the resource. Such priority inversion disrupts the desired transaction scheduling order and may cause higher priority transactions to miss their deadlines. Furthermore, a higher priority transaction, say \( \tau_1 \), may also be indirectly blocked by medium priority transactions that do not share common semaphore with
\( \tau_1 \) but preempt the lower priority transaction that originally blocks \( \tau_1 \). Such indirect blocking creates unpredictable timing because the worst case duration of priority inversion that any transaction can encounter cannot be known before execution. One partial solution to the priority inversion problem is to prevent a transaction that is in its critical section from being preempted. However, this solution creates a new problem: a lower priority transaction that enters a long critical section may needlessly block higher priority transactions that do not access shared resources.

To address the priority inversion problem and to minimize the amount of blocking, Sha, et al. proposed the priority inheritance protocol \[39\]. The basic idea of priority inheritance protocols is that when a lower priority transaction blocks higher priority transactions, it ignores its original priority and executes its critical section at the highest priority of all the transactions it blocks. The transaction regains its original priority after exiting its critical section. Such inheritance mechanism may cause push-through blocking in which a medium priority transaction can be blocked by a lower priority transaction which has inherited a priority equal to or higher than that of the medium priority transaction. This push-through blocking is necessary to avoid the unpredictability created by indirect blocking.

Under the priority inheritance protocol, a higher priority transaction can be blocked by a lower priority transaction for at most one critical section regardless of the number of semaphores they share. However, the higher priority transaction can still be blocked by more than one lower priority transaction. This is called chain blocking in \[39\]. In addition, deadlocks may occur under a priority inheritance protocol if a circular chain of transactions exists in which each transaction holds a semaphore that is required by the next transaction in the chain \[11\].

Hence, Sha, et al. proposed another protocol called priority ceiling protocol \[39\] to prevent the deadlock and chain blocking. Under the priority ceiling protocol, each
semaphore is first assigned a priority ceiling that is equal to the priority of the highest priority transaction that may lock the semaphore. During system execution, a transaction can lock a semaphore if its priority is higher than the priority ceilings of all semaphores locked by all other transactions. Otherwise, the requesting transaction is blocked and its priority is inherited by the transaction that blocks it. This new form of blocking is referred as ceiling blocking. Ceiling block is similar to the deadlock prevention approach proposed by Havender [11] in which processes must request resources in a linear order. The priority ceiling protocol also guarantees that a transaction can be blocked for at most one critical section, hence preventing chain blocking.

A transaction set that contains \( m \) transactions is schedulable by the priority ceiling protocol if [39]

\[
\forall i, \ 1 \leq i \leq m, \ \frac{c_1}{p_1} + \frac{c_2}{p_2} + \ldots + \frac{c_i}{p_i} + \frac{B_i}{p_i} \leq i \left(2^{1/i} - 1\right)
\]

The \( c_i \) and \( p_i \) represent the estimated computation time and the period of each transaction, respectively. \( B_i \) is the longest critical section for which a transaction may be blocked. Note that such a critical section is the longest one among all transaction's critical sections that are shared with lower priority transactions and are guarded by semaphores whose priority ceilings are higher than or equal to the priority of the transaction.

One drawback of the priority ceiling protocol is that it requires prior knowledge of data access requirements and assumes this knowledge can be provided deterministically. In reality, this information is usually difficult to obtain. In addition, to prevent possible deadlocks the estimated data set accessed by each program could be considerably larger than the set a program really needs, causing unnecessary blocking. As a result, computer systems that rely on these protocols may be under-utilized.
In the next few subsections, we will review other pessimistic and optimistic real-time concurrency control protocols. We then consider a speculative real-time concurrency control protocol that attempts to reduce the overhead spent in restarting conflicting transactions.

### 2.2.3 Pessimistic Real-Time Concurrency Control Protocols

Abbott et. al. [1] studied the performance of various locking-based concurrency control protocols for real-time systems. The concurrency control protocols they considered include ordinary locking, priority inheritance, high priority, and conditional restart protocols.

Under ordinary locking, priority inheritance, and priority ceiling protocols, transactions that request a lock (referred to as requesting transactions) always wait for lock-holding transactions to finish and release their locks, despite the fact that requesting transactions may have higher priorities than lock holding transactions. **High priority** protocol, on the other hand, allows higher priority requesting transactions to abort lower priority lock-holding transactions, making it easier for higher priority transactions to meet their deadlines. The disadvantages of this protocol include: (1) the service time that has been spent on the aborted transactions is wasted, and (2) transactions may be repeatedly restarted.

However, such aborts are unnecessary if a lower priority lock holding transaction ($\tau_h$) can finish within the amount of time that a requesting transaction ($\tau_r$) can afford to wait. **Conditional restart** protocol avoids such unnecessary aborts when condition, $a_r \geq c_h - s_h$, is satisfied. $a_r$ is the time $\tau_r$ can afford to wait; $c_h$ is WCET (the Worst Case Execution Time) of $\tau_h$ and $s_h$ is the service time that has been spent on $\tau_h$. $a_r$ is calculated as $a_r = d_r - (t + c_r - s_r)$, where $d_r$ is the deadline of $\tau_r$ and $t$ is the time a conflict occurs. Unfortunately, this protocol relies on WCET and WCET could be
considerably larger than actual execution time [13, 46]. Therefore, the number of unnecessary aborts could remain very high.

2.2.4 Optimistic Real-Time Concurrency Control Protocols

Under the optimistic concurrency control (OCC) protocol [23], a transaction is divided into computation, validation and write phases. All computations of a transaction are performed in its computation phase. A transaction in this phase is called an active transaction. Conflicts among transactions are checked for serializability only when a transaction completes its computation phase and enters validation phase. A transaction in this phase is called a validating transaction. If no conflict is detected in a transaction’s validation phase, the transaction enters its write phase and commits. Otherwise, the transaction is aborted and restarted.

However, when OCC protocol detects a conflict, only the validating transaction can be aborted, regardless of the fact that the validating transaction may have higher priority than the transactions which conflict with it. This is because under OCC protocol transactions are validated only against committed transactions, and the committed transactions, of course, can never be aborted. Clearly, this backward validation mechanism offers no flexibility in dealing with transactions’ priorities for real-time systems [18, 19].

Forward validation, on the other hand, checks conflicts against active transactions [37]. Thus, when a conflict is detected, transactions to be aborted can be freely selected according to their priorities. If a validating transaction is selected to commit, it broadcasts its commitment to all active transactions that conflict with it and aborts them immediately. This method is also known as the broadcast commit (OCC-BC) variant [18] of the original OCC protocol.
Haritsa et. al [15] and Lin et. al. [31] introduced a *priority wait* mechanism to the OCC-BC protocol. If a validating transaction has higher priority than active transactions that conflict with it, it commits. All conflicting transactions are aborted. Otherwise, it is made to wait and not allowed to commit immediately. Deferring the commitment of a lower priority transaction gives higher priority transactions a chance to meet their deadlines first. Moreover, if transaction A conflicts with transaction B, it does not necessarily mean that the converse is true [15]. Hence, letting a validating transaction which has lower priority wait for the completion of higher priority transactions may eliminate some conflicts and allow all of the transactions to commit. However, the validating transaction may also develop new conflicts while it is made to wait, leading to more restarts.

### 2.2.5 Speculative Real-Time Concurrency Control Protocol

Unlike blocking-based pessimistic protocols, optimistic protocols rely on restarting transactions that violate correctness criteria. However, the cost of restarting transactions can be very high, especially when the restarted transactions are long transactions.

To reduce the impact of restarting transactions, Bestavros proposed a *speculative concurrency control* (SCC) protocol [6]. The basic idea of SCC is to create a standby process for a read operation, and use the standby process to speculate a possible conflict on the read. If later a write operation from another transaction conflicts with the read operation, the standby process replaces the original transaction and resumes its execution from the point the conflicting read occurred. In other words, under SCC, there is no need to re-execute the entire transaction. This shortened re-execution time may increase the chance of meeting transaction deadlines. However, the cost of forking speculative processes can be very high.
2.3 Epsilon Serializability

*Epsilon serializability* allows transactions to exhibit a user-specified temporary and bounded inconsistency [38]. The measure of the inconsistency of a transaction is based on the number of non-serializable conflicts experienced by the transaction. *Imported* inconsistency of an active query transaction measures the number of uncommitted objects that have been read by the transaction and the number of objects that have been changed after the transaction read them. *Exported* inconsistency of an active update transaction measures the number of uncommitted objects that have been produced by it and have been read by query transactions. A history is epsilon serializable (ESR) if (1) for each transaction, the importing and exporting inconsistencies are less than specified limits and (2) the history is serializable after removing query transactions.

ESR should allow more concurrency than serializability because (1) query transactions can read limited amounts of inconsistent data, and (2) updated transactions can modify data that have been read by query transactions. However, updated transactions are still required to be serializable.

2.4 Similarity

The protocols we have discussed so far are all based on serializability as their correctness criterion. Serializability requires transactions to wait for an up-to-date value in order to deliver correct results. Unfortunately, this criterion is unnecessarily restrictive for real-time systems. In many real-time systems, such as avionic systems, manufacturing control systems, and multimedia systems, data values recorded by sensors change periodically and changes among values over a few consecutive periods are often considered to be insignificant. Hence, data values produced within a short
interval, called a similarity bound (sb) [25], can be treated as if they are “similar” and interchangeable.

Kuo and Mok formalized this concept and introduced various similarity-based correctness criteria in [24]. In the following subsections, we introduce the definition of similarity. We then discuss two more relaxed criteria, view $\Delta$-serializability and conflict $\Delta$-serializability.

### 2.4.1 Definition of Similarity

Similarity is defined as a binary relation on the domain of a data object [24]. Every similarity relation is reflexive and symmetric, but not necessarily transitive. As pointed out in [24], two views of a transaction are similar if and only if all the values read by the transaction in both views are similar. Two values of a data object are similar if all transactions that may read them consider them as similar. Moreover, two events in a schedule are similar if they are of the same type and access similar values of the same data object. Finally, two database states are similar if the corresponding values of every data object in the two states are similar.

There are two restrictions on similarity relations [24]. The first requires that every similarity relation must be preserved by all transactions. This means that if the input state for a transaction $\tau_{i,j}$ is either $a$ or $a'$, where $a$ is similar to $a'$, then $\tau_{i,j}$ must finish its job at a system state $b$ or $b'$, where $b$ and $b'$ are also similar. A similarity relation that satisfies this requirement is called as a regular similarity relation [24]. A regular similarity relation assures that a transaction always produces similar results if the input values given for each execution are similar.

The second restriction is called strong similarity [24]. Strong similarity allows similar events in a schedule to be swapped two or more times and guarantees that similarity will still be preserved in the output. More formally, let $\Delta$ be a similarity
relation and $\Delta^\#$ be a power of $\Delta$. Then the similarity relation $\Delta$ is a strong similarity relation for a schedule $\pi$ if $\forall \tau_{i,j} \in \pi, \tau_{i,j}$ preserves $\Delta^\#$. A similarity relation $\Delta^\#$ is preserved when the input state for a transaction $\tau_{i,j}$ is either $a$ or $a'$, where $a$ is similar to $a'$ under $\Delta^\#$, then $\tau_{i,j}$ finishes its job in a system state $b$ or $b'$, where $b$ and $b'$ are also similar under $\Delta^\#$. Note that since $\Delta \subseteq \Delta^\#$, all transactions that preserve $\Delta^\#$ must also preserve $\Delta$.

2.4.2 View Similarity and View $\Delta$-Serializability

Suppose we swap the events in a schedule $\pi$ and obtain another schedule $\pi'$. The schedule $\pi'$ is called as a derived schedule of $\pi$ if (1) for every read event $r$, the two write events from which $r$ reads in both schedules are similar under $\Delta$ in $\pi$, and (2) the final values of every object produced in both schedules are also similar under $\Delta$ in $\pi$. Based on this definition, Kuo, et al. proved the following theorem [24]:

**Theorem 2.4.1** If one schedule is a derived schedule of another schedule, then (1) for every read events $r$, the two write events from which $r$ reads in both schedules must be similar under $\Delta^\#$, and (2) the final values of every object produced in both schedules must be similar under $\Delta^\#$.

If a schedule $\pi'$ is a schedule derived from another schedule $\pi$, $\pi'$ is said to be "view-similar" to $\pi$ under $\Delta^\#$. A schedule preserves view-$\Delta$-serializability if and only if it is view-similar to a serial schedule. The schedule is also referred to as a view $\Delta$-serializable schedule. A view $\Delta$-serializable schedule may or may not preserve view-serializability [36]. However, a schedule that preserves view-serializability must preserve view-$\Delta$-serializability [24].

**Example 2.4.1** Assume we have two transactions $\tau_i$ and $\tau_j$ with the periods $p_i$ and $p_j$, respectively. Let $\tau_i$ be an update transaction that updates objects $x$ and $y$, and let $\tau_j$
be a query transaction that reads objects \( x \) and \( y \). Assume the values of object \( x \) (and \( y \)) are similar if they are created within two consecutive periods \((2 \ast p_i)\). Figure 2.1 gives four different histories \((H_1-H_4)\) of executions over \( \tau_i \) and \( \tau_j \). We use \( W_{t,n}^{x,y} \) and \( R_{t,n}^{x,y} \) to represent the write and the read events issued by the \( n \)th transaction instance of \( \tau_i \), respectively. The superscript of a read or write event indicates the object the event accesses.

![Figure 2.1: Examples of view-similarity.](image)
Both $H_1$ and $H_2$ of figure 2.1 are serial histories. In $H_1$, $\tau_{j,1}$ reads the values of $x$ and $y$ that are both created by $\tau_{i,1}$ ($\tau_{j,1}$ reads the value of $x$ created by $w_{i,1}^x$ and $\tau_{j,1}$ reads the value of $y$ created by $w_{i,1}^y$). In $H_2$, $\tau_{j,1}$ reads the values of $x$ and $y$ that are both created by $\tau_{i,2}$ ($\tau_{j,1}$ reads the value of $x$ created by $w_{i,2}^x$ and $\tau_{j,1}$ reads the value of $y$ created by $w_{i,2}^y$).

History $H_3$ is neither a serial history nor a serializable history. However, $H_3$ is view similar to $H_1$. To see the reason, let’s compare the values of $x$ and $y$ read by $\tau_{j,1}$ in both histories $H_1$ and $H_3$. For object $x$, $\tau_{j,1}$ reads the value of $x$ created by $w_{i,1}^x$ of $\tau_{i,1}$ in both histories $H_1$ and $H_3$. Thus, $\tau_{j,1}$ of $H_1$ and $H_3$ have the similar (same) views on object $x$. For another object $y$, $\tau_{j,1}$ reads the value of $y$ created by $w_{i,1}^y$ of $\tau_{i,1}$ in the history $H_1$. However, in the history $H_3$, $\tau_{j,1}$ reads the value of $y$ created by $w_{i,2}^y$ of $\tau_{i,2}$. Since the temporal distance between $w_{i,1}^y$ and $w_{i,2}^y$ is less than the similarity bound of object $y$ ($sb_y = 2p_1$), the values created by $w_{i,1}^y$ and $w_{i,2}^y$ are similar. Hence, $\tau_{j,1}$ of $H_1$ and $H_3$ also have the similar views on object $y$. Since all read operations of $\tau_{j,1}$ in both histories have similar views and the final views of both histories are the same (the final views of $x$ and $y$ are both created by $\tau_{i,5}$ in $H_1$ and $H_3$), the history $H_3$ is view similar to the serial history $H_1$. For the similar reason, $H_3$ is also view similar to the serial history $H_2$.

In history $H_4$, $\tau_{j,1}$ reads the value of $x$ created by $w_{i,1}^x$ of $\tau_{i,1}$ in both histories $H_1$ and $H_4$. Thus, $\tau_{j,1}$ of $H_1$ and $H_4$ have the similar (same) views on object $x$. For another object $y$, $\tau_{j,1}$ reads the value of $y$ created by $w_{i,1}^y$ of $\tau_{i,1}$ in the history $H_1$. However, in the history $H_4$, $\tau_{j,1}$ reads the value of $y$ created by $w_{i,5}^y$ of $\tau_{i,5}$. Since the temporal distance between $w_{i,1}^y$ and $w_{i,5}^y$ is greater than the similarity bound of object $y$ ($sb_y = 2p_1$) in $H_4$, the values created by $w_{i,1}^y$ and $w_{i,5}^y$ are not similar. Hence, $H_4$ is not view-similar to the serial history $H_1$. In fact, we will get the same result after comparing $H_4$ with all other possible serial histories.
2.4.3 Conflict Similarity and Conflict $\Delta$-Serializability

Let $E$ be a set that contains all operations performed in a history $\pi$, and let $\Delta$ be a strong similarity relation. A $\text{free}(\pi)$ relation is defined over $E$ in $\pi$ as a set $\{(e_1, e_2)\}$ of swappable operation pairs in which $e_1$ and $e_2$ satisfy one of the following conditions [25, 26]:

1. $e_1$ and $e_2$ do not conflict,

2. $e_1$ and $e_2$ are conflicting write operations, but they are similar under $\Delta$,

3. $e_1$ precedes $e_2$, where $e_1$ and $e_2$ are read and write operations, respectively, and $e'$, the write operation from which $e_1$ reads, is similar to $e_2$ under $\Delta$.

A history $\pi'$ is conflict similar to $\pi$ if $\pi'$ can be obtained from $\pi$ by swapping consecutive operations defined in $\text{free}(\pi)$. In other words, $\pi$ is conflict similar to $\pi'$ if and only if every two conflicting operations $e$ and $e'$ occur in the same order in $\pi$ and $\pi'$ unless $e$ and $e'$ satisfy the $\text{free}(\pi)$ relation. Obviously, if a history is conflict equivalent to another history, it is also conflict similar to that history. But, the converse may not hold.

A history is conflict $\Delta$-serializable if and only if it is conflict similar to a serial history. For instance, in the Example 2.1, $\text{free}(H_3) = \{(w_{t,1,i}, w_{t,2,i}), (w_{t,1,i}, w_{t,2,i}), (r_{j,1,i}, w_{i,2,i}), (r_{j,1,i}, w_{i,2,i}), ...\}$. By swapping $r_{j,1,i}$ with $w_{t,2,i}$, and then swapping $r_{j,1,i}$ with $w_{i,2,i}$, we obtain the serial history $H_2$. Hence, $H_3$ is conflict $\Delta$-serializable to $H_2$. However, $H_3$ is not conflict $\Delta$-serializable to $H_1$ because there is no way to swap operations of $H_3$ and obtain $H_1$ by following the $\text{free}(H_3)$ relation.

Kuo, et. al. proved that every conflict $\Delta$-serializable history is also view $\Delta$-serializable [24, 25, 26]. However, as the history $H_5 = r_{j,1,i} w_{t,1,i} w_{t,2,i} w_{t,3,i} w_{t,4,i}$ suggests, a view $\Delta$-serializable history is not necessarily conflict $\Delta$-serializable. $H_5$ is view
\( \Delta \)-serializable to \( r_{i_1}^{\alpha} w_{i_1}^{u_1} w_{i_2}^{x_2} w_{i_3}^{y_3} w_{i_4}^{y_4} \). But there is no \( \text{free}(H_5) \) relation from which we can obtain a serial history.

In order to determine whether a history \( \pi \) is conflict \( \Delta \)-serializable, Kuo and Mok defined a transaction dependency graph \( TG(\pi) \) [24, 25, 26] as a directed graph whose nodes are committed transaction instances in \( \pi \) and whose edges are all \( \tau_{i,j} \rightarrow \tau_{k,l} \) such that there are two conflicting operations \( e \) and \( e' \) in \( \tau_{i,j} \) and \( \tau_{k,l} \), respectively, which do not satisfy the \( \text{free}(\pi) \) relation and \( e \) precedes \( e' \) in \( \pi \).

**Theorem 2.4.2** A history \( \pi \) is conflict \( \Delta \)-serializable if and only if \( TG(\pi) \) is acyclic [24, 25, 26].

### 2.4.4 Similarity Stack Protocol (SSP)

To utilize the concept of similarity, Kuo and Mok also developed a real-time concurrency control protocol, the similarity stack protocol (SSP) [25] that preserves view \( \Delta \)-serializability. The result shown in [25] indicates that the concept of similarity and SSP can increase system concurrency. The reason is, under similarity, a transaction can complete its computation using an older data value without being blocked to wait for the newest one. The increased concurrency can further help more transactions to meet their deadlines. In this subsection, we will explain the basic idea of SSP and discuss its disadvantages.

Under SSP, transactions are normally scheduled according to their priorities, with the restriction that the stack discipline must be imposed. To enforce the stack discipline, assume a system contains a preemption stack, and each transaction that is allowed to start is pushed on to the top of the stack. At any time, the system executes only the transaction sitting on the top of the stack. A transaction immediately below the top of the stack is said to be preempted by the transaction on the top of the stack. After the transaction on the top of the stack finishes, it is popped from the
stack and the transaction below it resumes its execution. Obviously, the preemption stack contains all started but uncommitted transactions, and the execution of those transactions are all overlapped. The sum of the execution time of all transactions in the stack is called as the temporal depth [25] of the stack.

If two overlapping transactions conflict and the conflict cannot be resolved by similarity, then the conflicting events are not interchangeable. As a result, the schedule may not be A-serializable. Hence, the major idea of SSP is to bound the temporal depth of the preemption so that conflicting events among overlapping transactions will be close enough in time to be resolved by similarity. The bound to be enforced by SSP is referred to as a recency bound in [25]. Before we explain the SSP rules that control the growth of the temporal depth, we shall first discuss the relationship between the similarity bound and the recency bound.

Suppose \( sb_x \) is the similarity bound for an object \( x \). Let us denote the maximum temporal distance between two write events on \( x \) which belong to overlapping transactions as \( \omega_x \). \( \omega_x \) must be such that \( \omega_x \leq sb_x \). Otherwise, two write events of overlapping transactions will not be similar and interchangeable. On the other hand, let \( \delta_x \) be the maximum temporal distance between a read and a write event on \( x \) which belong to overlapping transactions. \( \delta_x \) must obey \( \delta_x \leq sb_x - 2p_x \), where \( p_x \) is the smallest period of all transactions that may update \( x \).

Figure 2.2 illustrates the reason for this restriction. If the read event \( (r_x) \) occurs before the write event \( (w_x) \) and we want to swap these two events, then another write event \( (w'_x) \) that \( r_x \) reads from must be sufficiently close to \( w_x \) so that \( w_x \) and \( w'_x \) are similar. In other words, for \( w'_x \) and \( w_x \) to be similar, the greatest temporal distance between \( w'_x \) and \( w_x \) cannot be greater than \( sb_x \). Since the greatest temporal distance between \( w'_x \) and \( r_x \) is \( 2p_x \), and the greatest temporal distance between \( w'_x \) and \( w_x \) is \( sb_x \), \( \delta_x \) (the temporal distance between \( r_x \) and \( w_x \)) is constrained as \( \delta_x \leq sb_x - 2p_x \).
The recency bound for object \( x \), denoted as \( \alpha_x \), is then assigned as \( \alpha_x = \min\{\omega_x, \delta_x\} = sb_x - 2p_x \).

As we discussed before, the major idea of SSP is to control the temporal depth of a preemption stack under a recency bound so that conflicts among overlapping transactions can be resolved by similarity. Hence, the recency bound for an entire system must be the smallest recency bound of the objects in the system. However, the smaller the recency bound is, the smaller the temporal depth can grow, and a smaller temporal depth allows fewer overlapped transactions and lower concurrency. To increase the concurrency, objects are divided into interactive sets, each of which has its own recency bound. Two objects belong to one interactive set if transactions that may access them conflict. For example, objects \( x \) and \( y \) belong to one interactive set if both transactions \( i \) and \( j \) write to \( x \) (i.e. transactions \( i \) and \( j \) conflict), and one of these transactions accesses \( y \). Conversely, two objects belong to two different interactive sets if no transactions that may access them conflict. The recency bound \( \alpha_f \) of the \( f \)th interactive set is the minimum recency bound of objects in the \( f \)th set.
Since each interactive set has its own recency bound, the preemption stack can grow as long as the temporal depth accumulated on individual interactive set is smaller than the recency bound of the corresponding set. The temporal depth of an interactive set $f$ is the sum of the execution time of all transactions in the preemption stack that belong to the interactive set $f$. This basic rule guarantees that conflicts among overlapping transactions in any interactive set can be resolved by similarity. On a multiple processor system in which each processor has its own preemption stack, this rule can also be applied separately on each processor to prevent conflicts on individual processors. However, the sum of the temporal depth of an interactive set $f$ on all processors may still be greater than $\alpha_f$, making conflicts among overlapping transactions executed on different processors unresolvable by similarity. We use the following figure to illustrate this situation. In Figure 2.3, $\tau_{i,j}$ and $\tau_{k,l}$ represent the $j$th and the $l$th instances of transactions $\tau_i$ and $\tau_k$, respectively. Assume $\tau_{i,j}$ and $\tau_{k,l}$ are both in the $f$th interactive set. If the transaction $\tau_{m,n}$ is allowed to start and preempt $\tau_{k,l}$ on $P_1$ at time $t$ because the temporal depth of $f$th interactive set on the

![Figure 2.3: Calculation of recency bound on a multi-processor system.](image-url)
Processor $P_1$ is still less than the recency bound $\alpha_f$, then the total temporal depth of the $f$th interactive set on all processors can grow longer than $\alpha_f$. Hence, the safest way is to prevent the temporal depth of the two interactive sets $f$ on each processor from exceeding $\lfloor \frac{\alpha_f}{2} \rfloor$.

Next, we explain the SSP rules for controlling the growth of the temporal depth of each interactive set on every processor. Let us use $\text{accu}_{P,f}$ to denote the accumulated temporal depth of the $f$th interactive set on the $P$th processor, and let $\text{rnp}_{P,f}$ denote the number of uncommitted transactions in the $f$th interactive set on the $P$th processor. A transaction $\tau_{i,j}$ with the worst case execution time $w_{c_{i,j}}$ in the $f$th interactive set can be started on a processor $P$ if the following four SSP rules are all satisfied:

1. $\tau_{i,j}$ has higher priority than all unscheduled transactions across all processors and has higher priority than all scheduled transactions on Processor $P$,

2. $\text{accu}_{P,g} + w_{c_{i,j}} \leq \alpha_g$ for any $g$ (including $f$) such that $\text{rnp}_{P,g} \geq 1$,

3. $w_{c_{i,j}} \leq \alpha_f$ for any $P' \neq P$ such that $\text{rnp}_{P',f} \geq 1$, and

4. $\text{accu}_{P',f} \leq \alpha_f$ for any $P' \neq P$ such that $\text{rnp}_{P',f} \geq 1$.

The second SSP rule above controls the growth of the temporal depth of interactive sets so that the conflicting events of overlapping transactions will be sufficiently close in time. The third and fourth SSP rules are used to deal with the boundary condition of the second rule in a multi-processor system. If $\tau_{i,j}$ is allowed to start on the $P$th processor, then $\text{accu}_{P,g} = \text{accu}_{P,g} + w_{c_{i,j}}$ is performed for any interactive set $g$ (including $f$) such that $\text{rnp}_{P,g} \geq 1$. $\text{rnp}_{P,f}$ is also incrementated by 1. These computations reflect the growth of temporal depth of interactive sets. When $\tau_{i,j}$ finishes its job on
the processor $P$, $rn_{P,f}$ is decremented by 1. If $rn_{P,f} = 0$, $accu_{P,f}$ is reset to 0. Note that if $sb_x$ of an object $x$ in the $f$th interactive set is such that $sb_x \leq 2p_x$, SSP allows only exclusive access to the object $x$ because $\alpha_f \leq 0$.

Finally, SSP preserves view $\Delta$-serializability. For the detailed proof, readers are referred to [25].

2.4.5 Disadvantages of SSP

From the last subsection, we see that SSP requires two pieces of information: (1) the estimated resource access pattern of transactions, and (2) the estimated WCET of transactions. The resource access pattern is used to construct interactive sets and to calculate the recency bounds of these sets. The WCET is used by the second SSP rule to control the growth of a preemption stack.

If a protocol relies on prior knowledge of WCET and data access requirements, it becomes less scalable [40]. This is because restructuring application transactions would require re-evaluating all the transactions' WCETs and data access patterns, and could require rebuilding the system-level scheduling. In other words, the impact of changing transaction-level information is not restricted to the transaction level. Finally, since calculation of WCET is highly dependent on the architecture of the target machine on which a system is to run, WCET must be recalculated each time the system is to be moved onto machines with different architectures.

Recall that the recency bound of an object $x$ is $\alpha_x = \min\{\omega_x, \delta_x\} = (sb_x - 2p_x) / 2$, and the recency bound of an interactive set $I_g$ is $\alpha_g = \min\{\alpha_x\} \forall x \in I_g$. Setting $\alpha_x = sb_x - 2p_x$ assumes that the temporal distance between a read operation and a write operation from which the read operation reads from is always $2p_x$, the worst case situation. Dividing $(sb_x - 2p_x)$ by 2 is another worst case assumption that assumes transactions running on different processors always conflict. Combining these worst
case assumptions, $\alpha_x$ of an object $x$ is minimized, which in turn, leads to a smaller recency bound and may lower concurrency in an interactive set. In fact, if there is an object $x$ with similarity bound $sb_x \leq 2p_x$, the recency bound of the interactive set which $x$ belongs will become 0. In this situation, all transactions in the interactive set can only be executed serially.

Moreover, due to data-dependent loops and conditional branches in each program, the WCET and the objects accessed by each transaction are likely to be overestimated [13, 46]. As a result, a protocol which relies on this overestimated information may under-utilize system resources and may reject feasible schedules. We use the following example to illustrate this situation.

**Example 2.4.2** Assume a system is organized as shown in figure 2.4. The system contains three transactions, $\tau_1, \tau_2, \tau_3$, and two objects, $x$, and $y$.

![Figure 2.4: System organization of Example 2.4.2.](image)

Transactions $\tau_1$ and $\tau_2$ are update transactions that contain one single write operation to update objects $x$ and $y$, respectively. Transaction $\tau_3$ is a query transaction and contains the following operations: $r^x \ c \ r^y \ cc$. The symbol $c$ represents a time unit of computation, and $r^x$ and $r^y$ are read operations that read objects $x$ and $y$, respectively. The period ($p_i$), the exact execution time ($c_i$), the estimated WCET
(wc_i), and the system utilization (ui) of the three transactions in this example are given in the table 2.1. We assume the WCET of \( \tau_1 \) and \( \tau_2 \) are perfectly estimated and are exactly the same as their real execution times. The WCET of \( \tau_3 \), however, is overestimated by only one time unit. The similarity bounds of \( x \) and \( y \) are set to \( 5p_1 \) and \( 4p_2 \), respectively. The execution time-lines of these transactions under the controls of SSP are given in figure 2.5.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>( p_i )</th>
<th>( c_i )</th>
<th>wc_i</th>
<th>( u_i = wc_i/p_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0.17</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0.333</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>15</td>
<td>5</td>
<td>6</td>
<td>0.4</td>
</tr>
<tr>
<td>system utilization</td>
<td></td>
<td></td>
<td></td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 2.1: Transaction attributes of Example 2.4.2

Figure 2.5: Time-lines for executing Example 2.4.2

In Example 2.4.2, since both \( \tau_1 \) and \( \tau_2 \) may conflict with \( \tau_3 \), these three transactions belong to one single interactive set \( j \) according to SSP rules. The recency bound
for object $x$, $\alpha_x = sb_x - 2p_x = 5p_1 - 2p_1 = 18$, and the recency bound for object $y$, $\alpha_y = sb_y - 2p_y = 4p_1 - 2p_1 = 6$. Note that $p_x$ and $p_y$ are the smallest periods among the update transactions of objects $x$ and $y$, respectively. As a result, the recency bound of the interactive set $j$, $\alpha_j = \min \{\alpha_x, \alpha_y\} = 6$.

At time 3 of $H_1$ in figure 2.5, $accu_{i,j}$ is set to 6 because it is estimated that $\tau_{3,1}$ needs 6 time units to complete its job. At time 4, when $\tau_{2,2}$ becomes ready, it is unable to preempt $\tau_{3,1}$ because $accu + c_{2,2} = 6 + 1 = 7 > \alpha_j = 6$. Hence, $\tau_{3,1}$ finishes its job at time 7 without being preempted. Unfortunately, it is too late for $\tau_{2,2}$ to execute. Similarly, $\tau_{2,7}$ also misses its deadline at time 21. Hence, two transaction instances miss their deadlines in every major cycle. However, if all WCETs are perfectly estimated as the exact execution times, SSP allows the concurrent history $H_2$ as shown in figure 2.5 and no transactions miss their deadlines.

The disadvantages of SSP and the advantages of similarity motivate us to search for solutions that can better utilize similarity without relying on prior information of transactions. In the following chapters, we will present our approaches and the discuss the reasons of taking such approaches.

2.5 Real-time Simulator

The performance of real-time protocols is usually studied through simulations. Hence, various real-time simulators have been developed. Since these simulators are frequently designed to study some particular protocols, they consider only subsets of real-time problems and do not provide flexibility for experimenting with new protocols. In this section, we review several existing real-time simulators.

Tokuda and Kotera at Carnegie Mellon University developed a tool set [47] for handling timing issues in designing real-time software. The major tools include a real-time monitor/debugger, called Advanced Real-Time Monitor (ARM), and schedula-
bility analyzer—Scheduler 1-2-3. The ARM provides real-time visualization of a system’s runtime behavior. The Scheduler 1-2-3 verifies whether the set of real-time transactions can meet their deadlines under specific scheduling policy. However, it provides only a few scheduling algorithms, such as the rate monotonic, the FIFO, and the EDF. There is no indication how different scheduling algorithms can be constructed and experimented. Furthermore, this analyzer does not consider the concurrency control issue.

The research group in real-time systems at the University of York, UK, has taken an approach similar to our own in its STRESS system [4]. This simulator incorporates a flexible front end for easy description of real-time applications in terms of timing behaviors and interactions. The target of this simulation, however, is hard real-time systems only. Concurrency control in the simulator is limited to ad hoc use of transaction precedence plus synchronization and communication primitives. With such limited and non-extensible primitives, there is no way to simulate the very common concurrency control protocols like the read-lock and the optimistic protocols.

Abbott and Garcia-Molina [1] reported on the effects of concurrency control, scheduling, and overload management policy on real-time performance. Their work explored real-time database performance and simulated only serializable executions. Kuo and Mok [25] and Song [42] have also used simulation to investigate specific protocols that relax consistency constraints for real-time executions. However, these simulations were designed to evaluate specific protocols and are not directly adaptable for alternative approaches.
Chapter 3

THE MODELS

The blocking mechanism used by SSP relies on many worst case assumptions, i.e. estimated WCET, data access pattern, the worst case read/write distance, and the smallest object recency bound for interactive sets. Combining all these worst case assumptions inevitably makes SSP very conservative, reducing possible concurrency.

Optimistic approaches can permit higher concurrency, and they do not rely on prior knowledge of WECT or data access patterns. Therefore, in this thesis, we investigate how to integrate the concept of similarity into the optimistic approach. The first section of this chapter describes the transaction model for the optimistic approach. The information maintained by each object is then described in the object model. Finally, we discuss the system model studied in this thesis.

3.1 Transaction Model

We assume that a system consists of a set of periodic transactions. Each transaction \( \tau_i \) is defined as a 4-tuple \( < OP_i, \prec_i, p_i, d_i > \), where:

- \( OP_i \) contains a set of operations,
- \( \prec_i \) specifies a partial order among operations in \( OP_i \),
- \( p_i \) is the period length of \( \tau_i \) that specifies the interval at which the transaction \( \tau_i \) will be invoked (for aperiodic transactions, \( p_i \) is null),
- $d_i$ is the deadline length of $\tau_i$.

When a transaction $\tau_i$ is invoked, a transaction instance of $\tau_i$ is created. $\tau_{i,j}$ represents the $j$th instance of transaction $\tau_i$. A transaction instance $\tau_{i,j}$ is also defined as a 4-tuple $< OP_{i,j}, p_{i,j}, d_{i,j} >$, where each tuple element is defined as the corresponding tuple element of $\tau_i$. Each transaction instance $\tau_{i,j}$ also maintains the following data structures:

- $RS_{i,j}$ is the read set of $\tau_{i,j}$, the objects that are read by $\tau_{i,j}$,

- $WS_{i,j}$ is the write set of $\tau_{i,j}$, the objects that are updated (written) by $\tau_{i,j}$.

Since transactions may be aborted, transactions work on their own local buffers to hide the effects of aborts and failures. Hence, a transaction is run in its entirety or not at all. No transaction’s write phase is left only partly executed. This property is called atomicity and it is essential in preventing cascading aborts.

As in ordinary optimistic approaches, the execution of a transaction instance in our model can be divided into a computation phase, a validation phase, and an optional write phase. Execution of a transaction instance $\tau_{i,j}$ starts with a computation phase in which the read, write, and computation operations are performed. Each object read (written) by $\tau_{i,j}$ is added to the read set (the write set) of $\tau_{i,j}$. The write operations performed in a computation phase update the local buffer of $\tau_{i,j}$. We refer to such writes as local writes. $\tau_{i,j}$ is a query transaction if no write operation is issued during its computation phase. Otherwise, $\tau_{i,j}$ is an update transaction. We say $\tau_{i,j}$ is an active transaction when it is in its computation phase. After $\tau_{i,j}$ finishes its computation phase, it enters its validation phase and is referred to as a validating transaction.
In order to detect violations of correctness criteria, a validating transaction performs appropriate validation operations according to the specified criteria. If a violation is detected, either the validating transaction or another active transaction involved in the violation is aborted. Otherwise, the validating transaction enters its write phase, during which the local updates are made accessible to other transaction instances. We refer to the write operations performed in the write phase as committed writes. After the write phase, the transaction issues a commit operation and the transaction instance becomes a committed transaction.

In this thesis, the read, local write, computation, validation, commit, committed write, and the abort operations performed by $\tau_{i,j}$ are represented as $r_{i,j}^x$, $w_{i,j}^x$, $c_{i,j}$, $v_{i,j}^x$, $\hat{c}_{i,j}$, $\hat{w}_{i,j}^x$, and $a_{i,j}$ respectively. The superscript $x$ stands for the object on which the operation operates. Note that each transaction instance eventually issues either a commit or an abort operation, but not both.

3.2 Object Model

We assume a real-time system contains a collection of data objects. Each object $x$ contains: $ts_x$, which is a timestamp that records the most recent time at which the object $x$ is updated by a local write of a periodic transaction instance, and $sb_x$, the similarity bound of an object $x$, a time interval within which all local writes issued by different transaction instances on the object $x$ are considered to be similar.

The timestamp of an object value is assigned as the time a local write to the object is issued. The timestamp of the object value is updated only when the committed write of the issuing transaction takes place. We do not use the time at which the committed write occurs as the timestamp of an object value because doing so may artificially delay the creation time of an object value. More formally, let $time(op_{i,j})$ be a function that returns the time the operation $op_{i,j}$ occurs in a system, we have
Definition 3.2.1

\[
\begin{align*}
ts(\text{op}_{i,j}^x) = \begin{cases} 
time(\text{op}_{i,j}^x), & \text{if } \text{op}_{i,j}^x \neq \text{op}_{i,j}^z \\
ts_z = \time(\text{op}_{i,j}^z), & \text{if } \text{op}_{i,j}^x = \text{op}_{i,j}^z
\end{cases}
\end{align*}
\]

3.3 System Model

We focus our attentions on firm real-time systems in which a transaction that misses its deadline is aborted immediately. We define a real-time system \( S \) as a 3-tuple \( <T_S, O_S, CPU_S> \), where

- \( T_S \) is a set of transactions in \( S \),
- \( O_S \) is a set of objects in \( S \), and
- \( CPU_S \) is a set of CPUs contained in \( S \). \( CPU_S = \{P_k \mid k = 1..\eta\} \), where \( P_k \) indicate the \( k \)th CPU of \( S \) and \( \eta \) is the total number of CPUs in \( S \).

If \( \eta > 1 \) in \( S \), \( S \) is a multi-processor system in which we assume that a shared memory architecture is used so that there are no communication delays. When \( \eta > 1 \), different transaction instances can be executed concurrently on different processors. However, our model does not allow operations of one transaction instance to be executed concurrently on multiple processors. Also, once a transaction instance is started on one particular processor, the transaction instance cannot migrate to another processor.

Finally, we define an execution history \( H_S \) of the system \( S \) as a partial order of operations issued by the instances of the transaction set \( T_S \) [24]. It is common to write a partial order as an ordered pair \((\Sigma, \prec_S)\), where \( \Sigma \) is the set of operations performed by all transaction instances of \( S \), and \( \prec_S \) indicates the execution order of those operations [5].
Chapter 4

NEW SIMILARITY-BASED PROTOCOLS

Since optimistic approaches can allow higher concurrency [42, 18, 19, 22], it makes sense to combine similarity-based methods with optimistic approaches. In addition to increased concurrency, another important property of a real-time system is its predictability: A system's functional and timing behavior should be as deterministic as is necessary to satisfy the system specification [45]. Hence, before applying optimistic approaches to real-time systems, transaction operations and their interactions must be well understood, bounded, and predictable. In this chapter, we will discuss several new approaches toward making optimistic concurrency control a more deterministic protocol.

In Chapter 4.1, we review two optimistic validation mechanisms, forward validation and backward validation. Chapter 4.2 then lists the concerns that need to be addressed when applying optimistic concurrency control to real-time systems. In Chapters 4.3 and 4.4 we integrate the concept of similarity into the forward and backward validation mechanisms. We conclude this chapter with a description of a new protocol whose behavior is predictable.

4.1 Optimistic Protocols: Forward and Backward Validations

Several optimistic protocols have been proposed [23, 16, 37, 18, 19]. Their validations are either forward validation or backward validation. The two validation methods differ in when and how conflicts are detected. In forward validation, if the
write phases of transactions are not allowed to overlap, a validating transaction validates its write set against the read sets of other concurrently active transactions. In backward validation, a validating transaction is first assigned an incremented transaction number. The validating transaction then validates its read set and write set against the write sets of all transactions that have smaller transaction numbers and that overlap with the validating transaction.

Under forward validation, since the validating transaction and the transactions to be checked by the validating transaction have not yet committed, either the validating transaction or conflicting active transactions can be aborted. This feature has an interesting implication for real-time systems because transactions to be aborted can be freely selected based on their priorities [18, 19]. Another advantage of forward validation is that it can detect conflicts earlier than in backward validation, which must wait until a transaction is about to commit. On the other hand, under backward validation, the transactions to be checked by the validating transaction may have committed. Hence, if a conflict occurs, only the validating transaction can be restarted.

4.2 Applying Optimistic Approaches in Real-Time Systems

The first major concern of adopting optimistic approaches in real-time systems is that optimistic protocols have high risk of restarts that can increase the likelihood of transactions missing deadlines. However, under the concept of similarity, fewer transactions may need to be restarted because more conflicts can be resolved by similarity, especially when similarity bounds are increased. (See Chapter 6 for simulation results). This feature of similarity should reduce the impact of roll-backs and make optimistic approaches more attractive and practical for real-time systems.
Under an optimistic approach, a validating transaction must validate the objects it accesses against the objects accessed by other overlapping transactions. Because the number of overlapping transactions and the number of objects accessed by these transactions are dynamic, it is difficult to obtain the maximum validation time required by each transaction. Maintaining this dynamic information also requires additional interactions among transactions and the underlying system. For example, in both the forward validation and backward validation protocols, a validating transaction must access the read sets or write sets of overlapping transactions in order to perform the validation, and the system must keep track of transactions that overlap with the validating transaction. (For more detailed explanations, see Chapter 4.3 and Chapter 4.4.) The overhead imposed by these additional interactions could be substantial, and uncertainty is introduced.

Moreover, using optimistic approaches, conflicts are resolved by aborting and restarting conflicting transactions. If a transaction is restarted again and again due to its repeated conflicts with other transactions, the starvation problem occurs, introducing another uncertainty. Some solutions to this problem have been proposed [37, 19]. They rely on raising a transaction’s priority after the transaction has been repeatedly restarted. This approach is inadequate for real-time systems because two transactions that are restarted due to their conflicts with other transactions may conflict with each other, forcing one of them to be restarted again. Hence, the number of restarts a transaction may experience is still unbounded.

A practical concern in using optimistic approaches is that a newly created value may be overwritten by an old value. We explain the reason below. In an optimistic approach, a transaction that works in its computation phase updates objects only in its local buffer, and local updates are made available to other transactions only later in its write phase. If a transaction \( \tau_{ij} \) creates a value for \( x \) before another transaction
\(\tau_{k,l}\) in their computation phases, and \(\tau_{i,j}\) enters its write phase after \(\tau_{k,l}\) finishes its write phase, then \(\tau_{i,j}\) updates \(x\) with a value that is older than the current value updated by \(\tau_{k,l}\). Hence, an old value overwrites a more recent value. In many real-time systems like avionics systems, this race condition is not tolerable. Moreover, this race condition may result in violating a system correctness criterion (e.g. a cycle may be formed among transactions in serializability or similarity). We refer this problem as the data regression problem in our following discussions.

In conclusion, to apply the optimistic approach to real-time systems, the trade-offs between degree of concurrency, amount of system overhead, predictability, and consistency must be addressed. We summarize the issues that should be considered in the following list:

- Predictability,

- Validation overhead,
  - Messages exchanged among transactions,
  - Data items accessed by each validating transaction,
  - Data items kept in a system,

- Number of transaction restarts,

- Blocking time experienced by each transaction.

4.3 Similarity-based Optimistic Concurrency Control– Forward Validation (SOCC-FV)

Forward validation has the advantage of detecting conflict earlier than backward validation and it allows flexibility in selecting transactions to abort and restart.
Hence, we investigate how to integrate the similarity criterion into the forward validation approach.

4.3.1 Overview of SOCC-FV

Under ordinary forward validation, if the write set of a validating transaction $\tau_{k,l}$ intersects with the read set of a concurrently active transaction $\tau_{i,j}$, then a conflict occurs. Either $\tau_{i,j}$ or $\tau_{k,l}$ must be aborted to resolve the conflict. However, if the new data value to be created by $\tau_{k,l}$ is the same or similar to the data value read by $\tau_{i,j}$, then $\tau_{i,j}$ will produce similar results no matter which value $\tau_{i,j}$ reads. Hence, no transaction needs to be aborted.

In order to check if similarity exists between the value to be created by the validating transaction $\tau_{k,l}$ and the value read by $\tau_{i,j}$, $\tau_{k,l}$ must perform a similarity test to determine if $|ts(Wr_{\pi}(r^x_{i,j})) - ts(w^x_{k,l})| \leq sb_x$, in its validation phase. We use $Wr_{\pi}(r^x_{i,j})$ to denote a committed write operation $w^x_{e,f}$ ($e \neq i$ or $f \neq j$) from which $r^x_{i,j}$ reads in a history $\pi$ (e.g. $Wr_{\pi}(r^x_{i,j}) = w^x_{e,f}$ in a history $\pi$ if $w^x_{e,f}$ occurs before $r^x_{i,j}$, and no other committed write operation is issued between $w^x_{e,f}$ and $r^x_{i,j}$). $ts(op)$ is the timestamp we defined earlier.

Only active transactions that do not satisfy the similarity test must be restarted. When the similarity test fails, the validating transaction will create dissimilar values with respect to the values previously read by the active transactions, and active transactions that use dissimilar values may later produce dissimilar results. Hence, SOCC-FV will produce $\Delta$-serializable histories. (See proof in Chapter 4.3.2).

If the similarity bounds of all objects are set to zero, then $|ts(Wr_{\pi}(r^x_{i,j})) - ts(w^x_{k,l})| > sb_x$ is always true. That means, if a validating transaction $\tau_{k,l}$ is about to update an object that is currently being used by another active transaction $\tau_{i,j}$, then either $\tau_{i,j}$ or $\tau_{k,l}$ must always be restarted. This behavior is the same as the traditional
forward validation protocol like OCC-BC. Hence, when similarity is not applicable \((\forall x, sb_x = 0)\), SOCC-FV reduces to the ordinary forward validation protocol. It still accepts the conflict serializable histories that can be generated by OCC-BC.

On the first sight, SOCC-FV appears to be very appealing, since it generates both serializable and \(\Delta\)-serializable histories, and it does not rely on any prior information about transactions. Moreover, SOCC-FV also inherits the advantages provided by the traditional forward validation approach: detecting conflicts earlier and providing flexibility in selecting restarted transactions. However, in either a single-processor or a multi-processor system, since other transactions may run concurrently with the validating transaction, one critical point of the forward validation approach is that a validating transaction must validate against a dynamic set of active transactions and the dynamic read sets of those transactions. One solution to simplify the validation procedure is to quiesce system activity (all transactions other than the validating transaction are halted) [37].

In the following, we consider SOCC-FV in a single processor system and we use the disabling/enabling CPU preemption mechanism to halt all transactions other than the validating transaction. Procedure 4.3.1 gives the detailed operations performed in the validation and write phases of a validating transaction \(\tau_{v,k}\) under SOCC-FV.

In Procedure 4.3.1, after the validating transaction \(\tau_{v,k}\) disables the CPU preemption mechanism, it checks to see if its write set intersects with the read set of any transaction instance in \(\tau_a\), where \(\tau_a\) is a set that contains all active transaction instances when \(\tau_{v,k}\) enters its validation phase. If the read set of an active transaction intersects with the write set of \(\tau_{v,k}\), an additional similarity test is performed. The result of the similarity test is stored in a flag VALID. If VALID is FALSE, then a conflict resolution procedure is invoked. The conflict resolution procedure selects transactions to restart based on some customized rules (e.g. based on the transaction
Procedure 4.3.1

Validate Write($\tau_{v,k}$) {
    disable preemption;
    $VALID = TRUE$;
    $\forall \tau_{i,j} \text{ such that } (\tau_{i,j} \in \tau_{a}) \land (RS_{i,j} \cap WS_{v,k} \neq \phi)$ do
    $\forall x \in WS_{v,k}$ do /* loop for write set objects */
    if ($x \in RS_{i,j}$) $\land$ ($\mid ts(Wr_{\pi}(x_{i,j})) - ts(w_{v,k}) \mid > sb_x$) then
        $VALID = FALSE$;
    endif
    enddo
    enddo
    if ($VALID == FALSE$) then
        conflict_resolution();
    endif
    WritePhase($\tau_{v,k}$) /* write phase */
    commit;
    enable preemption;
}

Figure 4.1: The validation procedure performed in the SOCC-FV protocol.
priority). If the validating transaction is selected to restart, the CPU preemption mechanism must first be re-enabled. If the active transactions are selected to restart, the validating transaction may proceed to its write phase. After the validating transaction finishes its write phase, it commits and the CPU preemption mechanism is then re-enabled.

In the write phase of a transaction, all the updated values of objects in the write set of the transaction $\tau_{v,k}$ are made available to the system (see Procedure 4.3.2). However, in order to prevent the data regression problem in which an old value may overwrite a more recently created value, the Thomas' Write Rule (TWR) [5] is first checked before a transaction updates the object with the value created by its local write.

**Procedure 4.3.2**

```
WritePhase($\tau_{v,k}$) {
    $\forall x \in WS_{v,k}$ do /* write phase */
    if ($ts(w_{v,k}^x) > ts_x$) then /* Thomas' Write Rule */
        write $x$;
    endif
    enddo
}
```

Figure 4.2: The write phase procedure that avoids the data regression problem.

Since the Thomas' Write Rule (TWR) says that processing a sequence of writes in timestamp order produces the same result as processing the single write with maximum timestamp, late writes can be ignored. This rule can be checked by comparing
the timestamp at which a local write was issued with the timestamp of the current object value (see Procedure 4.3.2). If the first timestamp is smaller than the second timestamp, that means the value to be written is older than the current value. Hence, the write is skipped. Otherwise, the object is updated with the value created by the local write.

If quiescing system activity is not tolerable for performance reasons, then, at the expense of higher complexity, different mechanisms [37, 19] permitting parallel activities in computation and validation phases can be used. Since these mechanisms either require extra data exchanges between transactions or some hidden system control structures (e.g. garbage collection), the expected gain of this extension might be reduced by additional complexity and overhead in real implementations.

Even if quiescing system activity in SOCC-FV is acceptable, the forward validation mechanism has another disadvantage: unpredictability. The number of active transactions and the sizes of their read sets that need to be checked by a validating transaction change with time. Hence, the validation time required by each transaction under SOCC-FV is unpredictable.

Moreover, the number of restarts each transaction can encounter is difficult to bound. For example, if a particular active transaction is repeatedly restarted by the conflict resolution procedure due to its continuous conflict with different validating transactions, then starvation occurs. It is possible to implement the conflict resolution procedure so that the conflict resolution procedure selects a validating transaction to restart when an active transaction has been repeatedly restarted. However, the problem remains because a validating transaction can still be repeatedly restarted if the active transactions that conflict with it have been repeatedly restarted due to their conflicts with other transactions. Hence, the number of restarts a transaction can experience is, in principle, unbounded.
4.3.2 Correctness of SOCC-FV

To prove the correctness of SOCC-FV, we first characterize the orderings of operations that we know must hold in all histories that could be generated by the SOCC-FV protocol. We then prove that for every SOCC-FV history \( \pi \) with these orderings, \( TG(\pi) \) is acyclic. According to Theorem 2.4.2, SOCC-FV must preserve conflict \( \Delta \)-serializability if \( TG(\pi) \) is acyclic for every SOCC-FV history \( \pi \). Recall that \( TG(\pi) \) is a directed transaction graph whose nodes are committed transaction instances in \( \pi \) and whose edges are all \( \tau_{i,j} \rightarrow \tau_{k,l} \) such that there are two conflicting operations \( e \) and \( e' \) in \( \tau_{i,j} \) and \( \tau_{k,l} \), respectively, which do not satisfy the free(\( \pi \)) relation, and \( e \) precedes \( e' \) in \( \pi \).

Recall that in the optimistic approach a transaction is divided into three phases, the computation, validation, and write phases. A transaction issues read, write, and computation operations in its computation phase. In order to prevent cascading aborts, local writes of an active transaction update only the local buffer of the transaction. The local writes are made accessible to other transactions only when the transaction enters its write phase. Hence, we have the following proposition:

**Proposition 1** Only the read operations performed in the computation phase and the committed writes performed in the write phase of one transaction can conflict with operations issued by other transactions.

Since a write phase is protected within a non-preemptable \( \text{Validate}_\text{Write}() \) procedure in SOCC-FV, we have \( R_{i,j} < B_{i,j} < \hat{W}_{i,j} < E_{i,j} \) for any transaction instance \( \tau_{i,j} \). \( R_{i,j} \) and \( \hat{W}_{i,j} \) contain all read operations and committed writes performed by \( \tau_{i,j} \), respectively. \( B_{i,j} \) and \( E_{i,j} \) denote the beginning and the end of the \( \text{Validate}_\text{Write}() \) procedure executed by \( \tau_{i,j} \), respectively.
Lemma 4.3.1 Let $\tau_{i,j}$ and $\tau_{k,l}$ be two committed transaction instances in a history $\pi$ produced by SOCC-FV. If there is an edge $\tau_{i,j} \rightarrow \tau_{k,l}$ in $TG(\pi)$, then $E_{i,j} < B_{k,l}$ in $\pi$.

Proof: Since there is an edge, $\tau_{i,j} \rightarrow \tau_{k,l}$ in $TG(\pi)$, there are two conflicting operations $e$ and $e'$ in $\tau_{i,j}$ and $\tau_{k,l}$, respectively, which do not satisfy any condition in the definition of the $free(\pi)$ relation and $e$ precedes $e'$ in $\pi$. The type of conflict may be one of the following three: (1) $\hat{w}_{i,j}^x < \hat{w}_{k,l}^x$, (2) $\hat{w}_{i,j}^x < r_{k,l}^x$, or (3) $r_{i,j}^x < \hat{w}_{k,l}^x$. Under each of these cases, if $\tau_{i,j}$ and $\tau_{k,l}$ do not overlap and $\tau_{i,j} \rightarrow \tau_{k,l}$, $\tau_{i,j}$ must complete its job before $\tau_{k,l}$ starts its execution. Hence, we have $E_{i,j} < B_{k,l}$ in $\pi$.

Next, consider these three cases when $\tau_{i,j}$ and $\tau_{k,l}$ overlap.

Case 1: $\hat{w}_{i,j}^x < \hat{w}_{k,l}^x$. This case implies that $\tau_{i,j}$ enters its write phase before $\tau_{k,l}$. Hence, $\tau_{k,l}$ can only start its Validate_Write() procedure only after $\tau_{i,j}$ exits from it. We have $E_{i,j} < B_{k,l}$ in $\pi$.

Case 2: $\hat{w}_{i,j}^x < r_{k,l}^x$. As we said before, a committed write is performed within a write phase which is inside the Validate_Write() procedure. Hence, $\hat{w}_{i,j}^x$ must be executed within a non-preemptable section. This implies $\hat{w}_{i,j}^x < E_{i,j}$. Since the Validate_Write() procedure is non-preemptable and $\hat{w}_{i,j}^x < r_{k,l}^x$, $r_{k,l}^x$ can only be executed after $\tau_{i,j}$ exits from the Validate_Write() procedure. We have $\hat{w}_{i,j}^x < E_{i,j} < r_{k,l}^x$. Moreover, since $\tau_{k,l}$ can enter the Validate_Write() procedure only after it finishes its computation phase, we have $\hat{w}_{k,l}^x < E_{i,j} < r_{k,l}^x < B_{k,l}$.

Case 3: $r_{i,j}^x < \hat{w}_{k,l}^x$. Since $r_{i,j}^x < \hat{w}_{k,l}^x$, $r_{i,j}^x$ must be issued before $B_{k,l}$. Otherwise, we will have $\hat{w}_{k,l}^x < r_{i,j}^x$. Assume that $E_{k,l} < B_{i,j}$ in $\pi$. Then $\tau_{k,l}$ must have performed a similarity test against $r_{i,j}^x$ in its validation phase. Since $r_{i,j}^x$ and $\hat{w}_{k,l}^x$ do not satisfy the $free(\pi)$ relation, $Wr_{\pi}(r_{i,j}^x)$ and $\hat{w}_{k,l}^x$ cannot be similar under $\Delta$ in $\pi$. Hence, the validation of $\tau_{k,l}$ must have caused $\tau_{i,j}$ to abort. After $\tau_{i,j}$ restarts, we will have $\hat{w}_{k,l}^x < r_{i,j}^x$, a contradiction. $\blacksquare$
Lemma 4.3.2 Let $\pi$ be a SOCC-FV history, and let $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ be a path in $TG(\pi)$, where each $T$ is a transaction instance in $\pi$ and $n > 1$. We have $E_1 \prec B_n$.

Proof: We prove this lemma by induction on $n$.

Induction basis: $n = 2$. This basis follows immediately from Lemma 4.3.1.

Induction hypothesis: The induction hypothesis assumes that the lemma holds when $n = k$, for some $k \geq 2$.

Induction step: We prove this lemma also holds when $n = k + 1$. By induction hypothesis, if there is a path $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k$ in $TG(\pi)$, then we have $E_1 \prec B_k$. By $T_k \rightarrow T_{k+1}$ and Lemma 4.3.1, we have $E_k \prec B_{k+1}$. Since $B_k \prec E_k$, by transitivity, we have $E_1 \prec B_{k+1}$.

Theorem 4.3.1 Every history generated by the SOCC-FV protocol is conflict $\Delta$-serializable.

Proof: Suppose $\pi$ is a history generated by SOCC-FV and there is a cycle in $TG(\pi)$ such as $T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 4.3.2, we have $E_1 \prec B_1$, a contradiction. Hence, $TG(\pi)$ cannot have a cycle. By Theorem 2.4.2, $\pi$ is conflict $\Delta$-serializable if and only if $TG(\pi)$ is acyclic.

4.4 Similarity-based Optimistic Concurrency Control- Backward Validation (SOCC-BV)

Forward validation has advantages of detecting conflicts earlier and providing flexibility in selecting transactions to be restarted. Unfortunately, under SOCC-FV, the time required to validate a transaction is dynamic. Since the functional and timing behavior of a real-time system should be as deterministic as necessary to
satisfy system specification [45], we now investigate backward validation to see if it can be more predictable.

4.4.1 Improving the Backward Validation Mechanism

It is well known that backward validation cannot detect conflicts as early as forward validation [16, 18, 19], resulting in higher risks of late restarts. However, as noted earlier, the risk of restarts under the optimistic approach can be reduced when the similarity criterion is used. When the similarity bounds increase slightly, the number of restarts can be decreased substantially. Hence, the next questions are: (1) Does backward validation require less overhead than forward validation? (2) Can backward validation be more predictable than forward validation?

In backward validation a validating transaction $\tau_v$ validates its read set and write set against the write sets of other transactions that have smaller transaction numbers and that overlap with $\tau_v$. Let us refer to the set of transactions that need to be validated by a validating transaction as $Target_v$ in the following discussions. Since a transaction obtains its transaction number only when it enters its validation phase, the transactions in $Target_v$ do not read or write anymore. Hence, the write sets of transactions in $Target_v$ are static. However, the number of transactions in $Target_v$ is still unbounded. Also, despite the fact that transactions in $Target_v$ have finished their jobs, their write sets must be kept for an arbitrary period of time until all transactions that overlap with them are also finished. This requires extra system overhead to keep track of transaction overlaps and extra overhead in cleaning up system memory. All these factors introduce uncertainty in the transaction execution time, making it difficult to guarantee the timely execution of transactions.

If we can prevent transactions from overlapping in their validation and write phases, then a validating transaction only needs to validate its read set against the
current values in the system. Let us use the following two scenarios to explain the reason.

**Example 4.4.1**

*Scenario 1*

\[
\begin{align*}
\tau_{i,j}: \ldots \bar{w}_x & \quad \bar{w}_y \hat{c} \\
\tau_{k,l}: \quad r_xr_yv_xv_y \hat{c}
\end{align*}
\]

*Scenario 2*

\[
\begin{align*}
\tau_{i,j}: \ldots \bar{w}_x & \quad \bar{w}_y \hat{c} \\
\tau_{k,l}: \quad r_xr_y & \quad v_xv_y \hat{c}
\end{align*}
\]

In scenario 1, the write phase of the committed transaction \(\tau_{i,j}\) and the validation phase of the validating transaction \(\tau_{k,l}\) overlap. If \(\tau_{k,l}\) only validates its read set (contains \(x\) and \(y\)) against the current global values of \(x\) and \(y\), then \(\tau_{k,l}\) will not be restarted because the current values of \(x\) and \(y\) are still the same as the values read by \(\tau_{k,l}\). As a result, a cycle will be formed between \(\tau_{i,j}\) and \(\tau_{k,l}\). However, if \(\tau_{k,l}\) validates against the write set of \(\tau_{i,j}\), the conflicts on \(x\) and \(y\) will be detected by \(\tau_{k,l}\) and \(\tau_{k,l}\) must be restarted. After \(\tau_{k,l}\) restarts, no cycle can be formed. This is the traditional backward validation approach.

In scenario 2, if we eliminate the possible overlap between the write phase of \(\tau_{i,j}\) and the validation phase of \(\tau_{k,l}\), then \(\tau_{k,l}\) needs to validate its read set only against the current global values in the system. A validating transaction will be restarted only if the object it read is updated by other transactions. In scenario 2, \(\tau_{k,l}\) will be restarted because an object it read \((y)\) has been updated by \(\tau_{i,j}\).

At the first glance, the concurrency level may be reduced if we prevent transactions from overlapping in their validation and write phases. However, since validation and write phases of a transaction are relatively short compared to a transaction's computation phase, reducing overhead in these phases can be more beneficial (see more detailed discussion below) than adding more complicated operations to these
short phases. Otherwise, the overhead created by complicated operations may quickly overshadow the slightly increased concurrency.

The advantage of preventing transactions from overlapping in their validation and write phases comes from the fact that, the time required for a validation phase is proportional to the size of transaction’s read set rather than depending on the number of transactions that may overlap with the validating transaction. In other words, only one comparison of time stamps is needed to validate an object, independent of the number of overlapping transactions and the sizes of their write sets. Hence, the set intersection operation usually required in the optimistic approach is not necessary. Moreover, a validating transaction need not validate against the write sets of committed transactions. Therefore, the write sets of committed transactions do not need to be kept after they finish their jobs, and the system does not need to monitor when to release transactions’ working memory. This can significantly reduce the system overhead and can make schedulability analysis easier.

4.4.2 Overview of SOCC-BV

To achieve the goal of eliminating overlaps between validation and write phases of transactions that conflict, we require each transaction to request a read validation lock (RVL) or a write validation lock (WVL) on every object it reads or writes before it enters its validation phase. A lock request on an object $x$ can be granted if no conflicting lock currently exists on $x$. Two locks are said to be conflict if one of them is WVL. We call the time period during which a transaction requests validation locks the validation lock phase of the transaction.

Only transactions that have successfully obtained all the locks it needs can enter the validation phase. Otherwise, the transaction must release all locks it has obtained and wait for the lock it fails to obtain is released by another transaction. This is the
"no wait with hold" method of deadlock prevention. Hence, under our approach, no deadlock can occur. However, if two transactions compete for the same set of locks, they may not be able to obtain all the locks they need, resulting in a situation where no transaction can enter its validation phase. Hence, we put the validation-lock phase in a critical section. If multiple transactions are waiting for the critical section, the one with the highest priority will get access first.

We give the detailed operations performed in the validation and write phases of a validating transaction in the following procedure.

**Procedure 4.4.1**

```plaintext
Validate_Write(τ_v,k) {
    ValidationLock();
    if (SimTest(τ_v,k) == FALSE) then
        release RVLs/WVLs, and restart;
    endif
    WritePhase(τ_v,k); /* write phase */
    τ_v,k commits, and releases RVLs/WVLs;
}
```

Figure 4.3: The validation procedure performed in the SOCC-BV protocol.

In Procedure 4.4.1, a validating transaction first enters the validation lock phase in which the validating transaction requests RVLs and WVLs on all the objects it accessed (see Procedure 4.4.2). The validation locks obtained by a validating transaction prevent other conflicting transactions from entering their validation and write phases, avoiding possible overlapping in these phases. If the validating transaction
fails to obtain any lock it needs, it releases all the locks it obtained. Hence, no deadlock can occur.

**Procedure 4.4.2**

ValidationLock() {
    BEGIN:
    begin_critical_section();
    request validation locks on accessed objects;
    if (fail) then
        release all locks;
        end_critical_section(), wait, and goto BEGIN;
    endif
    end_critical_section();
}

Figure 4.4: Validation lock phase.

After the validating transaction obtains all the validation locks it needs, it enters its validation phase and performs similarity tests on all objects it read. If any object it read is not similar to the current object value in the system, the similarity test function (shown in Procedure 4.4.3) returns FALSE. The validating transaction then releases all the locks it obtained and restarts.

The similarity test in Procedure 4.4.3 is performed by checking the temporal distance between the timestamp of an object read by the validating transaction and the current timestamp of the object. If the temporal distance between these two timestamps is greater than the similarity bound of the object, then the value read by
Procedure 4.4.3

SimTest(τ_{v,k})

\forall x \in RS_{v,k} do

if \((ts_x - ts(\text{Wr}_\pi(\tau_{v,k}^x))) > sb_x)\) then

return FALSE; /* not similar */

endif

endo

return TRUE;
}

Figure 4.5: The similarity test procedure.

the validating transaction is not similar to the current object value. The similarity function returns FALSE. If all objects read by the validating transaction pass the similarity test, the SimTest() function returns TRUE.

One interesting thing about this similarity test function is that when the similarity criterion is not applicable (\(\forall x, sb_x = 0\)), the similarity test function always returns FALSE if any transaction has updated the object that is currently read by the validating transaction. Thus, when similarity is not applicable, SOCC-BV behaves just like a traditional backward validation protocol, and still accepts serializable histories.

If all the objects read by the validating transaction pass the similarity test, then there are no read-write conflicts and the validating transaction proceeds to its write phase (see Procedure 4.3.2). Note that Procedure 4.3.2 implements TWR and prevents the data regression problem.
4.4.3 Correctness of SOCC-BV

To prove the correctness of SOCC-BV, we follow steps similar to those used in proving the correctness of SOCC-FV. That is, we first characterize the orderings of operations that we know must hold in all histories that could be generated by the SOCC-BV protocol. We then prove that for every SOCC-BV history \( \pi \) with the orderings, \( TG(\pi) \) is acyclic.

Again, by Proposition 1, under the optimistic approach only read operations performed in a computation phase and committed writes performed in a write phase of one transaction may conflict with operations issued by other transactions. If a transaction \( \tau_{i,j} \) attempts to update an object \( x \), we have \( \hat{w}^x_{i,j} \prec \hat{c}^x_{i,j} \) by definition. Moreover, under SOCC-BV, a transaction instance must obtain write validation locks (WVLs) before it enters its validation phase, and the WVLs held by \( \tau_{i,j} \) are released after \( \tau_{i,j} \) commits. We have the following proposition.

**Proposition 2** If \( \tau_{i,j} \) attempts to update an object \( x \), we have \( l^x_{i,j} \prec \hat{w}^x_{i,j} \prec \hat{c}^x_{i,j} \prec u^x_{i,j} \). \( l^x_{i,j} \) and \( u^x_{i,j} \) denote the operations that set and release the write validation lock on \( x \), respectively.

Since SOCC-BV implements Thomas's Write Rule, we also have the following proposition.

**Proposition 3** If a committed write \( \hat{w}^x_{k,l} \) is issued after another committed write \( \hat{w}^x_{i,j} \), then \( ts(\hat{w}^x_{i,j}) < ts(\hat{w}^x_{k,l}) \).

**Lemma 4.4.1** If \( ts(\hat{w}^x_{i,j}) < ts(\hat{w}^x_{k,l}) \) and \( \hat{w}^x_{i,j} \not\sim \hat{w}^x_{k,l} \) (\( \hat{w}^x_{i,j} \) is not similar to \( \hat{w}^x_{k,l} \)), then any committed write issued after \( \hat{w}^x_{k,l} \) cannot be similar to \( \hat{w}^x_{i,j} \).

**Proof:** According to Proposition 3, any committed write \( \hat{w}^x_{m,n} \) issued after \( \hat{w}^x_{k,l} \) must have \( ts(\hat{w}^x_{k,l}) < ts(\hat{w}^x_{m,n}) \). Since \( \hat{w}^x_{i,j} \not\sim \hat{w}^x_{k,l} \), \( ts(\hat{w}^x_{i,j}) - ts(\hat{w}^x_{k,l}) > sb_x \). Also, because \( ts(\hat{w}^x_{k,l}) < ts(\hat{w}^x_{m,n}) \), \( ts(\hat{w}^x_{m,n}) - ts(\hat{w}^x_{i,j}) > sb_x \). Hence, \( \hat{w}^x_{i,j} \not\sim \hat{w}^x_{m,n} \). \( \blacksquare \)
Lemma 4.4.2 Let $\tau_{i,j}$ and $\tau_{k,l}$ be two committed transaction instances in a history $\pi$ produced by SOCC-BV. If there is an edge $\tau_{i,j} \rightarrow \tau_{k,l}$ in $TG(\pi)$, then $ts(\hat{c}_{i,j}) < ts(\hat{c}_{k,l})$.

**Proof:** Since there is an edge, $\tau_{i,j} \rightarrow \tau_{k,l}$ in $TG(\pi)$, there are two conflicting operations $e$ and $e'$ in $\tau_{i,j}$ and $\tau_{k,l}$, respectively, which do not satisfy any condition in the definition of the $free(\pi)$ relation and $e$ precedes $e'$ in $\pi$. The type of conflict may be one of the following three: (1) $w_{i,j}^x < w_{k,l}^x$, (2) $w_{i,j}^x < r_{k,l}^x$, or (3) $r_{i,j}^x < w_{k,l}^x$. Under each of these cases, if $\tau_{i,j}$ and $\tau_{k,l}$ do not overlap and $\tau_{i,j} \rightarrow \tau_{k,l}$, $\tau_{i,j}$ must finish its job before $\tau_{k,l}$ starts its execution. Hence, we have $ts(\hat{c}_{i,j}) < ts(\hat{c}_{k,l})$.

Next, consider these three cases when $\tau_{i,j}$ and $\tau_{k,l}$ overlap.

**Case1:** $w_{i,j}^x < w_{k,l}^x$. By proposition 2, for $\tau_{i,j}$ and $\tau_{k,l}$ we know $l_{i,j}^x < w_{i,j}^x < \hat{c}_{i,j} < u_{i,j}^x$ and $l_{k,l}^x < w_{k,l}^x < \hat{c}_{k,l} < u_{k,l}^x$, respectively. Since no two transaction instances can set write validation locks on one object at the same time, $w_{i,j}^x < w_{k,l}^x$ implies that $\tau_{k,l}$ can get the validation lock on $x$ only after $\tau_{i,j}$ releases the write validation lock on $x$. We have $w_{i,j}^x < l_{k,l}^x$. By transitivity, we conclude that $ts(\hat{c}_{i,j}) < ts(\hat{c}_{k,l})$.

**Case2:** $w_{i,j}^x < r_{k,l}^x$. Again, by proposition 2, we know $l_{i,j}^x < w_{i,j}^x < \hat{c}_{i,j} < u_{i,j}^x$ and $r_{k,l}^x < l_{k,l}^x < \hat{c}_{k,l} < u_{k,l}^x$ must hold for $\tau_{i,j}$ and $\tau_{k,l}$, respectively. Since we have $w_{i,j}^x < r_{k,l}^x$, $u_{i,j} < l_{k,l}$ must hold. Again, by transitivity, we have $ts(\hat{c}_{i,j}) < ts(\hat{c}_{k,l})$.

**Case3:** $r_{i,j}^x < w_{k,l}^x$. Assume we have $ts(\hat{c}_{i,j}) \geq ts(\hat{c}_{k,l})$ under this case. We will have $u_{k,l} < l_{i,j}$. Hence, the validation phase of $\tau_{i,j}$ must detect $x$ has been updated after $r_{i,j}^x$ was issued. Since $r_{i,j}^x$ and $w_{k,l}^x$ do not satisfy the $free(\pi)$ relation, according to Lemma 4.4.1, any committed write issued after $w_{k,l}^x$ cannot be similar to $Wr(r_{i,j}^x)$. Hence, $\tau_{i,j}$ must be restarted. After $\tau_{i,j}$ is restarted, we will have $w_{k,l}^x < r_{i,j}^x$, a contradiction. ■
Lemma 4.4.3  Let π be a SOCC-BV history, and let $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ be a path in $TG(\pi)$, where each $T$ is a transaction instance in $\pi$ and $n > 1$. We have $ts(\hat{c}_1) < ts(\hat{c}_n)$.

Proof: We prove this lemma by induction on $n$. First, let $n = 2$, this induction basis follows immediately from Lemma 4.4.2. The induction hypothesis assumes that when $n = k$, for some $k \geq 2$, $ts(\hat{c}_1) < ts(\hat{c}_k)$. Let $n = k + 1$ and $T_k \rightarrow T_{k+1}$. We have $ts(\hat{c}_k) < ts(\hat{c}_{k+1})$ according to the Lemma 4.4.2. By transitivity, we obtain $ts(\hat{c}_1) < ts(\hat{c}_{k+1})$.

Theorem 4.4.1  Every history generated by the SOCC-BV protocol is conflict $\Delta$-serializable.

Proof: Suppose $\pi$ is a history generated by SOCC-BV and there is a cycle in $TG(\pi)$ such as $T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 4.4.3, we have $ts(\hat{c}_1) < ts(\hat{c}_n)$, a contradiction. Hence, $TG(\pi)$ cannot have a cycle. By Theorem 2.4.2, $\pi$ is conflict $\Delta$-serializable if and only if $TG(\pi)$ is acyclic.

Lemma 4.4.4  No deadlock can occur in any SOCC-BV history.

Proof: Since a transaction instance must release all the validation locks it obtains so far if it is unable to get all locks it needs, the "wait with hold" [11] condition of four necessary conditions for deadlock is denied. No deadlock can occur in any SOCC-BV history.

4.5 Similarity-based Optimistic-then-Pessimistic Protocol (SOPP)

The SOCC-BV protocol has advantages over the traditional backward and forward validation mechanisms because it requires relatively lower implementation overhead and its validation times are bounded by the sizes of the read sets of the transactions.
However, SOCC-BV also suffers two disadvantages: (1) transactions may be repeatedly restarted, and (2) higher priority transactions may be blocked by lower priority transactions for an unbounded amount of time.

In Chapter 4.5.1, we first explain the cause of the repeated restarts and propose our solution: the optimistic-then-pessimistic approach. The unbounded blocking problem is then discussed in Chapter 4.5.2. In Chapter 4.5.3, we extend our optimistic-then-pessimistic approach to the Similarity-base Optimistic-then-Pessimistic Protocol (SOPP) [30] that solves both the repeated restarts and the unbounded blocking problems. The properties of SOPP, the correctness of SOPP, and the schedulability of SOPP are then analyzed and proved in Chapter 4.5.4, Chapter 4.5.5, and Chapter 4.5.6, respectively.

4.5.1 The Starvation Problem

Transactions under the control of SOCC-BV may still suffer from the starvation problem because if a validating transaction detects a conflict, it must first release all the validation locks it has obtained and then restart. Since validation locks are released, other transactions can enter their write phases and create new conflicts with respect to the validating transaction. As a result, the restarted transaction may continuously detect conflicts in its validation phase, and the transaction will be restarted over and over again, resulting in starvation. This problem creates uncertainty for the schedulability analysis because the total number of restarts each transaction may encounter is unbounded.

One way to overcome this problem is to restart a transaction without releasing the validation locks the transaction obtained at the begin of its validation phase. Since a restarted transaction still holds validation locks on all the objects it accessed, no conflicting transactions can enter their validation and write phases (however, all
other transactions are still free to work concurrently in their computation phases). Hence, no new conflict can be developed with respect to the restarted transaction. As a result, a restarted transaction is guaranteed to commit. It also follows immediately that each transaction can be restarted at most once. The validation locks obtained by the validating transaction are released only after the transaction finally commits. In fact, since no conflict can occur with respect to a restarted transaction, a restarted transaction does not need to perform the similarity test again.

This modified approach has both optimistic and pessimistic flavors. A transaction is first executed in an optimistic stage where it assumes conflicts are rare events. If a conflict does occur, the validating transaction is restarted without releasing the validation locks it obtains in its optimistic stage. Hence, a restarted transaction is now executed in a pessimistic stage in which it assumes that conflicts might occur again. We call this approach as an Optimistic-then-Pessimistic approach. In the following, we show how to modify the SOCC-BV protocol to produce the optimistic-then-pessimistic method.

Procedure 4.5.1 is similar to Procedure 4.4.1. However, if the returned value of the SimTest() function is FALSE (a conflict cannot be resolved by similarity), the validating transaction is restarted immediately without releasing the validation locks it held. The validation locks are released only after the validating transaction commits.

4.5.2 The Problem of Chain Blocking

The optimistic-then-pessimistic mechanism guarantees that each transaction can only be restarted once, solving the starvation problem. Unfortunately, under this mechanism, a higher priority transaction can still be blocked by an unbounded number of lower priority transactions. We use the following figure to explain the reason.
Procedure 4.5.1

Validate_Write(\( \tau_{v,k} \)) {
    ValidateLock();
    if (SimTest(\( \tau_{v,k} \)) == FALSE) then
        execute computation phase of \( \tau_{v,k} \); /* restart */
    endif
    WritePhase(\( \tau_{v,k} \)); /* write phase */
    \( \tau_{v,k} \) commits, and release RVLs/WVLs;
}

Figure 4.6: Integrating the optimistic-then-pessimistic approach with SOCC-BV.

Figure 4.7: Blocking chain in a single processor system.
Referring to Figure 4.5.2, let $J_1$ have the highest priority and $J_3$ the lowest. Suppose that $J_1$ needs to access objects $x$ and $y$, and $J_2$ and $J_3$ update $x$ and $y$, respectively. Let $J_3$ start first and enter its validation phase. Since $J_3$ updates $y$, it sets a validation lock on $y$. Assume that $J_2$ preempts $J_3$ within $J_3$'s validation phase and sets a validation lock on $x$ within its own validation phase. Hence, $J_2$ and $J_3$ hold validation locks on the objects $x$ and $y$, respectively. When transaction $J_1$ is initiated, it preempts $J_2$ and executes until its validation lock phase where it is blocked by the validation lock set by $J_2$. Therefore, $J_2$ is executed again and finishes its job. After $J_2$ finishes its job, $J_1$, unfortunately, is still blocked because $J_3$ still holds a validation lock on the object $y$. Hence, $J_1$ can start its validation phase only after both $J_2$ and $J_3$ finish their jobs. As a result, $J_1$ is blocked by a chain of blockings [39]. Since the number of blocks in a blocking chain can be varied, the blocking time each transaction can experience may be substantial. This uncertainty again makes the schedulability analysis difficult.

One way to prevent the chained blocking is to integrate the priority ceiling protocol with the validation lock mechanism so that a lock request is granted only if it satisfies the priority ceiling rules. That is, a transaction can obtain a validation lock if its priority is higher than the priority ceilings of all validation locks held by all other transactions. However, to set up priority ceiling rules, prior knowledge of data access requirements of transactions must be assumed. If we want to prevent the blocking chain without relying on prior information about transactions, we need another mechanism to protect the validation, restarted, and write phases of a transaction.

Another way to avoid chained blocking in a single processor system is to turn off the CPU preemption mechanism just before a transaction enters its validation lock phase. Because the CPU preemption mechanism is turned off, a validating transaction cannot be preempted and no conflicting writes can be issued by other transactions.
Hence, the validating transaction can only be restarted due to the conflicts developed before the transaction enters its validation lock phase. As a result, a transaction can be restarted at most once. In other words, a restarted transaction is guaranteed to commit. After the validating transaction commits, the CPU preemption mechanism is turned on again. At this moment, the transaction with the highest priority among all the transactions blocked by the validating transaction is executed. Since a validating transaction can never be preempted, and the highest priority transaction gets the CPU after a lower priority transaction commits (and re-enables the CPU preemption mechanism), no chain blocking can be formed.

Unfortunately, this mechanism can work only on a single processor system. The reason is because all other processors are still able to execute different transactions, and these transactions may enter their validation phases concurrently and set validation locks on different objects. When a higher priority transaction is ready to enter its validation lock phase on a particular processor, the objects it accessed may have been locked by other transactions running on different processors. As a result, the higher priority transaction still needs to wait for multiple transactions, resulting in an unbounded blocking time. The following figure illustrates this situation.

In figure 4.8, assume a system has three processors. Let transaction $J_2$ enter its validation phase first on processor 2 and set a validation lock on object $x$. Since $J_2$ disables the CPU preemption mechanism of processor 2, no transaction can be executed on processor 2. However, all other processors are still free to process other transactions. Assume transaction $J_1$ is then executed on processor 1, and $J_1$ access objects $x$ and $y$. When $J_1$ is ready to enter its validation phase, $J_1$ is blocked because $J_1$ cannot get the validation lock on the object $x$. If $J_3$ is executed at this instance on processor 3 and sets a validation lock on object $y$, then even when $J_2$ finishes its job, $J_1$ will still be blocked because object $y$ is still locked by $J_3$. 
We present in the next subsection the Similarity-based Optimistic then Pessimistic Protocol (SOPP) that solves the problem of chain blocking and still guarantees that each transaction is restarted at most once.

4.5.3 SOPP Overview

The advantage of the validation lock mechanism is that it allows transactions that access different objects to enter their validation phase concurrently, resulting in higher concurrency. However, if objects locked by different transactions are later requested by a higher priority transaction, then the higher priority transaction must wait for multiple transactions to release validation locks, resulting in chained blocking.

One way to solve this problem, at the expense of lower concurrency, is to allow only one transaction to work in its validation phase at any time. To achieve this goal, each
transaction, before entering its validation phase, must first request a unique system
lock. If the system lock is currently held by one transaction, all other transactions
that try to enter their validation phases must wait. If multiple transactions are ready
to enter their validation phases, the one with the highest priority gets the lock.

Since only one transaction can work in its validation phase at any time, a validat­ing
transaction need not request validation locks on objects it accesses any more. In
other words, under SOPP, requesting validation locks is no longer necessary. How­
ever, once a transaction obtains a system lock (to work in its validation phase), it
must also disable the preemption mechanism on the CPU it occupies. Otherwise, a
lower priority transaction that holds the system lock (and hence possibly blocks high
priority transactions) may later be preempted by other medium priority transaction,
resulting in chain blocking again. The system lock can be released and the CPU
preemption mechanism can be re-enabled after a transaction successfully commits.

In summary, to prevent a blocking chain in a multi-processor system, what we
need is to prevent a validating transaction from being preempted on one processor,
and, at the same time, allow only one of transactions running on different processors
to work in its validation phase. Asking each transaction to obtain the system lock
achieves the goal that only one transaction can work in its validation phase at any
time in a multi-processor system. Disabling CPU preemption mechanism, on the
other hand, prevents a validating transaction from being preempted. It is important
to note that these two operations must be implemented in an atomic instruction.
Otherwise, the problem of unbounded blocking can still occur.

The following procedure gives the details of SOPP.

In Procedure 4.5.2, a validating transaction first obtains a system lock and disables
the CPU preemption mechanism. The validating transaction then enters its validation
phase and performs similarity tests on all objects it read. If any object it read is not
**Procedure 4.5.2**

Validate_Write(\(\tau_{v,k}\)) {

get system lock and disable preemption; /* atomic */

if (SimTest(\(\tau_{v,k}\)) == FALSE) then

eexecute computation phase of \(\tau_{v,k}\); /* restart */
endif

WritePhase(\(\tau_{v,k}\)); /* write phase */

\(\tau_{v,k}\) commits;

release system lock and enable preemption; /* atomic */

};

Figure 4.9: The validation procedure performed in the SOPP protocol.

similar to the current object value in the system, the similarity test function (shown in Procedure 4.4.3) returns FALSE. The validating transaction is then restarted within a pessimistic stage in which no transactions can enter their write phases to update objects.

If all objects read by the validating transaction pass the similarity test, then there is no read-write conflict, and the validating transaction proceeds to its write phase (see Procedure 4.5.2). Note that Procedure 4.5.2 implements TWR and prevents the data regression problem. After the write phase, the validating transaction commits. The system lock is then released and the CPU preemption mechanism is re-enabled.

The pessimistic approach taken in SOPP is different from traditional pessimistic two-phase locking mechanisms (2PL) in many aspects. First, our approach is integrated with an optimistic approach. The pessimistic approach is taken in SOPP only when conflicts occur in the optimistic stage. Secondly, on a multiple processor
system, the transaction that holds the system lock does not prevent other transactions from working in their computation phases, while under 2PL any conflict between read/write locks blocks the conflicting transactions. Third, unlike in 2PL, our approach is deadlock free. Finally, perhaps the most important aspect is that the blocking time and the number of restarts each transaction may encounter under SOPP are bounded and predictable. We will provide a sufficient schedulability analysis in Chapter 4.5.6.

4.5.4 Properties of SOPP

An important feature of SOPP is that one can develop a schedulability analysis for it in the sense that a schedulability bound can be determined. If the utilization of the transaction set stays below this bound, then the deadlines of all the transactions can be guaranteed. In order to create such a bound, it is necessary to determine the worst case duration of blocking time and the total number of restarts that any transaction can encounter. In this section, we first quickly review SOPP and define some terms. We then analyze the properties of SOPP. The correctness of SOPP and the SOPP schedulability will be discussed in the next two subsections.

A transaction \( J \), which has the highest priority among the transactions ready to run, is assigned to the processor. Before transaction \( J \) enters its validation phase, it must first obtain the system lock and disable the CPU preemption mechanism. Transaction \( J \) will be blocked from entering its validation phase if another transaction \( I \) has entered its validation phase. In this case, if transaction \( J \) has higher priority than \( I \), transaction \( J \) is said to be blocked by \( I \). Otherwise, transaction \( J \) is waiting for \( I \). If transaction \( J \) detects a conflict, \( J \) is restarted. When transaction \( J \) finally commits, it releases the system lock and re-enables the CPU preemption mechanism. The highest priority transaction, if any, blocked by transaction \( J \) obtains the lock.
Definition 4.5.1 We say a transaction $\tau_{i,j}$ starts its **final section** when $\tau_{i,j}$ obtains the system lock and disables the CPU preemption mechanism. We say $\tau_{i,j}$ exits from its final section when $\tau_{i,j}$ releases the system lock and re-enables the CPU preemption mechanism. Hence, a final section of a transaction $\tau_{i,j}$ includes its validation phase, a possible restarted computation phase (see the proof below), and a possible write phase (if the $\tau_{i,j}$ is an update transaction).

Proposition 4 For a transaction to enter its final section, it must obtain a system lock and disable the CPU preemption mechanism. Hence, at any time, only one transaction can work in its final section.

Proposition 5 A transaction $\tau_{i,j}$, which has the highest priority among the transactions ready to enter their final sections obtains the system lock.

Definition 4.5.2 A transaction $\tau_{i,j}$ is said to be blocked by another transaction $\tau_{k,l}$ if $\tau_{i,j}$ has higher priority than $\tau_{k,l}$ and $\tau_{i,j}$ has to wait for $\tau_{k,l}$ to exit its final section in order to continue execution.

Lemma 4.5.1 Let $B_{i,j}$ and $E_{i,j}$ denote the begin and the end of the final section of a transaction $\tau_{i,j}$. Under SOPP, for any pair of committed transactions, $\tau_{i,j}$ and $\tau_{k,l}$, either $E_{i,j} < B_{k,l}$ or $E_{k,l} < B_{i,j}$ must hold.

Proof: According to Proposition 4, at any time, only one transaction can work in its final section. Hence, for any pair of committed transaction $\tau_{i,j}$ and $\tau_{k,l}$, one transaction can enter its final section only after another transaction finishes its final section. In other words, the final sections of different transactions cannot be overlapped. We have either $E_{i,j} < B_{k,l}$ or $E_{k,l} < B_{i,j}$. 

Lemma 4.5.2 Under SOPP, if a higher priority transaction $\tau_{i,j}$ is blocked by a lower priority transaction $\tau_{k,l}$, then $\tau_{k,l}$ must be executing within its final section.
Proof: If \( \tau_{k,l} \) is not executing in its final section, then \( \tau_{k,l} \) cannot disable the CPU preemption mechanism and \( \tau_{k,l} \) cannot hold the system lock. Hence, any higher priority transaction can preempt \( \tau_{k,l} \), and \( \tau_{k,l} \) is not able to block \( \tau_{i,j} \). 

Lemma 4.5.3 Under SOPP, a lower priority transaction \( \tau_{k,l} \) can block a high priority transaction \( \tau_{i,j} \) for at most the duration of its final section.

Proof: By Lemma 4.5.2, for \( \tau_{k,l} \) to block \( \tau_{i,j} \), \( \tau_{k,l} \) must be currently executing in its validation section. Before \( \tau_{k,l} \) enters its validation section, \( \tau_{k,l} \) cannot block \( \tau_{i,j} \). Once \( \tau_{k,l} \) exits its validation section, \( \tau_{k,l} \) must release its system lock and re-enable the CPU preemption mechanism, it cannot block \( \tau_{i,j} \) anymore.

Lemma 4.5.3 proves that a high priority transaction can be blocked by a lower priority transaction for a bounded amount of time. However, we still need to prove that a high priority transaction can be blocked by only one lower priority transaction before it finishes its job.

Lemma 4.5.4 A transaction can be blocked by at most one lower priority transaction.

Proof: Suppose that transaction \( J \) can be blocked by \( n > 1 \) lower priority transactions (refer to as \( J_1, J_2, ..., J_n \)) before it finishes its job. According to Lemma 4.5.2, \( J \) can be blocked by those \( n \) transactions only when they are in their final sections. Moreover, according to Lemma 4.5.1, for any pair of committed transactions, either \( E_i < B_j \) or \( E_j < B_i \) must hold. Hence, \( J \) must be blocked by \( J_1...J_n \) in \( n \) non-overlapping periods. Suppose that the first two lower priority transactions that block \( J \) are \( J_1 \) and \( J_2 \). Hence, we have either \( E_1 < B_2 \) or \( E_2 < B_1 \).

Let us first consider the case where \( E_1 < B_2 \). For \( J \) to be blocked by \( J_1 \), the weakest condition that must hold is \( S < E_1 \), where \( S \) is the start time of the transaction \( J \). If
$E_1 < S$, then $J_1$ has finished its job and it cannot block $J$. For $J$ to be also blocked by $J_2$, the second condition that must hold is $E_2 < B$. However, since $J$ can be executed right after $J_1$ exits from its final section and $J_2$ has lower priority than $J$, $J$ can be executed without being preempted by $J_2$ until it finishes its job (Proposition 5). We have $B < E_2$ which contradicts the second condition. Hence, these two conditions can not be held at the same time. In other words, if $S < E_1$ holds, we must have $E < B_2$.

In the second case where $E_2 < B_1$, we must have $S < E_2$ and $E_1 < B$ so that $J$ can be blocked by both $J_1$ and $J_2$. However, for the similar reasons explained above, these two conditions can not be held at the same time. Hence, the lemma follows.

**Lemma 4.5.5** Under SOPP, each transaction can be restarted at most once.

**Proof:** Under SOPP, for a transaction $\tau_{i,j}$ to be restarted, $\tau_{i,j}$ must detect that it has read a value from an object $x$ which is dissimilar to the current value of $x$. In other words, after $\tau_{i,j}$ read $x$, another transaction must have created a dissimilar value for $x$. Now assume $\tau_{i,j}$ is restarted more than once. Then, some other transactions must have entered their write phases and updated $x$ each time when $\tau_{i,j}$ enters its first and restarted computation phases. However, under SOPP, for a transaction to create a new value for an object $x$, the transaction must enter its write phase, which must be embedded within its final section. Since, by Definition 4.5.1, the validation, the restart computation, and the write phases of $\tau_{i,j}$ are also embedded within one final section, we will have multiple transactions work in their final sections concurrently. This contradicts Proposition 4 and Lemma 4.5.1.

Finally, when similarity is not applicable, then for every object $x$ in the system, we have $sb_x = 0$. That means the similarity test in Procedure 4.4.3 always returns FALSE and forces the validating transaction to restart if any write has taken place.
before the similarity test. Although in this situation transactions are more likely to be restarted, each transaction, according to Lemma 4.5.5, can still be restarted at most once. Moreover, since transactions are still able to run concurrently in their computation phases, SOPP can still accept serializable histories.

4.5.5 Correctness of SOPP

To prove the correctness of SOPP, we have characterized the orderings of operations that we know must hold in all histories that could be generated by SOPP in the previous subsection. In this subsection, we prove that for every SOPP history \( \pi \) with these orderings, \( TG(\pi) \) is acyclic. According to Theorem 2.4.2, if \( TG(\pi) \) is acyclic for every SOPP history \( \pi \), SOPP must preserve conflict \( \Delta \)-serializability.

According to proposition 1, we know that, under the optimistic approach, only the read operations performed in the computation phase and the committed writes performed in the write phase of one transaction may conflict with operations issued by other transactions. Moreover, since SOPP also implements TWR, Proposition 3 and Lemma 4.4.1 must also hold for SOPP.

Since a write phase is protected within a non-preemptable final section in SOPP, we have \( R_{i,j} < B_{i,j} < \hat{W}_{i,j} < E_{i,j} \) for any transaction instance \( \tau_{i,j} \). \( R_{i,j} \) and \( \hat{W}_{i,j} \) contain all read operations and committed writes performed by \( \tau_{i,j} \), respectively. \( B_{i,j} \) and \( E_{i,j} \) denote the begin and the end of the final section executed by \( \tau_{i,j} \), respectively.

**Lemma 4.5.6** Let \( \tau_{i,j} \) and \( \tau_{k,l} \) be two committed transaction instances in a history \( \pi \) produced by SOPP. If there is an edge \( \tau_{i,j} \rightarrow \tau_{k,l} \) in \( TG(\pi) \), then \( E_{i,j} < B_{k,l} \) in \( \pi \).

**Proof:** Since there is an edge, \( \tau_{i,j} \rightarrow \tau_{k,l} \) in \( TG(\pi) \), there are two conflicting operations \( e \) and \( e' \) in \( \tau_{i,j} \) and \( \tau_{k,l} \), respectively, which do not satisfy any condition in the definition of the \( free(\pi) \) relation and \( e \) precedes \( e' \) in \( \pi \). The type of conflict may be one of the
following three: (1) \( \hat{w}_{i,j}^x < \hat{w}_{k,l}^x \), (2) \( \hat{u}_{i,j}^* < r_{k,l}^* \), or (3) \( r_{i,j}^* < \hat{w}_{k,l}^x \). Under each of these cases, if \( \tau_{i,j} \) and \( \tau_{k,l} \) do not overlap and \( \tau_{i,j} \rightarrow \tau_{k,l} \), \( \tau_{i,j} \) must complete its job before \( \tau_{k,l} \) starts its execution. Hence, we have \( E_{i,j} < B_{k,l} \) in \( \pi \). Next, consider these three cases when \( \tau_{i,j} \) and \( \tau_{k,l} \) overlap.

Case 1: \( \hat{w}_{i,j}^x < \hat{w}_{k,l}^x \). Under SOPP, committed writes can only take place within the write phase of a transaction, and the write phase of a transaction is embedded in the final section of the transaction. We have \( B_{i,j} < \hat{W}_{i,j} < E_{i,j} \) and \( B_{k,l} < \hat{W}_{k,l} < E_{k,l} \).

In the case 1 where \( \hat{w}_{i,j}^x < \hat{w}_{k,l}^x \), according to Lemma 4.5.1, we must have \( E_{i,j} < B_{k,l} \) in \( \pi \). That means \( \tau_{i,j} \) must enter the final section before \( \tau_{k,l} \), and \( \tau_{k,l} \) can only enter the final section after \( \tau_{i,j} \) exists from it.

Case 2: \( \hat{u}_{i,j}^* < r_{k,l}^* \). Again, a committed write is performed within a write phase which is embedded within a final section. Hence, \( \hat{u}_{i,j}^* \) must be executed within the final section of \( \tau_{i,j} \). We have \( B_{i,j} < \hat{W}_{i,j} < E_{i,j} \). Since \( \tau_{k,l} \) can enter the final section only after it finishes its computation phase, we also have \( r_{k,l}^* < B_{k,l} \). By transitivity, we will have \( B_{i,j} < B_{k,l} \) in case 2. If \( B_{k,l} < E_{i,j} \), Lemma 4.5.1 cannot hold. Hence, we must have \( E_{i,j} < B_{k,l} \).

Case 3: \( r_{i,j}^* < \hat{w}_{k,l}^x \). Assume that \( B_{k,l} < E_{i,j} \) in \( \pi \). Then, according to Lemma 4.5.1, \( E_{k,l} < B_{i,j} \) must hold in \( \pi \). Thus, \( \tau_{i,j} \) must have performed a similarity test against the current value of \( x \) (either created by \( \hat{w}_{k,l}^x \) or any committed write issued after \( \hat{w}_{k,l}^x \)) in its validation phase. Since \( r_{i,j}^* \) and \( \hat{w}_{k,l}^x \) do not satisfy the \( free(\pi) \) relation, \( Wr_{\pi}(r_{i,j}^*) \) and \( \hat{w}_{k,l}^x \) cannot be similar under \( \Delta \) in \( \pi \). If \( Wr_{\pi}(r_{i,j}^*) \) and \( \hat{w}_{k,l}^x \) are not similar in \( \pi \), then, according to Lemma 4.4.1, any value created after \( \hat{w}_{k,l}^x \) in the system cannot be similar to \( Wr_{\pi}(r_{i,j}^*) \). Hence, \( \tau_{i,j} \) must be aborted. After \( \tau_{i,j} \) restarts, we will have \( \hat{w}_{k,l}^x < r_{i,j}^* \), a contradiction.

**Lemma 4.5.7** Let \( \pi \) be a SOPP history, and let \( T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \) be a path in \( TG(\pi) \), where each \( T \) is a transaction instance in \( \pi \) and \( n > 1 \). We have \( E_1 < B_n \).
Proof: We prove this lemma by induction on \( n \). Induction basis: \( n = 2 \). This basis follows immediately from Lemma 4.5.6. Induction hypothesis: The induction hypothesis assumes that the lemma holds when \( n = k \), for some \( k \geq 2 \).

Induction step: We prove this lemma also holds when \( n = k + 1 \). By induction hypothesis, if there is a path \( T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \) in \( TG(\pi) \), then we have \( E_1 \prec B_k \). By \( T_k \rightarrow T_{k+1} \) and Lemma 4.5.6, we have \( E_k \prec B_{k+1} \). Since \( B_k \prec E_k \), by transitivity, we have \( E_1 \prec B_{k+1} \). 

Theorem 4.5.1 Every history generated by SOPP is conflict \( \Delta \)-serializable.

Proof: Suppose \( \pi \) is a history generated by SOPP and there is a cycle in \( TG(\pi) \) such as \( T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_1 \), where \( n > 1 \). By Lemma 4.5.7, we have \( E_1 \prec B_1 \), a contradiction. Hence, \( TG(\pi) \) cannot have a cycle. By Theorem 2.4.2, \( \pi \) is conflict \( \Delta \)-serializable if and only if \( TG(\pi) \) is acyclic.

4.5.6 Schedulability of SOPP

After proving the properties and the correctness of SOPP in the previous two subsections, we now study the effect of transaction blocking on the schedulability analysis. In this subsection, we develop a sufficient condition under which a set of periodic transactions using SOPP can be executed without missing their deadlines by the rate-monotonic priority protocol.

Under the optimistic approach, each transaction must go through the computation, the validation, and the possible write phases. If all the transactions in a system are executed sequentially, the execution time \( E_{i,j} \) required by a transaction \( \tau_{i,j} \) will be \( E_{i,j} = C_{i,j} + V_{i,j} + W_{i,j} \), where \( C_{i,j} \), \( V_{i,j} \), and \( W_{i,j} \) denote the time required in the computation phase, the validation phase, and the write phase of \( \tau_{i,j} \), respectively. According to the Procedure 4.5.2, the validation time required by \( \tau_{i,j} \) is proportional
to the number of objects read by \( \tau_{i,j} \). Hence, if \( RS_{i,j} \) is the set of objects read by \( \tau_{i,j} \) and the number of elements in \( RS_{i,j} \) is denoted as \( |RS_{i,j}| \), then \( V_{i,j} = |RS_{i,j}| \times T_R \), where \( T_R \) is the time required to validate an object. Similarly, the time required in the write phase of \( \tau_{i,j} \) is proportional to the number of objects updated in \( \tau_{i,j} \). Hence, if \( WS_{i,j} \) is the set of objects updated by \( \tau_{i,j} \) and the number of elements in \( WS_{i,j} \) is denoted as \( |WS_{i,j}| \), then \( W_{i,j} = |WS_{i,j}| \times T_W \), where \( T_W \) is the time required to write a value back to the system.

If transactions are executed concurrently, then a transaction may encounter conflicts and may need to be restarted. Moreover, a transaction may also be blocked by at most one transaction. Hence, we also need to consider the effects of these two factors on the schedulability analysis.

We first consider the effect of transaction restarts. According to Lemma 4.5.5, we know that each transaction can be restarted at most once. In the worst case, each transaction may need to execute its computation phase twice. Hence, the worst case execution time required by each transaction becomes \( E_{i,j} = 2 \times C_{i,j} + V_{i,j} + W_{i,j} \). Note that, although a restarted transaction needs to execute its computation phase again, it does not need to re-execute its validation phase. This is because according to Proposition 4, at any time, only one transaction can enter its final section. Since the restarted transaction is still working in its final section, no other transaction can enter its final section (and then its write phase) and introduce conflicts with respect to the restarted transaction. Hence, the restarted transaction does not need to re-execute its validation phase.

We now consider the effect of blocking. By Lemma 4.5.3 and Lemma 4.5.4, a transaction \( \tau_{i,j} \) can be blocked for at most the duration of the final section of one lower priority transaction \( \tau_{k,l} \). Let \( L_{i,j} \) denote a transaction set which contains all the transactions whose priorities are lower than \( \tau_{i,j} \). Then, the worst case blocking
time for \( \tau_{i,j} \) is at most the duration of the longest final section of one transaction in \( L_{i,j} \). Let \( B_{i,j} \) and \( F_{k,l} \) denote the worst case blocking time for \( \tau_{i,j} \) and the duration of the longest final section of a transaction \( \tau_{k,l} \) in \( L_{i,j} \), respectively. Since a final section may contain a validation phase, a computation phase, and a write phase, we have \( B_{i,j} = F_{k,l} = C_{k,l} + V_{k,l} + W_{k,l} \). Note that, given a set of \( n \) periodic transactions, \( B_n = 0 \). This is because \( \tau_n \) is the lowest priority transaction in the system. Hence, no other lower priority transactions can block \( \tau_n \).

Theorem 2.1.1 proved by Liu has considered the effect of a transaction being preempted by higher priority transactions. However, we also need to consider the effect of a transaction being blocked by lower priority transactions. In order to derive the schedulability, three conditions need to be considered for a transaction \( \tau_i \):

- the preemption caused by higher priority transactions,
- blocking from lower priority transactions, and
- \( \tau_i \)'s own utilization.

According to Lemma 4.5.3 and Lemma 4.5.4, the blocking time experienced by \( \tau_{i,j} \) does not exceed \( B_{i,j} \). Hence, the history generated by SOPP can be treated as a special history generated by the priority ceiling protocol in which there is only one semaphore in the system. We have:

**Theorem 4.5.2** A set of \( n \) periodic transactions using SOPP can be scheduled by the rate-monotonic scheduling protocol if the following conditions are satisfied:

\[
\forall i, \ 1 \leq i \leq n, \ \left( E_1/p_1 \right) + \left( E_2/p_2 \right) + ... + \left( E_i/p_i \right) + \left( B_i/p_i \right) \leq i \left( 2^{\frac{1}{i}} - 1 \right)
\]

**Proof:** The first \((i - 1)\) terms represent the effect of preemptions from all higher priority transactions. The \(i\)th term is \( \tau_i \)'s execution time. \( B_i \) of the last term represents the worst case blocking time a transaction \( \tau_i \) may experience. If the equation
is satisfied for each transaction $\tau_i$, then Liu's theorem [33] will also be satisfied when $n = i$ and $c_i = (E_i + B_i)$. □
Chapter 5

OTHER CONSIDERATIONS IN REAL-TIME SYSTEMS

In the previous chapter we discussed several approaches to reduce concurrency control overhead and to make optimistic approaches more predictable. In this chapter, we will consider concerns other than correctness and predictability in designing real-time concurrency control protocols. We first consider aperiodic transactions that may interfere with similarity and protocols based on it in Chapter 5.1. We then discuss the data freshness problem in Chapter 5.2.

5.1 Aperiodic Transactions

As we have discussed, similarity methods are constructed based on the fact that periodic transactions represent the expected evolution of an object, so that data values produced over a few consecutive periods may be considered to be similar and interchangeable. However, aperiodic transactions are also essential in real-time systems for handling unpredictable events such as operator requests [8, 9, 44]. If the value of an object is changed unexpectedly by an aperiodic transaction, similarity and protocols based on it may be interfered. Hence, real-time concurrency control protocols should be able to handle conflicts caused by either periodic or aperiodic transactions. In this subsection, we first identify the possible conflicts that may occur between aperiodic and periodic transactions. We then discuss how to extend the similarity test to resolve these conflicts.
Under backward validation, if a validating transaction detects that an object value read by it has been updated by another transaction, then a conflict occurs. If the update is created by a periodic transaction, then similarity can be used to resolve the conflict. The validating transaction is restarted only when the conflict cannot be resolved by the similarity criterion. If the update is created by an aperiodic transaction, then the conflict must be resolved by traditional serializability before similarity can be re-established. This is because aperiodic transactions interrupt the expected evolution of objects and reflect a discontinuous change in the real world. If a transaction read a data value that is later updated by an aperiodic transaction, then the transaction must be restarted because it has used obsolete data.

In order for the validating transaction to be able to distinguish between periodic and aperiodic updates, we extend our object model so that each object \( x \) contains an additional timestamp \( \text{ats}_x \). The timestamp \( \text{ats}_x \) records the time at which an aperiodic update on \( x \) was issued. The default value of \( \text{ats}_x \) is zero. The similarity test discussed in Procedure 4.4.3 is then extended in Procedure 5.1.1 to detect possible aperiodic update.

In Procedure 5.1.1, a validating transaction first checks whether an aperiodic transaction has updated the object \( x \) read by the validating transaction. The check can be done by comparing the timestamp of \( x \) read by the validating transaction with \( \text{ats}_x \). If \( \text{ats}_x > \text{ts}(W\tau_x(r^x_{v,k})) \), an aperiodic update has occurred on \( x \) after the validating transaction read \( x \). Hence, the \( \text{SimTest}() \) function returns FALSE to restart the validating transaction. If no aperiodic update has occurred, the \( \text{SimTest}() \) function proceeds as usual to check any possible conflicts caused by the periodic transactions.

The write phases of transactions are also extended to incorporate the \( \text{ats}_x \). In Procedure 4.3.2, a value of \( x \) created by a local write is made available to the system
Procedure 5.1.1

SimTest(\tau_{v,k}) \{
    \forall x \in RS_{v,k} \text{ do }
    \text{if } (ats_x > ts(Wr_{x}(r_{v,k}))) \text{ then }
    \quad \text{return FALSE; /* aperiodic update occurs */}
    \quad \text{endif}
    \text{if } ((ts_x - ts(Wr_{x}(r_{v,k}))) > sb_x) \text{ then }
    \quad \text{return FALSE; /* not similar */}
    \quad \text{endif}
\text{enddo}
\text{return TRUE;}
\}

Figure 5.1: The similarity test procedure that considers aperiodic transactions.
only if the value is created more recently than the current value of \( x \). Since the current value may be created by either a periodic transaction or an aperiodic transaction, the timestamp of the local write must be compared with \( ts_x \) and \( ats_x \). We give the updated write procedure in Procedure 5.1.2.

**Procedure 5.1.2**

\[
\text{WritePhase}(r_{v,k}) \{
\forall x \in WS_{v,k} \ do \quad /* \text{write phase} */
\text{if} \ (ts(w^x_{v,k}) > ts_x) \text{ and } (ts(w^x_{v,k}) > ats_x) \text{ then } /* \text{Thomas' Write Rule} */
\text{write } x;
\text{endif}
\text{enddo}
\};
\]

Figure 5.2: The write phase procedure that considers aperiodic transactions.

Finally, the SOCC-FV protocol can be extended to handle conflicts caused by aperiodic transactions. As we said above, an aperiodic transaction interrupts the expected evolution of objects and reflects the newest situation of the real world. Hence, if the validating transaction is an aperiodic transaction and its write set is not empty, then any active transaction whose read set intersects with the write set of the validating transaction must be restarted to read the newest value created by the aperiodic transaction.
5.2 Data Temporal Constraints

Real-time systems are often continuous in nature. States of such systems are usually updated at regular intervals to reflect the real-world situation. If a data value used by a transaction is not recent enough to describe the real-world situation (possibly caused by the effects of transaction scheduling or concurrency control [42]), a transaction could perform inadequately and deliver meaningless results. Hence, in addition to transactions’ temporal and logical constraints, a real-time system must also consider data temporal constraints which determine whether data values are sufficiently recent to be usable [42, 41, 3, 32, 8, 9, 48].

In this chapter we first extend our object model to include the information needed for data temporal constraints. The relationship between data temporal constraints and logical constraints is then discussed in Chapter 5.2.2. A recently proposed force-wait protocol [48] that attempts to maintain data temporal constraints is described in Chapter 5.2.3. Finally, we present our own solution to this problem in Chapter 5.2.4.

5.2.1 Extended Object Model

In order to examine whether a value of an object is sufficiently recent, we must first measure the age of the value. The age of a value is the time elapsed since the value of the object was created. More formally, we have the following definition:

Definition 5.2.1 The age of a value of an object x at time t is defined as: \( \text{age}_x = t - ts_x \).

Ages of values are compared with thresholds associated with each object. The threshold specifies an age bound under which a value can be considered to be sufficiently recent in reflecting a real-world situation. A value with age less than the threshold of the object satisfies the temporal constraints of the object and is said to
be fresh in reflecting a real-world situation. Hence, the threshold associated with an object \( x \) is referred to as a freshness bound of \( x \) (\( f_{bx} \)). On the other hand, an object value is stale if its age is greater than the threshold of the object. We say a system preserves data temporal consistency if no transactions ever use stale data.

We extend the object model introduced in Chapter 3.2 to \( < ts_{x}, ats_{x}, sb_{x}, f_{bx} > \), where

- \( ts_{x} \) is a timestamp that records the most recent time at which the object \( x \) is updated by a local write of a periodic transaction instance,
- \( ats_{x} \) is a timestamp that records the most recent time at which the object \( x \) is updated by a local write of an aperiodic transaction instance,
- \( sb_{x} \) specifies a time interval within which all local writes issued by different transaction instances on the object \( x \) are considered to be similar,
- \( f_{bx} \) specifies a freshness bound.

### 5.2.2 Relationships between Data Similarity and Data Freshness

Since \( f_{b} \) and \( s_{b} \) are both defined based on the timing semantics of an application, and they are both specified as a length of time, it is natural to ask what the relationships between \( f_{b} \) and \( s_{b} \) are. In this subsection, we first define data freshness and data similarity in terms of \( f_{b} \) and \( s_{b} \), respectively. We then discuss the relationship between \( f_{b} \) and \( s_{b} \).

We define the concept of data freshness and the concept of data similarity as functions of \( \text{Fresh}(ts_{x}, t_{now}, f_{bx}) \) and \( \text{Similar}(ts_{x_{i}}, ts_{x_{j}}, s_{bx}) \), respectively, in the following definitions:
Definition 5.2.2

\[ Fresh(t_{sx_1}, t_{now}, fb_x) = \begin{cases} \text{TRUE} & \text{if } t_{now} - t_{sx_1} \leq fb_x \\ \text{FALSE} & \text{otherwise} \end{cases} \]

where \( t_{sx_1} \) is the time-stamp of \( x_i \) and \( x_i \) is the \( i \)th version of object \( x \).

Definition 5.2.3

\[ Similar(t_{sx_i}, t_{sx_j}, sb_x) = \begin{cases} \text{TRUE} & \text{if } | t_{sx_j} - t_{sx_i} | \leq sb_x \\ \text{FALSE} & \text{otherwise} \end{cases} \]

where \( t_{sx_i} \) and \( t_{sx_j} \) are the time-stamps of \( x_i \) and \( x_j \), respectively. \( x_i \) and \( x_j \) are the \( i \)th and the \( j \)th versions of object \( x \), respectively.

From definition 5.2.2 and 5.2.3, we see that one immediate difference between data freshness and data similarity is that data freshness describes the relation between the current time and the creation time of one version of an object, while data similarity describes the relation between the creation times of two versions of one object. Moreover, according to definition 5.2.2, a value \( x_i \) used by a transaction becomes stale if \( x_i \) has aged over \( fb_x \). On the other hand, according to definition 5.2.3, a transaction that uses a value \( x_i \) may violate logical consistency only when another transaction creates a value \( x_j \) that is dissimilar to \( x_i \). Hence, a transaction that has accessed \( x_i \) may violate the temporal constraint on \( x_i \) if it has not finished before \( x_i \) aged beyond \( fb_x \). However, a transaction that has accessed \( x_i \) may never violate \( x_i \)'s logical constraint if (1) \( \exists x_j \) such that \( j > i \), or (2) \( x_j \approx_{\Delta} x_i, \forall x_j \) such that \( j > i \).

We use \( fb \) and \( sb \) as separate thresholds for the \( Fresh \) and the \( Similar \) functions not only because \( Fresh \) and \( Similar \) are two different functions, but also because the ranges of values that can be assigned to them are different. The maximum and the minimum values that can be assigned to \( fb \) and \( sb \) actually depend on application semantics. However, the minimum values that can be assigned to \( fb \)
and $sb_x$ are restricted by $\max_{\tau_i \in T_x} WCET(\tau_i)$ and 0, respectively. We define $T_x$ as a set which contains all transactions that read object $x$, and $WCET(\tau_i)$ is the worst case estimated execution time of transaction $\tau_i$. Setting $fb_x = \max_{\tau_i \in T_x} WCET(\tau_i)$ is necessary so that the longest transaction $\tau_i \in T_x$ can finish its job before the object $x$ read by it becomes stale. Since $\forall \tau_i WCET(\tau_i) > 0$, $fb_x$ must be greater than 0. The minimum value that can be assigned to $sb_x$ is 0. When $sb_x = 0$, it indicates that consecutive values of $x$ are all distinct and the accesses to $x$ must be mutual excluded. Since a transaction does not necessarily violate logical constraints if it does not finish before $sb_x$, assigning $sb_x = 0$ does not prevent a transaction from finishing its job.

If $fb_x$ and $sb_x$ of some object $x$ are assigned as: $sb_x < fb_x$, then a transaction $\tau_{k,l}$ that reads $x_i$ must finish its job before the time $t \leq fb_x + ts_{x_i}$. However, if $x$ is updated by another transaction at time $t'$, where $sb_x + ts_{x_i} < t' \leq fb_x + ts_{x_i}$, and $\tau_{k,l}$ is still active, then $\tau_{k,l}$ must be aborted. This is because $x$'s logical constraint has been violated despite the fact that the temporal constraint on $x$ is still obeyed.

The situation is more interesting when $fb_x < sb_x$. As we said before, a transaction that has read $x_i$ must be aborted and restarted if it does not finish at a time $t > fb_x + ts_{x_i}$ ($x_i$ becomes stale). However, since $fb_x < sb_x$, the re-started transaction may either read $x_i$ again or read another value $x_j$ such that $i < j$ and $x_i \approx_{\Delta} x_j$. If this is the case, aborting and restarting the transaction is unnecessary. To avoid such unnecessary aborts when $fb_x < sb_x$, the freshness bound of $x$ can be extended to the boundary specified by $sb_x$. We call such a relaxed freshness bound as a similar freshness bound ($sfb$).

Hence, the final freshness bound of an object $x$ should be

$$fb_x = \begin{cases} 
fb_x & \text{if } fb_x \geq sb_x \\
\text{otherwise} & sb_x
\end{cases}$$
5.2.3 Related Work – The Force-Wait Protocol

Song [42, 43] has investigated the effectiveness of maintaining data temporal consistency under conventional two-phase locking and optimistic concurrency control protocols. They observed that two-phase locking protocols are generally more effective in maintaining data temporal consistency than optimistic protocols, even though the former may lead to higher deadline miss rates.

Xiong, et. al. study another approach [48] for maintaining data temporal consistency. They assumed that a system contains temporal and non-temporal objects, which are objects with and without temporal constraints, respectively. They further assume that each temporal object is written by only one write-only transaction, and a new version is created for the object each time the transaction writes. Hence, no concurrency control is considered for temporal objects. Under these assumptions, they propose the Force-Wait protocol.

The basic idea of this protocol is to block transactions from reading objects with temporal constraints that are impossible to meet. To predict the possibility of meeting a temporal constraint, the condition $\left[t + RE_t(\tau_{i,j}) \leq ts_x + fb_x\right]$, referred to as a force-wait condition, is checked before any read operation $\tau_{i,j}$ issued at time $t$ can proceed. $RE_t(\tau_{i,j})$ is the estimated remaining execution time of $\tau_{i,j}$. Under the force-wait protocol, a transaction that satisfies the above condition is allowed to proceed because it is predicted that the transaction that issues the read operation can finish its job before the object it reads becomes stale. If the transaction fails to pass the force-wait condition, it is blocked. The blocked transaction can resume the read operation after the object that causes the block is updated.

However, several factors may invalidate such predictions. The first factor comes from the force-wait protocol itself. Consider the situation if the $k$th read operation ($k > 1$) issued by a transaction does not satisfy the force-wait condition, the transaction
is blocked according to the force-wait protocol. We refer to this blocking period as a force-wait period. Since the force-wait period created by the $k$th read operation was not taken into account when all $k-1$ read operations were checked against the force-wait condition, some objects read by those $k-1$ read operations may become stale before the transaction is ready to commit.

A second factor comes from transaction scheduling protocol. Assume a transaction $\tau_{i,j}$ that has successfully issued $k$ read operations is preempted by other higher priority transactions for an arbitrary amount of time (referred to as a scheduling-block period). Since the scheduling block period is also not taken into account when those $k$ read operations were issued, some objects read by those read operations may become stale when the transaction is ready to commit.

Finally, we point out that the test of the force-wait condition actually relies on the estimated execution time of a transaction. If the time is over-estimated due to data-dependent loops and the conditional branches, transactions are more likely to be forced to wait for newer versions of data. As a result, more transactions may miss their deadlines.

5.2.4 Enforcing Data-Freshness in Real-Time Systems

Our goal is to design protocols that guarantee data temporal constraints without relying on estimated information about transactions. One way to achieve this goal is to have each transaction check to see whether all data objects read by it are still fresh before allowing the transaction to commit. If any of the data objects read by the transaction have become stale, the transaction is restarted to read more fresh data.

Procedure 5.2.1 gives the function that tests the data freshness of every object $x$ read by a validating transaction $\tau_{v,k}$. First, the age of $x$ read by $\tau_{v,k}$ is calculated as $age_x = t - ts(Wr_\pi(\tau_{v,k}))$, where $t$ is the current time and $ts(Wr_\pi(\tau_{v,k}))$ returns a
timestamp of a committed write from which \( r_{v,k}^x \) reads. If \( \text{age}_x > f b_x \), that means the object \( x \) read by \( \tau_{v,k} \) has become stale. The \( \text{Freshness}() \) function returns FALSE. If all objects read by \( \tau_{v,k} \) are still fresh, the function returns TRUE.

**Procedure 5.2.1**

\[
\text{Freshness}(\tau_{v,k}) \{
\forall x \in RSV_{v,k} \ do
\]
\[
\text{if } (t - ts(Wr_x(r_{v,k}^x))) > f b_x \text{ then} \]
\[
\text{return FALSE;} \quad /* t: current time */
\]
\[
\text{endif} \quad /* \text{violate temporal constraints} */
\]
\[
\text{endif}
\]
\[
\text{return TRUE;}
\}
\]

Figure 5.3: The data freshness test procedure.

A validating transaction controlled by SOCC-BV or SOPP calls Procedure 5.2.1 before it enters its write phase. If the return value from the \( \text{Freshness}() \) function is TRUE, the validating transaction proceeds to its write phase. Otherwise, the validating transaction restarts to read more fresh data. Note that, under SOPP, if a validating transaction is to be restarted due to its violation of data freshness, the validating transaction must first release the system lock and re-enable the CPU interrupt mechanism. Otherwise, no transaction can enter its final section to update the stale data. We give a validation procedure for SOPP that considers the data freshness in Procedure 5.2.2.
Procedure 5.2.2

Validate_Write(Tv,k) {
    get system lock and disable preemption; /* atomic */
    if (SimTest(Tv,k) == FALSE) then
        execute computation phase of Tv,k; /* restart */
    endif
    if (Freshness(Tv,k) == FALSE) then
        release system lock and enable preemption; /* atomic */
        restart Tv,k; /* restart */
    endif
    WritePhase(Tv,k); /* write phase */
    Tv,k commits;
    release system lock and enable preemption; /* atomic */
}

Figure 5.4: A SOPP validation procedure that considers data freshness.
The validation procedure of SOCC-FV can also call Procedure 5.2.1 to test the data freshness of objects read by validating transactions. However, the freshness test should be performed before the similarity test. Otherwise, a validating transaction that has used stale data (hence, itself needs to be restarted) may unnecessarily abort other active transactions during its similarity test.
Chapter 6

SIMULATION EXPERIMENTS

In this chapter we study the real-time performance of protocols proposed in this thesis. We first discuss a simulation framework used in this study [29]. We then discuss the simulation setup used in our simulation experiments. We also provide the performance metrics measured in our simulations. Finally, we present the results of several sets of experiments and an analysis of the results.

6.1 Implementation of the Simulator

Concurrency management in real-time systems involves a complex set of policies: scheduling, overload management, and concurrency control. The scheduling policy determines the transactions to which resources, particularly the CPU, are allocated. The overload management policy determines the actions to be taken when the system is overloaded, i.e., when some timing constraints cannot be met. The concurrency control policy resolves competing requests for resources according to correctness conditions that assure the consistency of the data maintained by the application. Interactions among these policies affect the real-time performance of an application. More deadlines may be met under some combinations of policies for a particular application than under other combinations. Hence, we implement an object-oriented framework [29] that enables the customization of modules which implement policies for scheduling, overload management, and a variety of concurrency control policies. Figure 6.1 gives a top-level view of our simulator.
Simulation of a real-time application starts with a user-provided input file that describes a target real-time application. A target application is described in terms of execution environment, objects, and transactions in our simulation framework.

The execution environment describes the attributes of a system on which the simulated application is ultimately to run. In the current version of the simulator, the target execution environment may be either a single processor or multiprocessor computer. Objects correspond to real-world entities and will be accessed by concurrently executing transactions.

Each transaction can read values of objects, process the values, calculate new values, perform validations, and write the new values back to the system. We provide three basic transaction phases: compute, validate, and writeback. The objects which are the targets of each read or write operation must be identified in the input file. The validate and writeback phases are input placeholders that delineate the end of the transaction and provide points at which operations of a specific concurrency
control protocol may be simulated. The control actions to be simulated in each phase are defined through the customization of the simulator for a particular concurrency control policy. The validate and writeback phases, in particular, may be empty, if appropriate in the protocol.

The simulation kernel is based on the *Structural Active Object System (SAOS)* [27, 28]. It acts as an underlying real-time operating system to invoke transactions and policy managers at appropriate time. Specific real-time policies can be implemented by refining classes of our simulation framework.

The Dispatcher decides which transaction to execute at each simulated scheduling opportunity. The default dispatcher assumes that the ready queue is a priority queue and simply assigns the first transaction on the ready queue to the next available processor. The priority manager implements real-time scheduling policies by manipulating the relative priorities of the simulated transactions and ordering transaction entries into system queues according to these priorities. Versions of the rate monotonic and the earliest deadline first (EDF) algorithms are available with the simulation framework. The Overload Manager implements policies for both detection and reaction to transaction deadline violations. The default policy for detection and reaction is to abort a transaction immediately after its deadline is missed.

The simulation kernel invokes the Priority Manager and the Overload Manager at each time step to set priorities of transactions and check for tardy transactions. The next eligible transaction, as determined by the Dispatcher, is then executed for one time unit. A transaction counter is updated so that the transaction remembers the phase to be resumed and the number of execution time units remaining when it next becomes eligible to execute. Simulator primitive operations like read, write, and compute are executed within the transaction phases. As a simulated system executes, the simulator produces an event stream that characterizes the application’s progress.
Information preserved by each event includes the time and the location at which the event takes place, the transaction that issues the event, and the target-object on which the event operates.

At the conclusion of a simulation run, the performance monitor produces an analysis file containing performance statistics for the target system. Items monitored by the simulator include (1) number of ready instances, (2) number of deadlines missed, (3) number of restarts, and (4) CPU utilization. The performance monitor also records the values of individual performance measures at user-defined intervals in separate files. This segregated data is available for additional processing and / or graphical display.

6.2 Simulation Settings

In this section, we describe important parameters that characterize the target real-time application we simulate. We first describe the parameters of an underlying system on which a simulated application is to be executed. Parameters that define the characteristics of a transaction set are then discussed. We also give the performance metrics measured in our simulations.

6.2.1 System Configuration

In our study, we assume that systems are resident in main memory and a transaction is aborted as soon as its deadline expires. This corresponds to a firm real-time system in which finishing a transaction after its deadline expires does not impart any value to the system [48]. Table 6.1 lists five system parameters used in our simulation experiments.

The number of CPUs in the system and the system utilization to be simulated are specified by \( C \) and \( U \), respectively. Our experiments simulate transaction sets
Table 6.1: System parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>number of CPUs in system</td>
</tr>
<tr>
<td>$U$</td>
<td>system utilization</td>
</tr>
<tr>
<td>$T$</td>
<td>number of transactions in system</td>
</tr>
<tr>
<td>$J$</td>
<td>number of objects in system</td>
</tr>
<tr>
<td>$H$</td>
<td>scheduling protocol</td>
</tr>
</tbody>
</table>

executing on one, two, or four processors. We also experiment with systems under normal-load, under-load, and over-load situations. The system utilizations for normal-load, under-load, and over-load situations are calculated as $U_{\text{normal}} = 1 \times C$, $U_{\text{under}} = 0.9 \times C$, and $U_{\text{over}} = 1.1 \times C$, respectively. The number of transactions is specified by $T$ and the number of objects in a system is specified by $J$.

Preemption is based on transaction priorities which are assigned according to either the rate monotonic (RM) or the earliest deadline first (EDF) scheduling protocol. We use $H$ to indicate the scheduling protocol selected in our simulations.

6.2.2 Transaction Configuration

Transaction characteristics are controlled by the parameters listed in Table 6.2. The period and computation time of each transaction is selected by the random number generator of our simulator. The period of each transaction is distributed uniformly between $P_{\text{min}}$ and $P_{\text{max}}$. The execution time of each transaction is chosen uniformly between $E_{\text{min}}$ and $E_{\text{max}}$. After the periods and execution times of the transactions are generated, the periods of transactions are scaled such that the total
utilization does not exceed the system utilization specified by $U$. In other words, 
transaction periods are uniformly distributed over the interval $[1, B]$, where $B$ is 
the period ratio, the ratio of the longest period to the shortest period in a transaction 
set [42]. In our simulation, the deadline for a transaction is defined to be the end of 
its period.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{min}, P_{max}$</td>
<td>range of transaction period</td>
</tr>
<tr>
<td>$E_{min}, E_{max}$</td>
<td>range of transaction execution time</td>
</tr>
<tr>
<td>$R_{min}, R_{max}$</td>
<td>number of objects read by a transaction</td>
</tr>
<tr>
<td>$W_{min}, W_{max}$</td>
<td>number of objects written by a transaction</td>
</tr>
</tbody>
</table>

Table 6.2: Transaction parameters

Each transaction consists of a sequence of object read and write accesses. The 
number of objects read (written) by a transaction varies uniformly between $R_{min}$ and 
$R_{max}$ ($W_{min}$ and $W_{max}$). Object accesses are generated from a uniform distribution 
spanning across all the objects in the system. The order of read, write, and computa-
tion operations of each transaction is randomly constructed. The similarity bounds 
for objects are ranged uniformly from $S_{min}$ to $S_{max}$.

Transactions are more likely to share common data objects if (1) the number of 
objects in a system is small, or (2) the number of objects accessed by each transaction 
is increased. Hence, by changing these values we can smoothly vary the degree of 
competition for resources. We can also change values of similarity bounds to observe 
how well a protocol can utilize the concept of similarity, i.e., if similarity bounds for
objects are increased, conflicts are more likely to be resolved and more transactions can meet their deadlines.

6.2.3 Performance Metric

Unlike conventional performance evaluations of concurrency control protocols, our experiments do not measure transaction response time. Instead, the primary performance metric is the percentage of transactions which miss their deadlines. We also measure the total number of restarts caused by SOPP and we study how a protocol behaves in systems with different workloads.

All of our experiments were conducted with two scheduling protocols: the rate monotonic protocol and the earliest deadline first protocol. For each experiment we ran the simulation with the same parameters for 10 different random number seeds. Each run is executed for 100,000 simulation time units. For each protocol tested, performance statistics were collected and averaged over the 10 runs. These averages are then plotted in the performance graphs presented in the following subsections.

In order to draw statistical conclusion about population parameters from our simulations, we use t distribution to conduct two types of inferences: (1) interval estimation, and (2) hypothesis testing. For interval estimation, we obtain a 95% confidence interval for the difference on the mean deadline miss rates between SSP and SOPP. Hence, we are 95% confident that the lower and upper confidence limits obtained from our simulations will cover the true population difference. For testing a hypothesis, we claim that SSP has higher deadline miss rates than SOPP. We then use a 0.5% level of significance to decide whether our claim is strongly supported by our simulation results. We formulate the hypotheses as follows:

\( \mu_1 = \) population mean deadline miss rate of SSP,

\( \mu_2 = \) population mean deadline miss rate of SOPP,
Null hypothesis ($H_0$): $\mu_1 - \mu_2 = 0$
Alternative hypothesis ($H_1$): $\mu_1 - \mu_2 > 0$

Consulting the $t$ table, we find the rejection region is $T \geq 2.878$. In other words, if the $T$ value obtained from our simulations is greater than 2.877, then $H_0$ is rejected and $H_1$ stands.

6.3 Baseline Model

In this section, we develop a baseline model around which we conducted further experiments. Table 6.3 lists the values assigned to the workload parameters in our baseline model. These settings are compatible with the settings described in [42, 25]. The abbreviations will be used to refer to different parameter settings in the following simulation graphs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>value</th>
<th>Parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>2</td>
<td>$P_{min}$, $P_{max}$</td>
<td>40, 100</td>
</tr>
<tr>
<td>$U$</td>
<td>2</td>
<td>$E_{min}$, $E_{max}$</td>
<td>5, 25</td>
</tr>
<tr>
<td>$T$</td>
<td>15</td>
<td>$R_{min}$, $R_{max}$</td>
<td>0, 2</td>
</tr>
<tr>
<td>$J$</td>
<td>15</td>
<td>$W_{min}$, $W_{max}$</td>
<td>0, 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S_{min}$, $S_{max}$</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

Table 6.3: Baseline parameters

We first compare the performance of the optimistic protocols developed in this thesis: SOCC-FV, SOCC-BV, and SOPP. Transaction priorities are assigned according to the rate monotonic and the earliest deadline first scheduling protocols in our
experiments. Table 6.4 summarizes the deadline miss rates produced by SOCC-FV, SOCC-BV, and SOPP.

<table>
<thead>
<tr>
<th>protocol</th>
<th>%miss (RM)</th>
<th>%miss (EDF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCC-FV</td>
<td>11.45</td>
<td>7.26</td>
</tr>
<tr>
<td>SOCC-BV</td>
<td>13.95</td>
<td>9.88</td>
</tr>
<tr>
<td>SOPP</td>
<td>14.10</td>
<td>10.04</td>
</tr>
</tbody>
</table>

Table 6.4: Deadline miss rates between SOCC-FV, SOCC-BV, and SOPP

Our simulation experiments show that SOCC-FV and SOCC-BV can help more transactions to meet their deadlines than SOPP. This is because, under SOPP, a restarted transaction has to prevent other transactions from working in their final sections. This requirement results in lower concurrency level compared with SOCC-FV and SOCC-BV. Hence, in our following experiments, we compare the performance of this most conservative protocol (with respect to SOCC-FV and SOCC-BV) with SSP.

6.4 Comparison of SSP and SOPP

In this section, we evaluate the performance of SSP and SOPP under a variety of operating conditions, workloads, and data access patterns. Since taking the cross product of these settings yields millions of different test sets we have selected the graphs which best illustrate the differences between these protocols.
6.4.1 *Effect of Similarity Bounds*

In order to measure how well each protocol can utilize the concept of similarity, we experiment our test sets with different similarity bounds, ranging from 0, 1, 2, 3, 4, to "v" period. For the "v" period, the similarity bounds of objects are selected randomly from 0 to 4. This setting reflects the real-world situation where objects may have different similarity tolerance. All other parameters are set as the values listed in Table 6.3. The results are shown in Figure 6.2.

When similarity bounds are 0, both SSP and SOPP have high deadline miss rates. However, SOPP helps more transactions to meet their deadlines than SSP. This is because SOPP still allows concurrent (serializable) executions of transactions even when all similarity bounds are set to zero. When a conflict occurs, transactions involved in the conflict are restarted. This characteristic of SOPP may contribute to the deadline miss rate. SSP, on the other hand, allows only sequential executions of conflicting transactions when similarity bounds are all zero, resulting in lower processor utilization rate and higher deadline miss rate.

When the similarity bounds are increased up to 2 periods, SSP is still unable to utilize similarity and generates only sequential histories (because \( (s_b - 2p) / 2 = 0 \)). SOPP, on the other hand, starts using the concept of similarity as soon as the similarity bounds for objects are increased to 1 period. When the similarity bounds of objects are increased, the number of restarts due to conflicts is reduced in SOPP (see Table 6.5 for the percentage of transactions being restarted). Hence, the pessimistic approach is less likely to be taken in SOPP, resulting in less blocking time. The deadline miss rate is also reduced correspondingly.

SSP starts using similarity and improving the schedulability of transaction sets only after the similarity bounds are larger than 2 periods. When the similarity bounds are increased to more than 2 periods, SSP and SOPP have almost identical
Figure 6.2: Effect of similarity bounds. (top: RM, bottom: EDF, left: mean miss rates, right: confidence intervals)
performance. This is because almost all conflicts can be resolved by similarity, and the concurrency control protocols rarely cause blockings or restarts.

So far we have discussed the situations where all objects have the same similarity bounds. However, in the real world situation, objects are more likely to have different similarity bounds. We varied similarity bounds at random between 0 and 4. In this situation, SOPP can perform much better than SSP. This is because the SSP rule, \((sb_x - 2p_x) / 2\), prevents system concurrency from growing proportional to the number of processors, especially when \(sb_x\) is small. If one object has a smaller similarity bound than other objects in the same interactive set, then the recency bound for the entire set is set to the smallest value (very often 0). Therefore, transactions are blocked often and processors are not properly utilized, resulting in a higher deadline miss rate. SOPP, on the other hand, can use similarity to resolve conflicts even when similarity bounds are still small (e.g. \(sb_x = 1\)).

Table 6.6 gives the results of statistical inferences. When the similarity bounds are less than or equal to 2 periods, and when similarity bounds are various, the confidence intervals confirm that SSP has substantially higher deadline miss rates than SOPP. The conclusion of \(t\) test also supports our claim. However, when all the objects have similarity bounds set to 3 or 4 periods, the intervals show that, under RM, SOPP has slightly higher deadline miss rates than SSP. The \(t\) test also concludes that the

<table>
<thead>
<tr>
<th>Similarity Bounds</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart%(under RM)</td>
<td>7.82</td>
<td>1.93</td>
<td>0.08</td>
<td>0.01</td>
<td>0</td>
<td>1.39</td>
</tr>
<tr>
<td>restart%(under EDF)</td>
<td>3.85</td>
<td>1.08</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>0.63</td>
</tr>
</tbody>
</table>

Table 6.5: Percentage of transactions being restarted under SOPP.
### Table 6.6: Inference results of different similarity bounds.

<table>
<thead>
<tr>
<th>SB</th>
<th>Confidence interval%</th>
<th>$t$ test</th>
<th>EDF</th>
<th>Confidence interval%</th>
<th>$t$ test</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11.37</td>
<td>5.21</td>
<td>rejected</td>
<td>12.55</td>
<td>5.03</td>
</tr>
<tr>
<td>1</td>
<td>20.57</td>
<td>8.50</td>
<td>rejected</td>
<td>19.50</td>
<td>6.77</td>
</tr>
<tr>
<td>2</td>
<td>23.27</td>
<td>9.45</td>
<td>rejected</td>
<td>22.62</td>
<td>7.54</td>
</tr>
<tr>
<td>3</td>
<td>-2.60</td>
<td>-7.43</td>
<td>not rejected</td>
<td>-0.24</td>
<td>0.23</td>
</tr>
<tr>
<td>4</td>
<td>-1.61</td>
<td>-5.66</td>
<td>not rejected</td>
<td>-0.24</td>
<td>0.23</td>
</tr>
<tr>
<td>v</td>
<td>21.14</td>
<td>8.62</td>
<td>rejected</td>
<td>20.75</td>
<td>7.07</td>
</tr>
</tbody>
</table>

claim $H_1$ is not substantiated. Under EDF, although $H_0$ is not rejected, the intervals indicate that SSP and SOPP perform equally well.

### 6.4.2 Effect of Number of CPUs

In this section, we experiment systems with different number of processors. We first discuss the situation where the number of processors is increased to 4. The load for the 4-processors case is set to 4. (For the performance of different workloads with different numbers or processors, see Chapter 6.4.4.) All other parameters are set as the values listed in Table 6.3. Figure 6.3 depicts the performance of SSP and SOPP under the RM and EDF scheduling protocols.

The results show that SOPP can perform substantially better than SSP in most of cases. This is because, under the SSP rules, a transaction executing on one processor frequently forces other processors into idle status, resulting in lower processor
Figure 6.3: Effect of increasing number of processors to 4. (top: RM, bottom: EDF, left: mean miss rates, right: confidence intervals)
utilization rate and contributing to higher deadline miss rate. On the other hand, SOPP allows conflicting transactions running on different processors to work simultaneously in their computation phases and, therefore, improves the schedulability of transaction set.

However, the performance gained by SOPP does not come for free. When the similarity bounds of all objects are set to 4, SSP performs slightly better than SOPP under the rate monotonic scheduling protocol. Despite the fact that most conflicts can be resolved by similarity when the similarity bounds become 4, there is still a small chance that transactions can be restarted and hence block other transactions under SOPP. The impact of blocking is more likely to be observed when transactions are scheduled under the RM protocol, which originally has lower schedulability than the EDF protocol.

Table 6.7 shows that SSP has substantially higher deadline miss rate than SOPP in most cases except when the similarity bounds are set to 4 periods. Under RM, when similarity bounds set to 4, neither the interval estimation nor the $t$ test, supports our claim. However, under EDF, the intervals indicate that SSP and SOPP perform equally well.

Next, we experiment systems with only one processor. The load for the single processor experiments is set to 1. Again, all other parameters are set as the values listed in Table 6.3. We present the performance of SSP and SOPP under the RM scheduling protocol in Figure 6.4. Transactions scheduled by EDF can all meet their deadlines in our experiment.

In all our experiments, we found that SSP performs slightly better than SOPP under the RM scheduling protocol in a single processor system. We think the reason is because the cost of restarts and blocking under SOPP may overload the capacity of the single processor, while under SSP, there is only the cost of blocking. This impact
<table>
<thead>
<tr>
<th>SB</th>
<th>interval%</th>
<th>T</th>
<th>$H_0$</th>
<th>interval%</th>
<th>T</th>
<th>$H_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21.10</td>
<td>38.16</td>
<td>7.30</td>
<td>rejected</td>
<td>33.82</td>
<td>57.58</td>
</tr>
<tr>
<td>1</td>
<td>36.16</td>
<td>50.50</td>
<td>12.70</td>
<td>rejected</td>
<td>45.49</td>
<td>68.32</td>
</tr>
<tr>
<td>2</td>
<td>41.10</td>
<td>55.11</td>
<td>14.43</td>
<td>rejected</td>
<td>52.74</td>
<td>74.47</td>
</tr>
<tr>
<td>3</td>
<td>14.38</td>
<td>32.66</td>
<td>5.41</td>
<td>rejected</td>
<td>17.36</td>
<td>36.95</td>
</tr>
<tr>
<td>4</td>
<td>-1.98</td>
<td>-0.59</td>
<td>-3.90</td>
<td>not rejected</td>
<td>-0.55</td>
<td>0.41</td>
</tr>
<tr>
<td>v</td>
<td>37.69</td>
<td>51.71</td>
<td>13.40</td>
<td>rejected</td>
<td>48.77</td>
<td>71.72</td>
</tr>
</tbody>
</table>

Table 6.7: Inference results of increasing number of processors to 4.

Figure 6.4: Effect of decreasing number of processor to 1. (RM, left: mean miss rates, right: confidence intervals)
Table 6.8: Inference results of decreasing number of processor to 1.

of restarts is more likely to be observed when transactions are scheduled under the RM protocol. The statistical inferences shown in Table 6.8 also tell us that the evidence against $H_0$ is weak.

6.4.3 Effect of Increasing Conflicts

We also want to determine how resource competition affects SSP and SOPP. For this purpose, we have carried out simulations with different levels of data contention by varying the maximum number of objects read and written by each transaction from 2 to 4. All other parameters are set as the values listed in Table 6.3. In the following we present the results where the maximum number of objects read (and written) by a transaction is set to 2, 3 and 4.

Figure 6.5 and 6.6 depict the performance of SSP and SOPP under the RM and the EDF scheduling protocols, respectively. As we can see, under SSP, the deadline miss rate is increased almost 10% each time the number of objects accessed by a
Figure 6.5: Effect of increasing conflicts under RM. (top: \( R_{\text{max}} = W_{\text{max}} = 2 \), middle: \( R_{\text{max}} = W_{\text{max}} = 3 \), bottom: \( R_{\text{max}} = W_{\text{max}} = 4 \), left: mean miss rates, right: confidence intervals)
Figure 6.6: Effect of increasing conflicts under EDF. (top: $R_{\text{max}} = W_{\text{max}} = 2$, middle: $R_{\text{max}} = W_{\text{max}} = 3$, bottom: $R_{\text{max}} = W_{\text{max}} = 4$, left: mean miss rates, right: confidence intervals)
<table>
<thead>
<tr>
<th>SB</th>
<th>Confidence</th>
<th>RM t test</th>
<th>EDF t test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>interval%</td>
<td>T</td>
<td>H₀</td>
</tr>
<tr>
<td>0</td>
<td>11.37</td>
<td>5.21</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>26.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>20.57</td>
<td>8.50</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>34.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>23.27</td>
<td>9.45</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>36.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-2.60</td>
<td>-7.43</td>
<td>not rejected</td>
</tr>
<tr>
<td></td>
<td>-1.45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-1.61</td>
<td>-5.66</td>
<td>not rejected</td>
</tr>
<tr>
<td></td>
<td>-0.74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>21.14</td>
<td>8.62</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>34.76</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SB</th>
<th>Confidence</th>
<th>RM t test</th>
<th>EDF t test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>interval%</td>
<td>T</td>
<td>H₀</td>
</tr>
<tr>
<td>0</td>
<td>18.02</td>
<td>8.71</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>29.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>30.96</td>
<td>17.33</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>39.51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>34.16</td>
<td>19.41</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>42.45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-2.84</td>
<td>-9.63</td>
<td>not rejected</td>
</tr>
<tr>
<td></td>
<td>-1.82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-1.75</td>
<td>-8.71</td>
<td>not rejected</td>
</tr>
<tr>
<td></td>
<td>-1.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>32.18</td>
<td>17.68</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>40.86</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SB</th>
<th>Confidence</th>
<th>RM t test</th>
<th>EDF t test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>interval%</td>
<td>T</td>
<td>H₀</td>
</tr>
<tr>
<td>0</td>
<td>13.96</td>
<td>7.22</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>25.41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>34.96</td>
<td>38.99</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>38.95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>42.35</td>
<td>66.44</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>45.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-3.12</td>
<td>-16.22</td>
<td>not rejected</td>
</tr>
<tr>
<td></td>
<td>-2.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-1.89</td>
<td>-9.64</td>
<td>not rejected</td>
</tr>
<tr>
<td></td>
<td>-1.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>34.54</td>
<td>22.67</td>
<td>rejected</td>
</tr>
<tr>
<td></td>
<td>41.60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.9: Inference results of increasing conflicts. (top: $R_{max} = W_{max} = 2$, middle: $R_{max} = W_{max} = 3$, bottom: $R_{max} = W_{max} = 4$)
The deadline miss rate under the control of SOPP increases smoothly against the number of conflicts. These results show the fact that SSP is more sensitive to the number of conflicts than SOPP. The reason is because when the number of objects accessed by each transaction is increased, it is more likely for transactions to fall into the same interactive set under SSP. If one object has a very low similarity bound in one interactive set, the recency bound for the entire set can become very small and less concurrency is allowed. As the size of interactive sets grows, the impact may affect more transactions and result in even lower concurrency.

The statistics of inference are shown in Table 6.9. In most cases, the results confirm that SSP has higher deadline miss rates than SOPP. Under RM, we obtain negative lower and upper confidence limits as well as negative T value when the similarity bounds are increased to 3 or 4 periods. In other words, SSP performs better than SOPP under these situations.

6.4.4 Effect of Different Workloads

In the previous experiments, we studied the performance of protocols under the normal load situation. That is the system utilization is $U = 1 \times C$. However, in reality, the workload of a system may be higher or lower than the system capacity. In this section we experiment with systems with under-load and over-load situations. The system utilizations for under-load and over-load situations are calculated as $U_{\text{under}} = 0.9 \times C$, and $U_{\text{over}} = 1.1 \times C$, respectively. $C$ is set to 2 as the baseline model. All other parameters are set as the values listed in Table 6.3. Figure 6.7 and 6.8 depict the results of different workloads under RM and EDF, respectively.

As we can see, SOPP can help more transactions to meet their deadlines in most of cases under different workloads. When the similarity bounds are set to 3 or 4, SSP performs slightly better than SOPP under the RM scheduling protocol. We
Figure 6.7: Different workloads under RM. (top: $U = 1.8$, middle: $U = 2$, bottom: $U = 2.2$, left: mean miss rates, right: confidence intervals)
Figure 6.8: Different workloads under EDF. (top: $U = 1.8$, middle: $U = 2$, bottom: $U = 2.2$, left: mean miss rates, right: confidence intervals)
explained the reason for this above. From the results presented, we notice also that the deadline miss rate of SOPP increases slightly quicker than that of SSP when workloads are increased. SOPP is more sensitive to the change of workloads. This phenomenon also suggests that when the number of processors is increased, SOPP is more likely to help more transactions to meet their deadlines because the workload can be distributed over multiple processors.

To confirm our conjecture, we conducted another set of experiments for systems with different workloads \((U = 2, 3, 4)\) running over different number of processors \((C = 2, 3, 4)\). The similarity bounds of objects are randomly selected between the range from 0 to 4. All other parameters are set to the values listed in Table 6.3. We present the results in Figure 6.9.

From Figure 6.9, we form two conclusions. First, when the number of processors is increased, SSP cannot fully utilize these processors, resulting in poor improvement of deadline miss rate. For example, with the utilization rate \(U = 3\), despite an increase in the number of processors from 2 to 4, the deadline miss rate remains almost the same under the control of SSP. This is because, under the SSP rules, a transaction executing on one processor frequently forces other processors into idle status, resulting in lower processor utilization rate. On the other hand, under SOPP, each time a new processor is added to the system, the deadline miss rate is improved significantly.

Secondly, we notice that, under EDF, when a system has fewer processors \((C = 2)\) and the system is highly over-loaded (when \(U = 3, 4\)), SSP performs slightly better than SOPP. This is because the cost of restarts created by SOPP can add an extra load to an already over-loaded system, resulting in an even heavier load. Hence, a more conservative protocol like SSP should be used when a system is highly overloaded. The statistics of inferences are shown in Table 6.10.
Figure 6.9: Effect of increasing number of processors over different workloads. (top: RM, bottom: EDF. $Ux$ indicates $U = x$.)
<table>
<thead>
<tr>
<th>SB</th>
<th>RM Interval%</th>
<th>T</th>
<th>H₀</th>
<th>EDF Interval%</th>
<th>T</th>
<th>H₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14.69</td>
<td>29.69</td>
<td>6.22</td>
<td>rejected</td>
<td>18.80</td>
<td>34.74</td>
</tr>
<tr>
<td>1</td>
<td>21.28</td>
<td>35.19</td>
<td>8.53</td>
<td>rejected</td>
<td>19.04</td>
<td>34.97</td>
</tr>
<tr>
<td>2</td>
<td>22.04</td>
<td>35.93</td>
<td>8.77</td>
<td>rejected</td>
<td>19.04</td>
<td>34.97</td>
</tr>
<tr>
<td>3</td>
<td>-0.50</td>
<td>-0.23</td>
<td>-5.69</td>
<td>not rejected</td>
<td>0.01</td>
<td>-0.00</td>
</tr>
<tr>
<td>4</td>
<td>-0.53</td>
<td>-0.16</td>
<td>-3.89</td>
<td>not rejected</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>v</td>
<td>21.34</td>
<td>35.27</td>
<td>8.54</td>
<td>rejected</td>
<td>19.04</td>
<td>34.97</td>
</tr>
<tr>
<td>0</td>
<td>11.37</td>
<td>26.72</td>
<td>5.21</td>
<td>rejected</td>
<td>12.55</td>
<td>30.56</td>
</tr>
<tr>
<td>1</td>
<td>20.57</td>
<td>34.08</td>
<td>8.50</td>
<td>rejected</td>
<td>19.50</td>
<td>37.03</td>
</tr>
<tr>
<td>2</td>
<td>23.27</td>
<td>36.59</td>
<td>9.45</td>
<td>rejected</td>
<td>22.62</td>
<td>40.10</td>
</tr>
<tr>
<td>3</td>
<td>-2.60</td>
<td>-1.45</td>
<td>-7.43</td>
<td>not rejected</td>
<td>-0.24</td>
<td>0.23</td>
</tr>
<tr>
<td>4</td>
<td>-1.61</td>
<td>-0.74</td>
<td>-5.66</td>
<td>not rejected</td>
<td>-0.24</td>
<td>0.23</td>
</tr>
<tr>
<td>v</td>
<td>21.14</td>
<td>34.76</td>
<td>8.62</td>
<td>rejected</td>
<td>20.75</td>
<td>38.31</td>
</tr>
</tbody>
</table>

Table 6.10: Inference results of different workloads. (top: \( U = 1.8 \), middle: \( U = 2 \), bottom: \( U = 2.2 \))
6.4.5 Effect of Overestimating Transaction Execution Time

From the experiments we have discussed so far, we observe that SSP performs better when similarity bounds in a system are less restrictive. Unfortunately, it is only partially true. In the experiments described above, we assume we have estimated the execution time of each transaction perfectly; the estimated times are the same as the real execution times. This allows SSP to use precise information in scheduling transactions. However, as Stoyenko and Haban pointed out in [13, 46], the transaction information might be over-estimated due to data-dependent loops and conditional branches in each transaction. What is the result if execution times of transactions are over-estimated?

We modeled situations in which the execution times of transactions are over-estimation by varying degrees. For the first situation, the WCET of each transaction is slightly over-estimated with the degree of over-estimation selected randomly between 0% to 25%. In the second situation, the maximum rate of over-estimation is increased to 50%. To observe the effects of over-estimation within the under-loaded, the normal-loaded, and the over-loaded situations, we set \( U = 3 \) with \( C = 2, 3, \) and 4. The similarity bounds of all objects are set as \( S_{min} = S_{max} = 3 \). Other parameters remain the same as the values listed in Table 6.3.

From the results presented in Figure 6.10, we can make the following conclusions. With increasing relaxed similarity bounds, since fewer transactions are blocked under SSP and fewer transactions are restarted under SOPP, both SSP and SOPP allow higher concurrency and more transactions meet their deadlines. However, as the experiments demonstrated, SSP is unable to fully utilize the more relaxed similarity bounds. This is because SSP relies on WCETs, and a slightly over-estimated WCET may quickly overshadow the benefits provided by the increasing similarity bounds.
Figure 6.10: Effect of over-estimating transaction execution time. (top: RM, bottom: EDF)
SOPP, on the other hand, is independent of WCETs and is able to fully utilize the increased similarity bounds.

Secondly, when WCETs of transactions are over-estimated, SSP no longer performs slightly better than SOPP in the situation where RM is used and the similarity bounds of all objects are set to 3 for the reason we presented in the preceding paragraph. However, when the similarity bounds of all objects are set to 4, SSP can tolerate larger over-estimations. Finally, we observe that the over-estimations of WCETs may in fact reduce the deadline miss rate when a system is heavily overloaded (for example, see $C = 2$). The impact is more obvious when EDF is used. This phenomenon matches the conclusion we obtained in the end of last subsection. That is, a protocol that allows lower concurrency should be used when a system is overloaded. Koren, et. al. [21] reached similar conclusion.

6.5 Summary

Our simulation experiments compare the performance of the SSP and the SOPP protocols in both single and multiple processor environments. While there is no significant difference in performance in the single processor environment, SOPP performs substantially better than SSP in multiprocessor environments.

Moreover, in situations in which the similarity bounds may be varied, SOPP can perform much better than SSP. Intuitively, this is what we expect since the conservative rules of SSP tend to serialize computations, where as SOPP starts using similarity bounds even when the bounds are still very small (e.g. $sb_x = 1p$).

Finally, we found that SSP can perform slightly better than SOPP under the RM scheduling protocol if (1) all objects have the same similarity bounds up to 3 or 4, and (2) all transaction execution times are perfectly estimated.
Chapter 7

CONCLUSION

In this dissertation, we have reviewed the concept of similarity and explained its advantages against the traditional serializability. We also discussed the rules of the similarity stack protocol (SSP) that utilize similarity. Since SSP rules rely on many worst case assumptions and estimated information, SSP is unable to fully utilize the benefits provided by similarity as we have discussed in Chapter 2 and Chapter 6. The advantages of similarity and the drawbacks of SSP motivate us to design other similarity-based protocols that can better utilize similarity without relying on any prior information. Since similarity can reduce the number of transaction restarts, and the optimistic approaches usually do not require prior information of transactions, we explored the ideas of integrating optimistic approaches with similarity in this thesis.

We started our discussions by reviewing the forward-validation mechanism in which a validating transaction validates its write set against the read sets of active transactions. In order to test whether two conflicting events can be resolved by similarity, we integrated a similarity test procedure and the Thomas's Write Rule (TWR) with the traditional forward-validation mechanism in our Similarity-based Optimistic Concurrency Control-Forward Validation (SOCC-FV) protocol. The similarity test procedure calculates the temporal distance between two conflicting events. If the temporal distance of two conflicting events are greater than a specified similarity bound, one of the conflicting transactions is restarted. The Thomas's Write Rule is used to prevent the data regression problem. SOCC-FV has advantages of
detecting conflicts earlier and providing flexibility in selecting restarted transactions. However, because the transactions being validated by the validating transaction are still active, the sizes of their read sets and the number of active transactions keep changing, making schedulability analysis difficult.

We then examined the backward-validation mechanism in which a validating transaction validates its read set against the write sets of committed transactions. Similar to SOCC-FV, we integrated a similarity test procedure and the Thomas's Write Rule (TWR) with the traditional backward-validation mechanism in our Similarity-based Optimistic Concurrency Control-Backward Validation (SOCC-BV) protocol. SOCC-BV is more static than SOCC-FV because the sizes of write sets of committed transactions do not change. However, the number of committed transactions that need to be validated by a validating transaction is still unknown. In order to solve this problem, each transaction, before entering its validation phase, is required to enter a validation lock phase in which validation locks are set on the objects it needs to validate. As a result, a validating transaction only needs to validate objects in its read set against the current object value in the system. This mechanism significantly reduces the implementation overhead.

Next, we extended SOCC-BV with an optimistic-then-pessimistic approach in which a validating transaction sets a system lock before entering its validation phase. The system lock is released by the validating transaction only after the validating transaction commits. Under this approach, a transaction is first executed in an optimistic stage where it assumes conflicts are rare events. If a conflict does occur, the validating transaction is restarted without releasing the validation locks it obtains in its optimistic stage. Hence, a restarted transaction is now executed in a pessimistic stage in which it assumes that conflicts might occur again. This Similarity-base Optimist-then-Pessimistic Protocol (SOPP) solves both the repeated restarts and
unbounded blocking problems because only one transaction can work in its final section at any time. The schedulability of SOPP is also derived in this thesis.

We also demonstrated that, in Chapter 5, our protocols can be easily extended to handle aperiodic transactions and data freshness requirements. Finally, we conducted extensive simulation experiments to compare the deadline miss rates between SSP and SOPP. The simulation results confirmed that SOPP can outperform SSP in most cases, especially when similarity bounds are small.
BIBLIOGRAPHY


