AUTOMATIC ARIMA TIME SERIES MODELING AND FORECASTING FOR ADAPTIVE INPUT/OUTPUT PREFETCHING

BY

NANCY NGOC TRAN

M.S., University of Illinois at Urbana-Champaign, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Abstract

This thesis presents a comprehensive software framework – Automodeler – to provide automatic modeling and forecasting for input/output (I/O) request interarrival times. In Automodeler, ARIMA models of interarrival times are automatically identified and built during application execution. Model parameters are recursively estimated in real-time for every new request arrival, adapting to changes that are intrinsic or external to the running application. Online forecasts are subsequently generated based on the updated parameters.

Our goal is to improve application-level I/O performance by using forecasts to guide the prefetch of I/O requests before they arrive, reducing stall time. We combine a just-in-time prefetcher with ARIMA time series predictions and Markov model spatial predictions [41] to adaptively determine when, what, and how many data blocks to prefetch.

To validate our approach, we built a prototype that integrates adaptive prefetching with caching and local disk striping in the PPFS2 [51] testbed. Results obtained for a computational physics code demonstrate 30% improvement in total execution time over the traditional Unix file system on three Linux clusters, equipped with different hardware configurations. More importantly, this performance improvement has small memory requirements and is shown to scale with increasing I/O intensity.

Acknowledgments

I would like to express my thanks and gratitude to my advisor, Professor Daniel A. Reed, for his guidance, support and encouragement. Not only is Professor Reed a fantastic visionary researcher, he is also a world-class educator who cares for his students' learning. I still remember his kindness when he took me into his group despite all odds. Professor Reed taught me to conduct research with rigor and set lofty goals. Throughout the years, he has given me full freedom to explore applied statistics, a field I used to fear, but that I now like the most. He provided superb guidance on writing scientific papers – 'one sentence, one idea, one insight'. Indeed, I am extremely fortunate to be his student and am much indebted to him.

Also, many thanks to my thesis committee members, Professors Vikram Adve, Roy Campbell, and David Padua for their time, effort and valuable suggestions.

I am also grateful for the help and support of many current and past members of the Pablo group. Ruth Aydt offered many insights on understanding Autopilot. She is always looking out for the students in the group through her own special way. Celso Mendes made special efforts in helping preparing for the exams despite his busy schedule. Huseyin Simitci introduced me to PPFS2's internals and answered my numerous questions with infinite patience. James Oly's help in integrating his work on Markov models was essential to this thesis. He also showed me the many fine points of the C++ language, amidst the tales of his collection of clay-animation and computer games. Other members of the PPFS2 team, Ryan Fox, John Mainzer, Frederick Rothganger, and Shannon Whitmore provided software and hardware support. Deb Israel diligently proofread this thesis with the eyes of an editor and gave me special advice on logistics in answering exam questions.

Special thanks to Phoebe Lenear and Ying Zhang for their encouragement and friendship, to Thomas Kwan for introducing time series analysis, to Luiz Tavera for explaining various debugging tools, which have helped conserved my sanity. An Lee, Mario Medina, Don Schmidt, and Fredrick

Vraalsen gave me many mini-tutorials on various aspects of Linux.

Barbara Armstrong, Barbara Cicone, Kathryn Herrera, Bonnie Howard, Lori Melchi, and Anda Ohlsson were always graceful and ready to provide much needed administrative assistance.

I owe huge thanks to Ed Seidel, his Cactus team, and John Shalf at NCSA for their valuable technical expertise and support in porting the Cactus code. Newmat09, used in implementing Automodeler, is a matrix package by Robert Davies [16].

Finally, thanks to my parents for their encouragement, support, and special care packages.

This work was supported in part by the National Science Foundation under grants NSF ASC 97-20202, 99-75248, and EIA 99-77284, by the NSF PACI Computational Science Alliance Cooperative Agreement, and by the Department of Energy under contracts DOE B-341492, W-7405-ENG-36, 1-B-333164, and CIT PC-179269.

Table of Contents

Chapter 1 Introduction 1
1.1 Online Spatial and Temporal Modeling 2
1.1.1 ARIMA Time Series 3
1.1.2 Markov Models 4
1.1.3 Online Modeling 4
1.2 Thesis Objectives 5
1.3 Thesis Organization 6

Chapter 2 Related Work 8
2.1 Offline Analysis 8
2.1.1 Static Look-Ahead Schemes 9
2.1.2 Access Pattern Hints – Usage and Synthesis 10
2.1.3 Statistical Modeling and Predictions 11
2.2 Online Analysis 12
2.2.1 Dynamic Look-Ahead Schemes 12
2.2.2 Non-compiler and Compiler Techniques 13
2.2.3 Probabilistic Predictions 14
2.3 Mixed Offline/Online Analysis 15
2.4 Our Proposed Method 15

Chapter 3 Time Series Real-Time Parameter Estimation and Forecasting 17
3.1 ARIMA Time Series Processes 17
3.2 Time Series Analysis Methodology 18
3.3 Time Series Automodeler – an Overview 20
3.4 Real-Time Parameter Estimator 23
3.4.1 ARMA(p,q) Models for Stationary Series 23
3.4.2 Ordinary Least Squares Parameter Estimation for AR(p) Models 25
3.4.3 Recursive Least Squares Parameter Estimation for AR(p) Models 29
3.4.4 Extended Least Squares Estimation for ARMA(p,q) Models 30
3.4.5 Implementation 31
3.5 Recursive Differencer/Integrator for Non-Stationary and Seasonal Series 36
3.5.1 Fundamental Concepts on Differencing 36
3.5.2 General ARIMA Models 39
3.5.3 Cascading Property 40
3.5.4 Recursive Differencer/Integrator (RDI) 41
3.5.5 Computation Cost 44
3.6 N-Step Ahead Predictor 44

3.6.1 One-Step Ahead Forecast for Stationary Arrival Processes . . . 44
3.6.2 N-Step Ahead Forecasts for Stationary Arrival Processes . . . 45
3.6.3 N-Step Ahead Forecasts for Non-Stationary and Seasonal Arrival Processes . 46
3.7 Summary . . . 47

Chapter 4 Automatic Model Structure Identification . . . 48
4.1 Autocorrelation Analysis . . . 49
4.1.1 Autocorrelations . . . 50
4.1.2 Partial Autocorrelations . . . 51
4.1.3 Pattern Characteristics in ACFs and PACFs of Stationary Processes . . . 54
4.1.4 Model Identification Methodology . . . 54
4.2 Automation of Autocorrelation Analysis . . . 58
4.2.1 Identification of Significant Correlations . . . 58
4.2.2 Automatic Detection of Seasons Via ACF . . . 59
4.2.3 Automatic Differentiation of Decay Patterns in ACF and PACF . . . 61
4.2.4 Implementation . . . 63
4.2.5 Identification of Model Structures for Seasonal Series . . . 64
4.3 Summary . . . 69

Chapter 5 Automatic Detection of Abrupt Phase Transitions via Haar Wavelet 70
5.1 Haar Wavelet . . . 70
5.2 Confidence Interval Computations via Jackknife . . . 73
5.2.1 The Jackknife Procedure . . . 73
5.2.2 Pseudo-Jackknife . . . 75
5.3 Detection of Phase Transitions in PRISM . . . 75
5.3.1 PRISM Phase Transition I . . . 76
5.3.2 PRISM Phase Transition II . . . 76
5.4 Detection of Seasonal patterns in ESCAT . . . 79
5.5 Summary . . . 79

Chapter 6 Adaptive Prefetching in PPFS2 . . . 83
6.1 PPFS2 – a File System with Adaptive Resource Control . . . 83
6.1.1 PPFS2 Base Architecture . . . 85
6.1.2 Qualitative Access Pattern Classification . . . 85
6.1.3 Quantitative Access Pattern Prediction via Markov Models . . . 86
6.2 Adaptive Prefetching System . . . 87
6.2.1 Prefetch Schedule Builder . . . 87
6.3 Implementation . . . 92
6.4 Summary . . . 93

Chapter 7 ARIMA Automodeler Experiments . . . 94
7.1 Experiment Environment . . . 95
7.1.1 Hardware Platform . . . 95
7.1.2 Integration with PPFS2 . . . 95
7.2 Modeling I/O traces of Scientific Applications . . . 96
7.2.1 Fluid Dynamics Code: PRISM . . . 96
7.2.2 Electron Scattering Code: ESCAT . . . 103
7.3 Synthetic Workload for Read Request Interarrival Times . . . 110

7.3.1 Read Interarrival Time Pattern . . . 110
7.3.2 Results and Analysis . . . 112
7.3.3 Request Interarrival Time Modeling Overhead . . . 113
7.4 Synthetic Workload for Block Read Interarrival Times . . . 115
7.4.1 Results and Analysis . . . 115
7.5 A Computational Physics Application – Cactus Wavetoy . . . 117
7.5.1 Wavetoy I/O Behavior . . . 118
7.5.2 Experiments on Wavetoy Read Interarrival times . . . 119
7.5.3 Wavetoy Interarrival Time Pattern . . . 119
7.5.4 Automatic Model Structure Identification . . . 120
7.5.5 Results and Analysis . . . 125
7.5.6 Model Structure Simplification . . . 125
7.6 Summary . . . 129

Chapter 8 Prefetching Experiments . . . 131
8.1 Experimental Hardware Platform . . . 132
8.1.1 16-Node Cluster . . . 132
8.1.2 32-Node Cluster . . . 132
8.2 Experiment Parameter Specifications . . . 132
8.3 Performance on the 32-Node Cluster . . . 133
8.3.1 Prefetching on Single Disk with Read-Write Caching . . . 133
8.3.2 Prefetching on Single Disk with Read-Only Caching . . . 136
8.3.3 Prefetching with Read-Only Caching and Disk Striping . . . 137
8.3.4 Cache Residency Measurements . . . 138
8.3.5 Prefetching Using Only Markov Model Predictions . . . 139
8.4 Prefetching Performance on Other Clusters . . . 140
8.5 Summary . . . 142

Chapter 9 Conclusions . . . 147
9.1 Automodeler Design Summary . . . 147
9.2 Adaptive Prefetcher Design Summary . . . 149

Chapter 10 Future Work . . . 151
10.1 Model Identification Enhancement . . . 151
10.2 Write-Behind . . . 152
10.3 Transformations other than Differencing . . . 152
10.4 System Resource Predictions . . . 153

References . . . 155

Vita . . . 162

List of Tables

4.1 Broad Characteristics in ACFs and PACFs of Stationary Processes 54
4.2 An Example of Level-1 Distances 59
4.3 An Example of Level-2 Distances 60

5.1 Haar Wavelet Decomposition into Average and Detail Coefficients 71
5.2 Haar Detail Coefficients after One Differencing Level 72
5.3 Systematic Splitting of Detail Coefficients into SubSamples 74
5.4 Pseudo-Jackknife Computations 75

7.1 Detection of Incrementally Seasonal Behavior in Wavetoy 119

8.1 Wavetoy Read Write Summary (1000 Iterations) 131
8.2 Percentage Improvement in Wavetoy Execution Time – 32-Node Cluster 135
8.3 Cache Miss Comparison between Prefetch and without Prefetch – 32-Node Cluster . 136
8.4 Cache Miss Comparison between Read-Write and Read-Only Caching 137
8.5 Cache Residency Measurements for 2 Cache Sizes 139
8.6 Cache Hits/Misses for 2 Cache Sizes 140
8.7 Percentage Improvement in Wavetoy Execution Time – 16-Node Cluster 144
8.8 Percentage Improvement in Wavetoy Execution Time – 8-Node Cluster 144
8.9 Cache Miss Comparison – 16-Node Cluster 146
8.10 Cache Miss Comparison – 8-Node Cluster 146

List of Figures

1.1 Access Pattern Correlations . . . 3
1.2 Markov Model File Block Example . . . 5

3.1 ARIMA Automodeler Architecture . . . 21
3.2 Outline of the Implemention of ELS . . . 32
3.3 Differencing a Seasonal Series . . . 38
3.4 Two Levels of Differencing . . . 38
3.5 Two Types of Correlations . . . 40
3.6 Recursive Differencer/Integrator Structure . . . 42
3.7 Duality between Differencing and Integration . . . 43

4.1 Phase Transitions . . . 49
4.2 ACF of a Synthetic Series . . . 51
4.3 PACF of a Synthetic Series . . . 53
4.4 PRISM ACF . . . 55
4.5 PRISM ACF after One Regular Differencing . . . 55
4.6 PRISM ACF after Seasonal Differencing . . . 56
4.7 PRISM PACF after Seasonal Differencing . . . 56
4.8 Season Detection Using PRISM ACF . . . 60
4.9 An Example PACF with Outliers . . . 62
4.10 Automatic Model Identification Architecture . . . 65
4.11 PRISM ACF Selected at Season Boundaries . . . 67
4.12 PRISM PACF Selected at Season Boundaries . . . 67
4.13 PRISM ACF within the First Season . . . 68
4.14 PRISM PACF within the First Season . . . 68

5.1 Haar Detail Coefficients for PRISM – Phase Transition I . . . 77
5.2 PRISM Series – Phase Transition I . . . 77
5.3 Haar Detail Coefficients for PRISM – Phase Transition II . . . 78
5.4 PRISM Series – Phase Transition II . . . 78
5.5 Automatic Model Identification Architecture Extended with Wavelet Transform . . . 80
5.6 ESCAT Detail Coefficients from the Haar Wavelet Transform . . . 81
5.7 ESCAT Series - Sharp Edges at Season Boundaries . . . 81
5.8 ESCAT Detail Coefficients with Upper Confidence Limit . . . 82
5.9 ESCAT Detail Coefficients Viewed at Close Range . . . 82

6.1 PPFS2 Architecture . . . 84
6.2 Adaptive Prefetcher Structure and Interacting Components . . . 88

6.3 Detection of a Constrained Disk . . . 90
6.4 Unconstrained Disk . . . 91
6.5 Prefetcher's Implementation . . . 92

7.1 PRISM ACF (after Seasonal Differencing) at Season Boundaries . . . 97
7.2 PRISM PACF (after Seasonal Differencing) at Season Boundaries . . . 97
7.3 PRISM ACF (after Seasonal Differencing) within the First Season . . . 98
7.4 PRISM PACF (after Seasonal Differencing) within the First Season . . . 98
7.5 PRISM's General Distribution of Interarrival Time Predictions . . . 100
7.6 PRISM's Interarrival Time Predictions after Phase Transition I . . . 101
7.7 Detailed Snapshot of PRISM's Interarrival Time Predictions . . . 101
7.8 PRISM's Interarrival Time Predictions after Phase Transition II . . . 102
7.9 PRISM's Prediction Error Ratios . . . 102
7.10 ESCAT ACF after One Regular Differencing . . . 104
7.11 ESCAT ACF (after Seasonal Differencing) at Season Boundaries . . . 105
7.12 ESCAT PACF (after Seasonal Differencing) at Season Boundaries . . . 105
7.13 ESCAT ACF (after Seasonal Differencing) within the First Season . . . 107
7.14 ESCAT General Distribution of Interarrival Time Predictions . . . 107
7.15 ESCAT Interarrival Time Predictions for the First 6 Seasons . . . 108
7.16 ESCAT Interarrival Time Predictions for Seasons 7 to 10 . . . 108
7.17 ESCAT Prediction Error Ratios . . . 109
7.18 Synthetic Workload – General Distribution of Interarrival Time Predictions . . . 110
7.19 Synthetic Workload – Predictions for the First Four Seasons . . . 111
7.20 Synthetic Workload – Predictions Viewed at Close Range . . . 111
7.21 Synthetic Workload – Prediction Error Ratios . . . 112
7.22 ARIMA Parameter Estimation Overhead . . . 114
7.23 Synthetic Workload – General Distribution of Block Interarrival Time Predictions . . . 116
7.24 Synthetic Workload – Predictions for the First 100 blocks . . . 116
7.25 Synthetic Workload – Block Prediction Error Ratios . . . 117
7.26 Wavetoy Detail Coefficients . . . 121
7.27 Wavetoy Detail Coefficients Viewed at Close Range . . . 121
7.28 Wavetoy ACF of the Major Series . . . 122
7.29 Wavetoy PACF of the Major Series . . . 122
7.30 Wavetoy ACF of the Minor Series - after Regular Differencing . . . 123
7.31 Wavetoy ACF of the Minor Series - after Seasonal Differencing . . . 123
7.32 Wavetoy ACF of the First Season in the Minor Series (after Seasonal Differencing) . . . 124
7.33 Wavetoy PACF of the First Season in the Minor Series (after Seasonal Differencing) 124
7.34 Wavetoy General Distribution of Interarrival Time Predictions . . . 126
7.35 Wavetoy Interarrival Time Predictions for the First 300 blocks . . . 126
7.36 Existence of Several Minor Seasons within a Major Season . . . 127
7.37 Wavetoy's Prediction Error Ratios . . . 127
7.38 Wavetoy Predictions Using a Simple Model . . . 128

8.1 Wavetoy Execution Time under Different Configurations – 32-Node Cluster . . . 134
8.2 Wavetoy Execution Time Improvement – 32-Node Cluster . . . 134
8.3 Wavetoy Execution Time Using Only Markov Predictions . . . 141
8.4 Synthetic Workload Execution Time Using Only Markov Predictions . . . 141
8.5 Wavetoy Execution Time under Different Configurations – 16-Node Cluster . . . 143

8.6 Wavetoy Execution Time Improvement – 16-Node Cluster 143
8.7 Wavetoy Execution Time under Different Configurations – 8-Node Cluster 145
8.8 Wavetoy Execution Time Improvement – 8-Node Cluster 145

Chapter 1

Introduction

Today's computers have processors and disks located at opposing ends of a deep memory-storage hierarchy, operating at speeds differing by several orders of magnitude [43]. As processor performance continues to increase faster than disk performance (60% annual increase in compute speedup versus 8% annual reduction in disk seek and rotational latencies), the I/O bottleneck widens [48]. Hindered by slow mechanical arm movements in the magnetic storage technology, disk access time improvement is likely to remain modest in the near future. Given more disks and more processors, applications still suffer from significant processor stall times, especially those with growing demands for petabytes of data [18]. These applications cannot obtain sufficient disk transfer bandwidth to fully benefit from increasing processor performance and disk storage capacities.

Many caching and prefetching systems [31, 44, 9] seek to ameliorate the I/O bottleneck by overlapping computation with disk read accesses, masking I/O latency from applications. Prefetching for reads and write-behind for writes [32] are standard techniques. For prefetching, such techniques must strike a judicious balance between application-initiated demand fetches and prefetches. Late prefetching can make demand fetches stall for disk I/O completion. Premature prefetching can flood the file cache, evicting cache blocks that would otherwise satisfy future demand fetches. Similar constraints apply for write-behind strategies, caching write requests during phases of high I/O activity and flushing dirty cache blocks when the application becomes compute bound.

Providing timing estimates for prefetching and write-behind in the presence of time-varying and non-uniform file accesses is challenging. Fluctuations in processor loads and operating system queuing delays, together with application periodic checkpoints and nested loop structures, make I/O arrival patterns bursty. Such bursty patterns overflow system buffers and network links, leading

to extended I/O stalls. One effective approach to reducing I/O stalls for reads is to anticipate future arrival bursts, prefetching data before it is needed.

Analyses of parallel scientific codes have shown that both temporal and spatial access patterns are far more complex than the regular, sequential patterns long expected [13, 53]. Not only are application I/O patterns bursty, their spatial patterns are often irregular, due to mediation by multi-level I/O libraries.

Given the complexity and diversity of I/O patterns, we believe anticipation strategies are best realized via intelligent file systems that can adaptively model and track I/O access patterns in the spatial and temporal domains. Access pattern forecasts can then be used to make prefetch decisions during application execution.

In this thesis, we explore adaptive techniques to maximize overlapping of computation with disk read accesses. In particular, we investigate the use of time series analysis to predict temporal I/O access patterns, Markov models to predict spatial access patterns [41], and latency hiding by prefetching read requests.

1.1 Online Spatial and Temporal Modeling

In many scientific applications, I/O request interarrival times are often correlated as a consequence of program structure (e.g., periodic checkpoints or loop-mediated I/O). The result is a set of distinctive patterns amenable to statistical representation.

As an example, Figure 1.1 shows a series of read interarrival times extracted from an I/O trace of ESCAT, a Schwinger-Multichannel Electron Scattering code, via our Pablo instrumentation toolkit [46]. The series reveals periodically bursty I/O patterns, typical of scientific applications mediated by loops. Such patterns repeat at regular intervals — bursts of read arrivals within a period alternate with long computation intervals. These intervals[1], hovering around 250 milliseconds, are larger than the bursts (less than 3500 microseconds) by two orders of magnitude. The patterns also fluctuate non-deterministically around a non-stationary mean, rising linearly with an increasing trend, showing positive correlations (linear dependencies). It is precisely these correlations that time series analysis exploits to statistically model the underlying process.

[1] In Figure 1.1, the intervals are truncated at 4000 microseconds for presentation clarity.

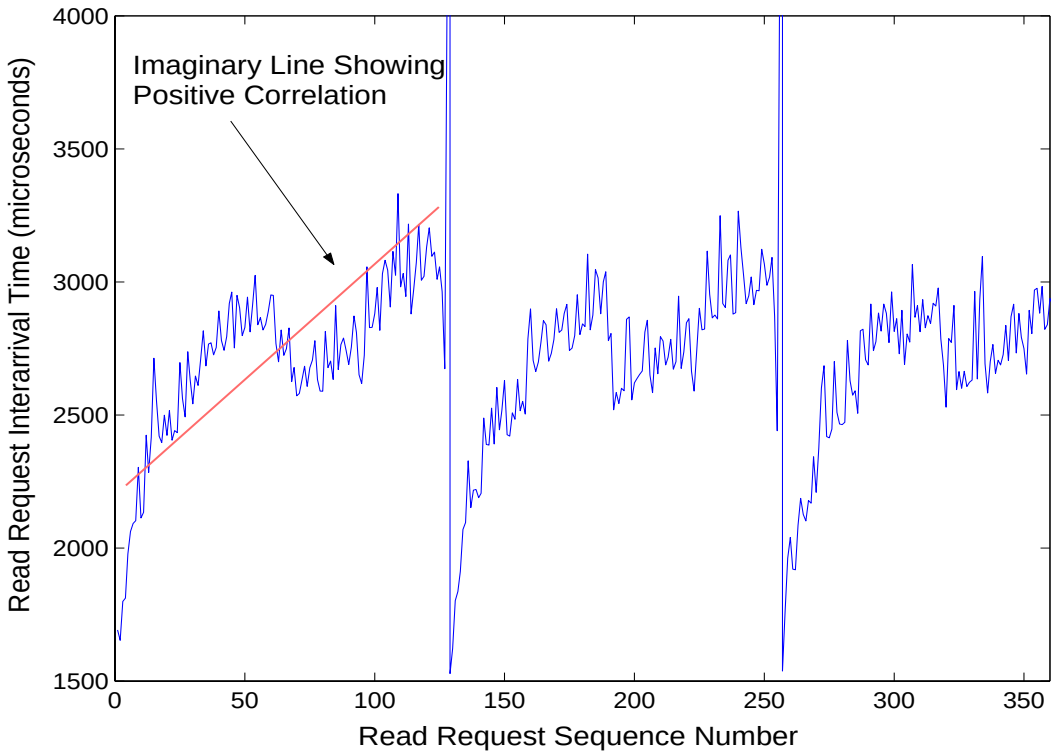

Figure 1.1: Access Pattern Correlations

Capturing such a complex I/O pattern requires both spatial and temporal models. In the temporal domain, we used future request interarrival times to guide when to prefetch for reads. In the spatial domain, we used predictions for future file block accesses to guide what to prefetch. ARIMA time series and Markov models are two attractive approaches to determining when, what, and how much data to prefetch. Below we outline the rationale for these choices and later describe their use for prefetching.

1.1.1 ARIMA Time Series

There are many possible techniques for capturing temporal behavior. Box-Jenkins's ARIMA (Autoregressive Integrated Moving Average) time series analysis is attractive [7] because it provides a comprehensive statistical modeling methodology for I/O processes. It covers a wide variety of patterns, ranging from stationary, to non-stationary, and seasonal (periodic). For example, the ESCAT pattern in Figure 1.1 has both non-stationary (increasing) and seasonal behavior, with periodically similar patterns repeating every 128 read requests. ARIMA systematically analyzes

the correlations among observed data to derive models through a sequence of stages — pattern identification suggesting a tentative model, model fitting to compute model parameters from observed data, and finally diagnostic checking to detect model inadequacy. The final model consists of components that indicate the areas of significant correlations, whereas the model parameters capture the magnitudes of these correlations.

Subsequently, the fitted model is used to produce forecasts for future read interarrival times. The probability limits of these forecasts, computed using the variance of the forecast errors, provide a confidence basis for making prefetch decisions. For example, prefetches are initiated only for those forecasts that fall within the 90% confidence level.

1.1.2 Markov Models

ARIMA models can describe temporal patterns but provide little insight on spatial access patterns. Both are needed to develop effective I/O latency hiding policies. Markov models (MMs) are one effective technique for representing spatial access patterns. MMs can describe complex I/O patterns via compact state-transition models. Typically, the states represent fixed-size file blocks, with the probabilities on edges representing the likelihood that the associated file blocks are accessed. Based on the most recently accessed block and the model's transition probabilities, one can predict future block accesses. Multi-step ahead predictions made from MMs provide a basis for multiple block prefetching.

As an example, Figure 1.2 illustrates an irregular access pattern typical of what might be generated by an I/O library on behalf of an application. First, metadata is accessed at the beginning of the file, then one of two possible sets of file blocks are accessed.

1.1.3 Online Modeling

As we noted earlier, the spatial and temporal access patterns in scientific applications are time-varying as a result of changes that can arise from either intrinsic (checkpoints, loops) or external (processor loads, communication system delays) sources. To be effective, prefetching designs need to consider these changing conditions — tracking and adapting to the changes. As these changes can vary widely across executions, predictions from traditional offline modeling schemes can become

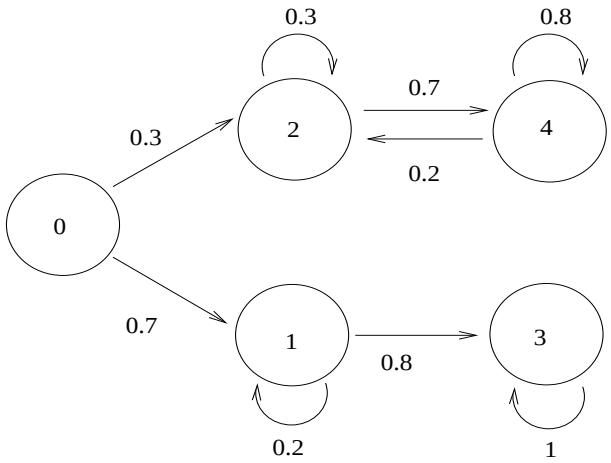

Figure 1.2: Markov Model File Block Example

obsolete in the presence of non-reproducible program behavior and execution environments.

As an example, consider the scenario where an emergency, unexpected batch job is inserted into the system during the ESCAT application execution. As this job shares processor and memory resources with ESCAT, ESCAT's read interarrival times increase. If the model parameters were statically estimated from observed data of previous executions (when no additional job was sharing the system), forecasts are no longer accurate because they do not capture the actual I/O behavior.

Consequently, one must adapt to changes in the underlying system dynamics. One approach to achieve such adaptation is modeling the read interarrival times during program execution, updating associated time series model parameters and Markov model state transitions dynamically. Such online predictions, made as a result of online modeling, are usually more efficient than those obtained from static techniques, especially in the presence of non-reproducible program behavior and execution environments.

1.2 Thesis Objectives

To achieve real-time modeling for temporal access patterns, we build a software framework capable of creating statistical models that can dynamically approximate and track application I/O patterns, according to observed data. Behavioral changes are captured through automatic re-computations of model parameters or even reconstruction of models to adaptively produce forecasts of future patterns and trends. Below, we outline our main objectives, focusing on prefetching for read

accesses. We believe our approach can equally be applied to improve the performance of write-behind strategies.

1. Building a *time series modeling engine – Automodeler*, capable of *automatically* identifying, tracking, estimating, and predicting in real time the stochastic and bursty behavior of application temporal I/O access patterns in the near future.

2. Building an *adaptive prefetcher*, capable of determining at run time when, what, and how many data blocks to prefetch. Prefetch decisions are made online, according to Automodeler's time series predictions and Oly's Markov model file block predictions [41].

3. *Experimentally validating* the modeling engine and the prefetcher by conducting tests on scientific applications. Emphasis is placed on:

   - Assessing Automodeler's ability to model I/O temporal access patterns. Specifically, the modeling engine's prediction accuracy and processing overhead are evaluated.
   - Assessing the prefetcher's effectiveness in improving read performance for local disk I/O (single and multiple disks) on PC clusters.

1.3 Thesis Organization

The remaining chapters are organized as follows. In Chapter §2, we survey the literature related to predictive prefetch methods and online/offline modeling approaches.

In Chapters §3, §4, and §5, we present the design and implementation of Automodeler, a framework for automatic, online time series modeling. Chapter §3 investigates the use of recursive and dynamic algorithms to estimate model parameters for stationary, non-stationary, and seasonal I/O behavior. Chapter §4 explores pattern recognition techniques to automatically identify a model structure. In Chapter §5, we enhance the identification process using wavelet transforms to detect program phase transitions.

In Chapter §6, we describe the architecture of our adaptive prefetcher and provide an overview of PPFS2, our experimental testbed for file system research. We present a variant of the Forestall

algorithm [60], combining spatial and temporal predictions to adaptively decide how much I/O data to prefetch at a given instant.

Experimental results and their analyses are provided in Chapters §7 and §8. Chapter §7 presents validation results – prediction accuracy and modeling overhead – of Automodeler for four sets of experiments: two I/O traces, one synthetic benchmark, and one gravitational physics application. Chapter §8 documents the improvement in the performance of the physics application when spatial and temporal modeling, caching, and prefetching were deployed concurrently. We also evaluate the impact of different caching strategies, software striping, and various hardware platforms on prefetching performance.

Finally, conclusions summarizing our contributions to relieve the I/O bottleneck via real-time modeling and forecasting of I/O behavior are given in §9, followed by discussions on possible future directions in §10.

Chapter 2

Related Work

This thesis explores automatic and adaptive techniques to shape and control prefetch strategies. As prefetching necessitates the knowledge of future accesses, predictions and prefetching must be jointly considered. In this chapter, we survey the literature related to predictive software/hardware prefetch methods and compare extant techniques with our approach.

Several prefetching techniques have been proposed to improve the performance of a wide array of system components, encompassing compilers, memory, I/O, and network communication subsystems. They range from simple sequential read-aheads for virtual storage systems [4] to prefetches of more complex data patterns for parallel systems [32, 37, 61] and wide area networks [22, 28]. The techniques exploit either application-specific information, supplied in the form of hints, or past reference patterns to infer the future. They can be broadly divided into three categories: offline, online, and mixed offline/online analyses. We will examine each category in sections §2.1, §2.2, and §2.3 respectively.

2.1 Offline Analysis

One can detect regular file access behavior via static inspection of program codes, traces, or profiles gathered from prior executions. Consider loop structures, commonly found in many scientific codes to manipulate arrays. These loops, well-known for having predictable disk referencing patterns, provide excellent opportunities for prefetching.

Based on the detected loop patterns, prefetch directives (e.g., the fetch instructions) can be explicitly issued to precede the array element references. These directives must be placed at judi-

cious locations in the programs to avoid late or premature prefetches. They can be hand-coded or automatically inserted at compile optimization time to schedule the prefetches.

Alternatively, when performance costs, code size, and simplicity are important design choices, one can forgo compile-time information and speculatively prefetch based on previous pattern information alone, from profiles or traces. This provides the basis for sequential prefetching, the efficacy of which depends on the access patterns, the cache size, the cache block size, and the prefetch block size [55, 64]. Large block sizes can pollute the cache, especially when speculations are incorrect, causing cache misses that would not have occurred without prefetching.

Below we summarize some representative compiler and non-compiler assisted techniques for prefetching.

2.1.1 Static Look-Ahead Schemes

One block look-ahead (OBL), probably the simplest prefetch policy, has long been used to improve the performance of sequential workloads [54, 55]. There are several variations to OBL; all optimistically assume that the next block to be accessed is next to the current block, or one located at a fixed distance S (stride-based prefetching). For applications with small loop-execution times or high data-request rates, Jouppi extended OBL to optimistically prefetch multiple blocks consecutively [30]. The prefetch depth (number of prefetched blocks) stayed constant throughout the application execution. Jouppi recognized the need for deeper prefetching because there is no timing information maintained in static look-ahead schemes.

The optimistic approaches above are well suited for sequential or regularly strided accesses, exploiting spatial data locality. Example applications, exhibiting strong data locality, include contiguous accesses to arrays in many scientific codes, as well as in image processing FFT (fast Fourier transform) computations and in multimedia sequential movie plays. Look-ahead schemes are relatively simple to implement in hardware. Hardware prefetching, without compiler intervention, has the advantage of reducing overhead introduced by software fetch instructions at the expense of additional hardware units. On the other hand, software prefetching, using compiler static analysis of past references, generally yields more accurate predictions.

However, for workloads with more complex, non-sequential access patterns and poor data lo-

cality, more selective methods are needed.

2.1.2 Access Pattern Hints – Usage and Synthesis

When developers have extensive knowledge of application I/O characteristics, they can manually modify the applications to insert access pattern hints, exposing future data needs. Hints can be generated from detailed examination of traces or program profiles to determine data/instruction access patterns (e.g., which instructions cause the most useful prefetches, which blocks are the most referenced, what are the most common stride values [69]). For I/O, this application-controlled hinting approach was most notably used by Cao to advise cache management policies [9] and by Patterson in the TIP system [44] to make prefetching decisions.

TIP (Transparent Informed Prefetching) exploits hints about future file blocks to determine what to prefetch. It uses a cost-benefit analysis to evaluate the reduction in I/O service time per buffer access for deciding when to allocate cache buffers. Prefetching a hinted request is initiated only when it could reduce I/O service time. Benchmark results showed more than 80% reduction in I/O stall time in the presence of a large disk array. However, when applications are large and complex, especially when I/O is embedded in multi-level libraries, detailed knowledge of application file access behavior may not be possible.

To automate the disclosure of hints, compiler techniques were explored by Mowry to statically examine read requests for scientific workloads and derive hints for future requests [40]. The hints consist of those highly reused data references that are most likely to be cache misses. Based on the hints, code is inserted to prefetch loop-mediated array accesses. With this compiler-directed insertion technique, Mowry achieved significant performance improvements for several scientific application benchmarks. However, this static technique fails to predict file accesses that can change across executions, depending on the values of the input/output data (i.e., data-dependent accesses).

Alternatively, one can automate the synthesis of hints using non-compiler approaches. Usually, these approaches analyze I/O traces to derive predictions for future accesses. For example, in [22, 23], Griffioen and Appleton built probability graphs based on file access frequencies in the traces. These graphs record the likelihood of various access paths (set of graph edges) to allow computation of the most probable sequence of visited files. However, this graph-based analysis is

not scalable as the numbers of nodes in the graphs explode. The costs of path discovery, graph modification, or reconstruction can be prohibitively expensive.

Many groups have explored other statistical techniques to address the challenges of complex access pattern recognition, modeling, and prediction. Below, we review some representative statistical modelers and predictors, reported to have produced accurate predictions.

2.1.3 Statistical Modeling and Predictions

Data patterns with non-deterministic behaviors, e.g. those observed in I/O accesses of scientific applications, have long been investigated by many researchers using applied statistics. Statistics provide mathematical tools to reason about uncertainties and predict favorable outcomes accordingly. The choice of outcomes can be made based on some desired criteria. Example criterion includes the most likely access path given the current access, or the set of future interarrival times that fall within the 90% confidence interval.

Joseph and Grunwald used a Markov predictor to implement a hardware prefetcher for memory accesses [29]. Only the addresses of those memory references that missed the cache were examined instead of all references, substantially reducing the total data volume analyzed. The Markov models were trained offline using the transition probabilities of miss reference addresses collected from traces. The Markov transition diagrams, represented via tables, were limited to a maximum of four transitions from each address. This design choice traded accuracy for overhead containment. Conflict with the L1-cache was resolved by assuming separate, on-chip prefetch buffers.

For a given a matched address, Joseph and Grunwald decided to prefetch all the associated transitions recorded in the prefetch table. The highest priority was given to the largest transition probability (i.e., the most likely transition). This technique introduced many opportunities for incorrect prefetches. The authors reported that a more effective approach was to give highest prefetching priority to the most recently used transition, instead of the most likely transition. Results of simulation studies indicated a 54% average reduction in memory CPI.

Markov models can describe spatial access patterns but are not amenable to efficient analysis of temporal access patterns. Other techniques are needed. Inspired by the works of Leland, Paxson, Willinger et. al. [35, 45, 65] on modeling self-similar arrival patterns in network traffic, Gomez

and Santonja analyzed arrival patterns in disk traces [21]. Three different statistical techniques (the rescaled adjusted range statistic, the variance-time plot, and the periodogram) were used to estimate and cross-validate the Hurst parameter, a self-similar pattern indicator. Large Hurst parameters were found in periods of intense I/O activity. They concluded that the arrival patterns are self-similar (fractal-like behavior), bursty over a wide range of time scales, and characterized by heavy-tailed distributions (having infinite variance).

Groschwitz and Polizos reported another method to forecast NSFNET backbone traffic [24]. They explored ARIMA time series analysis to model the daily volume of packets exchanged on the NSFNET between 1988 and 1993. Using a non-stationary and seasonal ARIMA model with only a few parameters, they successfully produced accurate forecasts on a scale of months up to one year. Given the explosive growth rate of network traffic as a result of the success of the Internet, their method provides a planning tool for future network requirements.

Whereas offline analysis achieves significant performance gains in many applications, its effectiveness may be diminished when used in heterogeneous environments with changing system conditions. Predictions derived from traces of one execution cannot be used across executions when applications, perturbed by external factors, produce very different reference patterns. For network traffic, with the emergence of new Internet, multimedia, and sensor related applications, traffic patterns are likely to change over the course of days and months. These observations are supported by the findings and conclusions in Joseph and Groschwitz's studies [29, 24]. For time-varying systems, online analysis utilizing intelligent techniques to dynamically model data accesses, seems to offer more advantages.

2.2 Online Analysis

Similar to offline analysis, online analysis has evolved from simple look-ahead schemes for sequential patterns to more sophisticated techniques for complex patterns.

2.2.1 Dynamic Look-Ahead Schemes

Kotz successfully predicted several sequential, regular access patterns found in scientific workloads [33]. Two major types of patterns were investigated: locally sequential accesses from uniprocessors

and globally sequential accesses from multiprocessors that share a file portion, all reading disjoint blocks from the portion. Heuristics were used 1) to identify and predict sequential patterns, and 2) to recognize the lengths and skips of the file portions. All were based on file references monitored at run time. Kotz's predictors anticipated regular file portion accesses, n blocks look-ahead for sequential accesses, but only one block look-ahead when the first, non-sequential access was encountered. Using a simulated environment for parallel independent disks and a round-robin file striping policy across all disks, he reported more than 50% performance improvement for a wide variety of workloads.

Whereas sequential prefetching exploits spatial locality in consecutive blocks, stride prefetching exploits the regularity of non-sequential accesses, separated by a constant interval. Such strided patterns occur frequently in arrays and matrices mediated by loop indexes.

Regular strides can be dynamically detected by examining the addresses of the load/store instructions and associating them with the addresses of the current and previous memory accesses [12, 20]. Alternatively, they can be isolated by analyzing the read/write accesses [25]. The most recently used addresses are cached in a reference prediction table to compute the strides and predict future strided sequences. The depth of these sequences are adaptively determined by approximating the spatial locality in observed memory references [15].

Dahlgren and Stenstrom evaluated the performance of hardware-based sequential and stride prefetching in shared-memory processors [14]. They concluded that 1) if the strides are smaller than the block sizes, sequential prefetching could be just as effective as stride prefetching, and 2) stride prefetching may result in fewer incorrect prefetches, hence less memory bandwidth consumption, at the cost of extra hardware support.

2.2.2 Non-compiler and Compiler Techniques

To detect patterns that are more complex than just sequential, Madhyastha deployed a feed-forward artificial neural network to automatically identify and qualitatively classify I/O access patterns [37]. Classification results provide qualitative information to describe different types of file requests (read or write), their access orders (sequential, strided or random), and their sizes (uniform, small, medium, large). By using classification knowledge to select file policies, significant performance

gain was attained for several experiments with scientific workloads. This qualitative approach can be supplemented by more detailed, quantitative predictions to cover a broader range of access patterns.

Chang et al. presented a non-probabilistic, compiler-assisted technique to produce online predictions for I/O [10]. Applications are speculatively pre-executed to reveal upcoming requests while stalled pending the completion of disk reads. Exposed read accesses are then issued as hints for future accesses to the TIP [44] prefetching and caching manager to guide prefetch decisions. This technique allows irregular access patterns to be dynamically detected. It also avoids the need for source code modifications. However, optimistic pre-execution, based on incomplete state information, may be erroneous when file request patterns are data dependent, compromising prediction accuracy.

2.2.3 Probabilistic Predictions

Other groups have explored probabilistic methods to generate quantitative predictions during program execution. Vitter applied the Ziv-Lempel data compression technique to create a prefetch algorithm that dynamically builds probabilistic models for access sequences [61]. Predictions are computed based on data accesses having the highest reference probabilities. Tested with accesses created by a Markov finite state automaton, the prefetch algorithm demonstrated significant performance improvement. However, its ability to recognize and prefetch non-sequential accesses, produced by non-Markov sources, remains to be investigated.

Jiang studied another probabilistic, adaptive prediction scheme for prefetching files on the World Wide Web [28]. Analytical queuing models are used to compute the probabilities of whole file accesses across the network. Prefetches are initiated only for accesses exceeding a preset probability threshold. This threshold is adaptively computed based on four major factors: Poisson arrival rates of requested files, transmission times, network bandwidth needed, and system utilization. Results through simulation showed 70 to 80% accuracy in predicting future file accesses. However, the assumption of Poisson arrivals may not be valid when applied to networks (LAN or wide area), where file request arrivals are usually not exponentially distributed [35, 45].

2.3 Mixed Offline/Online Analysis

Alternatively, one can combine the advantages offered by both static and dynamic analyses to design better predictors.

Dinda explored the use of time series analysis to predict processor loads [17]. Based on the belief that non-I/O workloads generally do not have ARIMA seasonal patterns, his study considers only stationary and non-stationary models. His methodology includes three steps: (a) manual identification of the model structure, (b) offline model parameter estimation using samples of past load measurements, and (c) online predictions for future CPU loads based on previously estimated parameters. Estimation costs range from 100 milliseconds for simple models to over 70 seconds for more complex models. Although Dinda's method was shown to work well for processor loads, it is not amenable to a large class of I/O workloads, which frequently have bursty and seasonal behaviors.

Recently, Oly investigated probabilistic Markov models to predict future, spatial I/O access patterns [41]. The models can be trained online or offline with transition probabilities, recording the frequencies of logical file block accesses. Predictions are made dynamically based on the current block access and the model's transition probabilities of all past references. To accommodate different classes of access patterns, three prediction schemes were devised. Modeling experiments, performed on I/O traces for a suite of scientific applications, achieved prediction accuracy between 70% to 90% for most traces.

2.4 Our Proposed Method

Our approach to quantitatively predicting temporal I/O access patterns, using time series analysis, differs from currently known methods in three major aspects:

- To accommodate the variability in I/O temporal behavior, we emphasize 1) *automatic, online identification* of model structure, and 2) *recursive, online parameter estimation* techniques to adapt to changes in I/O request interarrival times. Online identification exploits pattern recognition techniques to automaticaly select the most significant correlations. Unlike offline estimation, online estimation exploits recursion to reduce processing overhead by avoiding

refitting the entire series when new observations are added.

- To cover a broader spectrum of temporal patterns, we modeled time series in its *general form*, encompassing stationary, non-stationary, and seasonal processes. Based on findings that access patterns in scientific applications are often bursty and seasonal [38, 53], modeling this behavior is crucial to achieving a higher fraction of the theoretical performance peak.

  Specifically, we have expanded the recursive parameter estimation algorithm for modeling stationary processes [2, 36, 11] to include a recursive differencing/integration algorithm for the large class of non-stationary and seasonal processes. The correctness of our algorithm was validated experimentally via real workloads.

- We exploited quantitative predictions of I/O arrival patterns to allow file systems to plan prefetching based on *anticipated, rather than observed need* — late prefetching misses the window of opportunity for overlapping computation with disk I/O. The time series predictions are combined with the Markov model spatial predictions to achieve just-in-time prefetching. We believe that our approach can provide a more inclusive and timely treatment to the I/O prefetching problem.

Chapter 3

Time Series Real-Time Parameter Estimation and Forecasting

Accurate forecasts of temporal access patterns are essential in building an effective prefetch system, free of late or premature prefetches. However, extant offline forecasting techniques have difficulty reflecting changes across executions. These changes may be data-dependent or resource-dependent. Example applications include those with bursty I/O arrival patterns and those executing in a time-shared environment. For such applications, obsolete offline forecasts might weaken the effectiveness of a prefetch system. Real-time forecasts, on the other hand, can adapt to changes, providing more accurate projections for future arrivals to improve prefetch performance.

To generate real-time forecasts, this chapter investigates online time series modeling techniques that can capture underlying system dynamics through updates of model parameters in real-time. We motivate time series analysis and review ARIMA modeling methodology in §3.1 and §3.2. In §3.3, we present the architecture of an online modeling framework. Three architectural components: model parameter estimation for stationary processes, differencing/integration transformations for non-stationary and seasonal processes, and real-time forecasting, are described in §3.4, §3.5, and §3.6 respectively.

3.1 ARIMA Time Series Processes

Time series analysis uses statistical methods to analyze dependencies in a chronological sequence of observations. When applied to I/O, the observations can be any quantitative data gathered from an I/O process, such as request interarrival times. The objective is to characterize the behavior of the

underlying process and predict future behavior, based on the assumption that the future will tend to echo the past. This characterization is achieved by exploiting correlations among observations to identify a structure for the I/O process. Box-Jenkins's ARIMA time series analysis distinguishes three major types of processes: stationary, non-stationary, and seasonal [7].

Intuitively, in a *stationary series*, the underlying process remains in statistical equilibrium, with observations fluctuating around a constant mean with constant variance. In contrast, a *non-stationary* series has no natural mean, but tends to increase or decrease over time. For instance, within each period of 128 observations, the ESCAT series (§1.1) drifts upward and oscillates along a rising line, exhibiting an overall increasing trend.

The ESCAT series also displays *seasonal* characteristics, with distinctive patterns repeating at regular intervals. The length of each interval, the season length, is 128 request arrivals. The series is marked by increasing, non-stationary behavior, recurrent in every season throughout the application. Within a season, the read interarrival times are short, in the order of 3500 microseconds. However, at the boundary of each season is a long interarrival time of more than 20000 microseconds, corresponding to a round of long computations. Similar seasonal I/O patterns are commonly found in loop-mediated applications, where long computations within a loop alternate with bursts of file requests between loops.

3.2 Time Series Analysis Methodology

ARIMA time series analysis provides mathematical models to describe the stationary, non-stationary, and seasonal structures. The classical approach to building and fitting an ARIMA model to a times series includes three basic steps: model identification, parameter estimation, and diagnostic checking. Forecasting is an additional step following the modeling process to use the modeling results.

Model identification refers to determining the model structure, the number of past observations and past disturbances that are strongly correlated with the current observation. Autocorrelation functions, used as the primary tools to expose the dependencies, give recognizable patterns when plotted. Statistical packages such as SAS, Splus and Matlab provide tools to compute and plot these functions. One common approach to estimating a model structure is to examine the number of statistically significant spikes and their patterns in the plots.

Parameter estimation — a process of fitting a model to observed data — involves estimating the magnitudes of correlations found in observed data. These magnitudes are expressed as coefficients of the model parameters, given an identified structure. Several estimation methods are available: maximum likelihood, least squares, Marquardt [42], etc. Due to its simplicity and efficiency, the least squares method is widely used to fit *stationary* series. It seeks to minimize the sum of the squared fitted errors, i.e., the squares of the differences between estimated and actual values.

ARIMA non-stationary and seasonal series require transformations to become stationary before least squares parameter estimations can be applied. These transformations involve taking the differences between observations, based on the assumption that different parts of the series behave similarly, except for the differences in their means [7]. If this assumption is not valid, other types of transformations (e.g., logarithmic or power transformations) can be applied [6, 39]. Experiments on a suite of I/O traces, gathered using the Pablo instrumentation toolkit [46], confirm the validity of the differencing assumption for modeling I/O interarrival time series.

Diagnostic checking provides statistical tests to validate the adequacy of a model structure. Most tests are predicated on the analysis of residuals, the differences between observed and model fitted values. Discernible, non-random patterns remaining in the residuals are strong indications of lack of fit: the model structure has not captured all the significant correlations. When this situation occurs, the correlation functions of the residuals are examined, and the model is iteratively re-identified and refitted until no lack of fit is found.

Finally, estimated parameters and the adequate model structure are used to *forecast* future values for the series, extrapolating past data. One of the major objectives in time series analysis is to use predicted outcomes to guide policy decisions. Because decision-making applications can vary extensively in their time horizon requirements, ARIMA allows short-term and long-term forecasts, ranging from one to an arbitrary number of steps ahead.

Traditionally, the four steps described above are executed *offline* using past observations stored as time series. One major disadvantage of offline approaches is their inability to dynamically capture changes in underlying processes. Offline forecasts are computed based on parameters estimated using historical rather than real-time data. They are most likely to be invalidated by changes in system conditions. Processor upgrades, network equipment replacements, software upgrades,

or even the introduction/removal of a single job into/from the system can make offline forecasts obsolete. For I/O, obsolete forecasts can cause incorrect and untimely prefetches.

As a result, model parameters must be estimated in real-time and adaptively modified to track and reflect the evolution of I/O process dynamics. We need a *systematic, online, and adaptive* time series modeling framework, sufficiently *general* to accommodate all three types of ARIMA processes such that broad coverage of I/O access patterns can be predicted dynamically.

3.3 Time Series Automodeler – an Overview

Our proposed Automodeler is an online modeling framework, designed to provide automatic model identification, online tracking, estimation, and prediction of ARIMA time series in its *general form*. We use wavelet transforms and autocorrelation function analysis to automatically identify an ARIMA model structure. To realize real-time parameter estimation and prediction, we supplement the existing extended least squares (ELS) algorithm [2, 36] with a new recursive differencing/integration (RDI) algorithm. ELS is used to model stationary series, whereas RDI models non-stationary and seasonal series.

The architecture of Automodeler, illustrated in Figure 3.1, consists of four main components:

- A *model structure identifier* to approximate a structure for the correlations that exist in data observed at run time. We explored techniques to automatically recognize patterns in the autocorrelation functions and the wavelet transforms. These techniques isolate significant dependencies automatically, removing the need for manual intervention and visual inspection of plots.

  Several statistical tests on the residuals (fitted errors) are available to determine the adequacy of a model. Most notable are the Chi-square goodness of fit test for validating the normal (Gaussian) distribution assumption, the residual plot for verifying the constant variance assumption, and the portmanteau lack of fit test for analyzing all the residuals together [63].

  However, when applied to modeling I/O behavior online, the model adequacy tests incur large computational costs, increasing total application execution time. To minimize the iden-

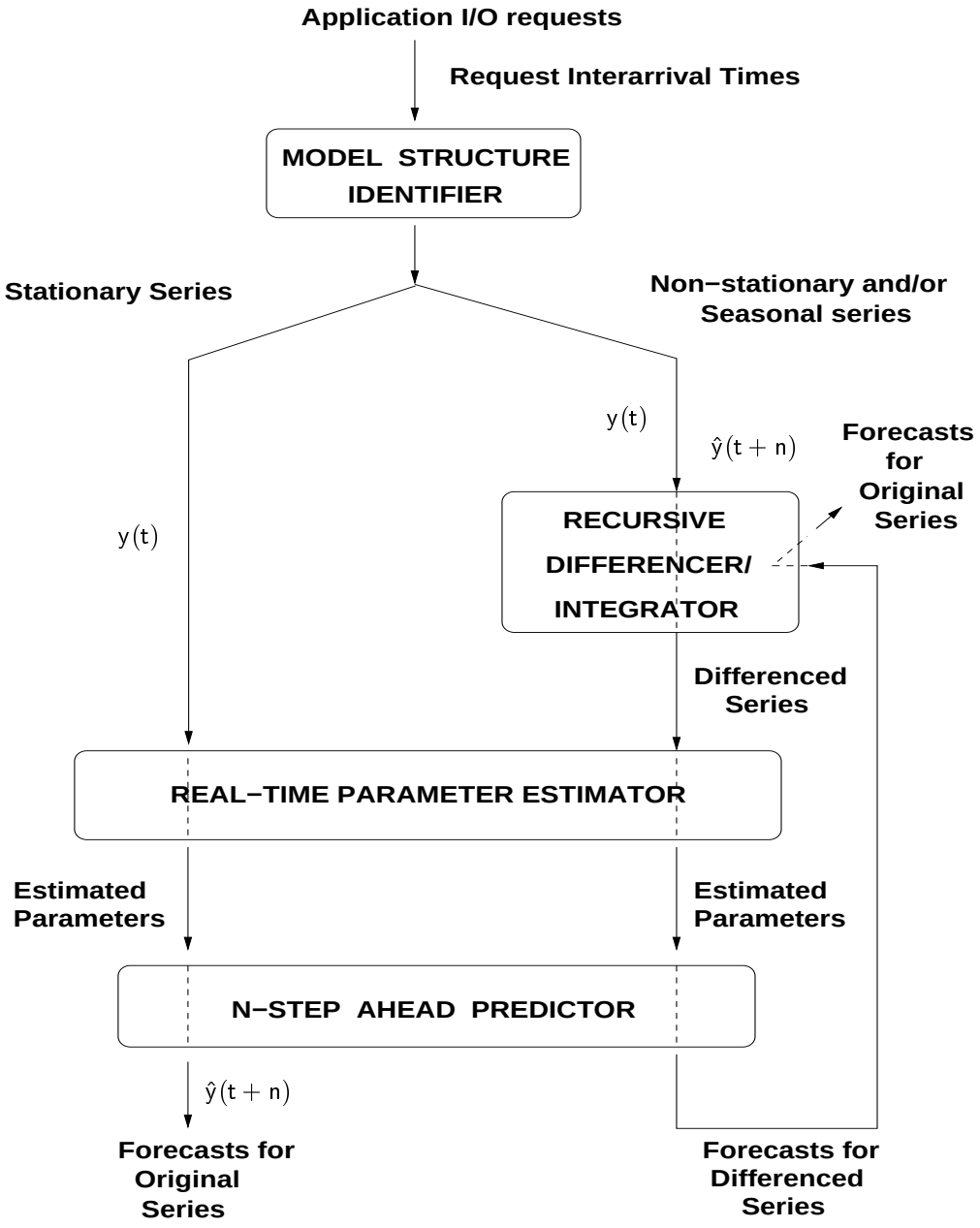

Figure 3.1: ARIMA Automodeler Architecture

tification costs, we choose a *simpler test based on the final forecast errors*, at the expense of accuracy. For example, if the relative forecast errors are less than 20%, the model is accepted. We believe that models for read/write interarrival times do not need to be very accurate for prefetching. From an I/O perspective, there is no large difference between 300 and 400 microseconds in request interarrival times.

- A *real-time parameter estimator* to dynamically estimate model parameters for stationary series using the recursive, extended least squares algorithm. This algorithm takes advantage of recursion to reduce computation overhead and the least squares technique to minimize the sum of the squared estimation errors. Least squares make two assumptions: that the models are linear in their parameters and that estimation errors are normally distributed with zero mean and constant variance. When these assumptions are not violated, least squares can provide reasonably accurate parameter estimates.

  I/O temporal access patterns in scientific applications are often more involved than simple stationary processes. To provide support for more complex I/O patterns, we designed a recursive differencing/integration algorithm.

- A *recursive differencer/integrator* to transform non-stationary and/or seasonal series into stationary series before applying the ELS algorithm for model parameter estimation. Transformation is achieved by filtering the series through the recursive RDI algorithm. Because estimated parameters are associated with the stationary series, differences removed during transformation must be reintegrated to recover predictions for the original series. For example, to successfully model and predict interarrival times in the ESCAT series, these differencing and integration operations are required.

- A *n-step ahead predictor* to produce forecasts for n time steps ahead based on parameters estimated by ELS. As file request arrivals can come in bursts with small interarrival times, looking ahead just one time step is not sufficient (i.e., a fixed prefetch horizon of one block). Prefetching too few blocks may cause late prefetches. Instead, effective prefetching demands longer range predictions, allowing multiple-block prefetching to more closely adapt to variations in I/O request arrivals.

Because model identification and parameter estimation use different algorithms, the four steps above are described in two chapters for clarity. First, we discuss real-time parameter estimation and forecasting in this chapter, because they are needed to test a model's adequacy. In the following chapter, we will examine model identification techniques.

3.4 Real-Time Parameter Estimator

In this section, we present the *structures and components of stationary* ARMA(p,q) models. Understanding these basic models is fundamental to identifying their roles in capturing the dependencies among observations and estimating their uncertainties. Recursive techniques used at run time to approximate the magnitudes of these dependencies and uncertainties, known as real-time parameter estimation, will be described in detail.

Specifically, we will examine the *evolution* of the estimation algorithm – from ordinary least squares for *offline* estimation of *autoregressive* processes, to recursive least squares for *online* estimation, and finally to extended least squares (ELS) for online estimation of *autoregressive and moving average* ARMA(p,q) processes. We will outline the major components in our implementation of the parameter estimator, and provide examples to illustrate the control flow of the recursive operations.

3.4.1 ARMA(p,q) Models for Stationary Series

To mathematically characterize the behavior of I/O request interarrival times $y(t)$ for stationary processes, we can use an ARMA(p,q) model:

$$\begin{aligned} y(t) &= a_0 + a_1 y(t-1) + a_2 y(t-2) + \cdots + a_p y(t-p) + e(t) \\ &\quad + b_1 e(t-1) + b_2 e(t-2) + \cdots + b_q e(t-q) \end{aligned} \tag{3.1}$$

This model describes the relationships among interarrival times of a stationary process in two parts:

- an autoregressive (AR) part: a linear combination of the past p observations $y(t-1) \ldots y(t-p)$, weighted by p linear coefficients $a_1 \cdots a_p$, plus the constant term a_0 and the current

disturbance term $e(t)$.

- a moving (MA) average part: a linear combination of the past q disturbances $e(t-1) \ldots e(t-q)$, weighted by q linear coefficients $b_1 \cdots b_q$.

The disturbance terms, representing noise and uncertainty, are assumed to be Gaussian (i.e., normally, identically, and independently distributed with mean zero and constant variance σ^2). This normality assumption is based on the central limit theorem: if the number of samples is sufficiently large, the distribution of the sample mean will be approximately normal. Because the number of I/O requests in most long-running scientific applications is huge (e.g., with a grid size of $50 \times 50 \times 50$, the Cactus Wavetoy application [49] issues more than 20 million read requests in ten minutes, on a PC equipped with an Intel 266 MHz processor), we can justify making the normality assumption.

An Example without the Constant Term

To understand the model's rationale, consider the following ARMA(2,1) model, representing the stationary behavior of a synthetic series of file request interarrival times:

$$y(t) = 0.7y(t-1) - 0.1y(t-2) + e(t) + 0.05e(t-1) \tag{3.2}$$

With two elements in the autoregressive portion ($p = 2$), this model specifies that the two most recent interarrival times $y(t-1), y(t-2)$ are significantly correlated with the current interarrival time $y(t)$. The magnitudes of these correlations are quantified through the coefficients (parameters) 0.7 and -0.1. They suggest that the current interarrival time is 70% positively correlated to the most recent observation, but only 10% negatively correlated to the second most recent observation. All other earlier interarrival times are not statistically significant and discarded. In short, the ARMA model tells us how far back we need to keep historical information to infer the future.

Any remaining behavior, unexplained by the autoregressive portion, is attributed to noise and captured by the moving average component. With one element ($q = 1$) in this component, the model indicates that the past noise term, $e(t-1)$, has a positive influence on the current interarrival time for 5%. All previous noise terms are non-significant and ignored. By the Gaussian distribution assumption, the current noise term $e(t)$ has an expected value of zero.

An Example with the Constant Term

In the example above, the mean μ is assumed to be zero. To gain insight into the significance of the constant term a_0, let's consider a non-zero mean, say, $\mu = 10$, applied to Equation 3.2. Then we have:

$$\begin{aligned} y(t) - \mu &= 0.7[y(t-1) - \mu] - 0.1[y(t-2) - \mu] + e(t) + 0.05e(t-1) \\ y(t) &= \mu(1 - 0.7 + 0.1) + 0.7y(t-1) - 0.1y(t-2) + e(t) + 0.05e(t-1) \\ y(t) &= 4 + 0.7y(t-1) - 0.1y(t-2) + e(t) + 0.05e(t-1) \end{aligned}$$

In this model, the constant 4 approximates a level about which the time series fluctuates. It is not the mean, but *summarizes the effects of various correlations on the mean.*

Given a stationary series of I/O request interarrival times, represented by an ARMA(p,q) model, the next section will discuss how to estimate model parameters a_i and b_i. First, we start with the offline, ordinary least squares technique to estimate parameters for pure autoregressive AR(p) models (no moving average portion). Then, the least squares method is modified to become recursive, and further extended to allow online parameter estimation for complete stationary ARMA(p,q) processes. Here, we *assume p and q are known a priori.* In the next chapter, we will describe techniques to determine p and q automatically.

3.4.2 Ordinary Least Squares Parameter Estimation for AR(p) Models

Rationale

In linear systems theory, the *ordinary least squares* algorithm has been widely used to estimate the parameters of stationary series with *only autoregressive components*:

$$y(t) = a_0 + a_1 y(t-1) + a_2 y(t-2) + \cdots + a_p y(t-p) + e(t)$$

It works as follows:

1. Starting from time step $(t-1)$ with a set of initial values for the coefficients $a_0, a_1 \cdots a_p$ and a set of past observations $y(t-1) \cdots y(t-p)$, the *model predicted value* for the next time step

t is computed as a linear combination of the past p observations:

$$a_0 + a_1 y(t-1) + a_2 y(t-2) + \cdots + a_p y(t-p)$$

2. At the next time step (t), the difference between the observed value $y(t)$ and the model predicted value is calculated. This difference, known as the *a priori prediction error*, is simply the noise term:

$$e(t) = y(t) - a_0 - a_1 y(t-1) - a_2 y(t-2) - \cdots - a_p y(t-p) \tag{3.3}$$

3. Computing the prediction error $e(t)$ for $p+1$ time steps results in a system of equations with $p+1$ unknown parameters $(a_0, a_1, \cdots, a_p)$ to be estimated:

$$\Delta(t) = \varphi(t) - \Psi(t-1)\gamma(t-1) \tag{3.4}$$

where:

- $\varphi(t)$ is the current column *observation vector*. It contains the observation $y(t)$ and the previous p observations.
- $\Psi(t-1)$ is the previous *observation matrix*. Each row in this matrix consists of the constant term 1, associated with a_0, followed by the previous p observations, $y(t-1)\ \cdots\ y(t-p)$.
- $\gamma(t-1)$ is the *parameter vector*, containing all the estimated parameters, $a_0, a_1, \ldots, a_p$.
- $\Delta(t)$ is the *prediction error vector*. It includes the differences between the current observation vector $\varphi(t)$ and the vector of predicted values $\Psi(t-1) \times \gamma(t-1)$.

4. *Minimizing the sum of squares of the prediction errors* in the vector $\Delta(t)$, with respect to all the parameters in $\gamma(t-1)$, gives the least squares parameter estimates. Squaring the prediction errors in Equation 3.4 and minimizing the sum of these squared errors, we have:

$$\begin{aligned} \min_{\gamma} \sum_{i=t-p+1}^{t} [e(i)]^2 &= \min_{\gamma} \Delta^T(t) \times \Delta(t) \\ &= \min_{\gamma} \ [\ \varphi(t) - \Psi(t-1)\gamma(t-1)]^T \times [\ \varphi(t) - \Psi(t-1)\gamma(t-1)] \end{aligned}$$

This error minimization system can be solved[1] by setting its partial derivatives (with respect to each parameter in γ) to zero. The result is the well-known estimation equation:

$$\gamma(t-1) = \Psi^T(t-1)\Psi(t-1))^{-1} \times \Psi^T(t-1)\varphi(t) \tag{3.5}$$

Below is an example illustrating how to apply ordinary least squares to model I/O behavior *offline*.

An I/O Example

This example uses ordinary least squares to estimate parameters for a short, synthetic series of file request interarrival times, fitted with a pure autoregressive AR(2) model. The current interarrival time $y(t)$ is statistically correlated to only two previous arrivals $y(t-1), y(t-2)$:

$$y(t) = a_0 + a_1 y(t-1) + a_2 y(t-2) + e(t)$$

The series contains 10 chronological observations, measured in milliseconds, and arranged from left to right with the latest interarrival time on the far right:

38.92, 48.68, 36.11, 51.17, 30.88, 52.95, 29.41, 55.49, 24.48, 60.93

Applying this series to estimate parameters via Equation 3.5 results in the following expansion:

[1] For a proof, refer to [68], Chapter §12 on multiple linear regression

$$\gamma(t-1) = \Psi^T(t-1)\Psi(t-1))^{-1} \times \Psi^T(t-1)\varphi(t) \qquad \text{(from equation 3.5)}$$

$$\gamma(t-1) = \left(\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 24.48 & 55.49 & 29.41 & 52.95 & 30.88 & 51.17 & 36.11 & 48.68 \\ 55.49 & 29.41 & 52.95 & 30.88 & 51.17 & 36.11 & 48.68 & 38.92 \end{bmatrix} \begin{bmatrix} 1 & 24.48 & 55.49 \\ 1 & 55.49 & 29.41 \\ 1 & 29.41 & 52.95 \\ 1 & 52.95 & 30.88 \\ 1 & 30.88 & 51.17 \\ 1 & 51.17 & 36.11 \\ 1 & 36.11 & 48.68 \\ 1 & 48.68 & 38.92 \end{bmatrix} \right)^{-1} \times$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 24.48 & 55.49 & 29.41 & 52.95 & 30.88 & 51.17 & 36.11 & 48.68 \\ 55.49 & 29.41 & 52.95 & 30.88 & 51.17 & 36.11 & 48.68 & 38.92 \end{bmatrix} \begin{bmatrix} 60.93 \\ 24.48 \\ 55.49 \\ 29.41 \\ 52.95 \\ 30.88 \\ 51.17 \\ 36.11 \end{bmatrix}$$

where

- The observation vector $\varphi(t)$ is formed with the current interarrival time $y(t) = 60.93$ placed at element 0; the past observation $y(t-1) = 24.48$ at element 1, etc.
- Similarly, the observation matrix $\Psi(t-1)$ is formed with the past two interarrival times stored in the top row, i.e., [1 24.48 55.49]; the next past two interarrival times [1 55.49 29.41] in the second row, etc. The first element with a value of 1 is associated with the parameter a_0.

Hence, the estimated parameters for our AR(2) example model are:

$$\gamma(t-1) = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 40.61 \\ -0.61 \\ 0.63 \end{bmatrix}$$

The parameters indicate that request interarrival times fluctuate around a level of 40.61 milliseconds. These fluctuations are about 61% negatively correlated to the last arrival and 63% positively correlated to the second last arrival.

One major drawback of the ordinary least squares approach is that it is not amenable to online parameter estimation, due to the following difficulties:

- Matrix inversion is required for every round of parameter estimation. This operation is computationally expensive, especially when the matrix is large. Large computation overhead can cancel or even outweigh the benefits of prefetching.
- Because the entire series is used for each estimation, the size of the observation matrix increases with each new I/O request arrival. As a result, estimation cost can become prohibitively high as the application progresses.

3.4.3 Recursive Least Squares Parameter Estimation for AR(p) Models

To achieve efficient online parameter estimation, recursive algorithms built atop the ordinary least squares method exist [2, 52]. They are founded on two principles:

- *Recursion* to reuse previously estimated parameters, eliminating the need for storing and computing *all* the data points in a series. Using recursion, one can adjust the previous parameters to compute new parameters for every new data arrival. The adjusted amount, known as the *gain factor*, increases or decreases depending on variations in interarrival times.
- *Matrix inversion avoidance.* By applying the matrix inversion lemma [34], expensive computations associated with inverting the observation matrix are avoided. Combined with recursion, the lemma allows us to directly derive an inverted matrix from that computed during the previous time step.

In a future section on implementation, we provide an example illustrating how to recursively compute the gain factor while avoiding matrix inversion, achieving online parameter estimation for I/O request interarrival times. However, recursive least squares are only suited for modeling pure autoregressive AR(p) processes. It must be extended to model complete ARMA(p,q) processes.

3.4.4 Extended Least Squares Estimation for ARMA(p,q) Models

I/O access patterns of scientific applications executing in real systems are often subject to noise. To statistically model these processes with noise components (i.e., pure MA(q) and complete ARMA(p,q) processes), the recursive least squares method is augmented to give the extended least squares (ELS) algorithm. ELS exploits bootstrapping techniques to compute estimates for the noise terms $e(t-1), e(t-2)$, etc. in the moving average component of an ARMA(p,q) process:

$$b_1 e(t-1) + b_2 e(t-2) + \cdots + b_q e(t-q)$$

A Posteriori Prediction Errors

In time series, noise represents the aggregate of all remaining portions in the data that cannot be explained by the main structure of a model. Possibly coming from several unknown or hidden sources, noise is often estimated. ELS uses the *a posteriori* prediction errors as noise estimates [2]. The key idea is to reflect the latest system dynamics by computing prediction errors based on the *most recent* parameters in $\gamma(t)$, instead of the previous parameters in $\gamma(t-1)$. Contrasting with the a posteriori prediction errors, errors computed using the previous parameters are defined as *a priori* prediction errors.

The a posteriori prediction error at time t is simply the difference between the *current* observation $y(t)$ and the predicted value $\phi^T(t-1)\gamma(t)$, computed based on the *newly estimated* parameters in $\gamma(t)$, instead of $\gamma(t-1)$:

$$e(t) = y(t) - \phi^T(t-1)\gamma(t)$$

$\phi^T(t-1)$ is the transpose of the augmented observation vector at time $t-1$. It contains the past p observations and the past q noise estimates, augmented with the constant 1 to represent the parameter a_0:

$$\phi^T(t-1) = [1 \;\; y(t-1) \;\; \ldots \;\; y(t-p) \;\; e(t-1) \;\; \ldots \;\; e(t-q)]$$

Hence, the a posteriori prediction error at time t is:

$$e(t) = y(t) - a_0 - a_1 y(t-1) \;\; \ldots - a_p y(t-p) - b_1 e(t-1) \;\; \ldots - b_q e(t-q) \tag{3.6}$$

Noise Bootstrapping Via A Posteriori Prediction Errors

Bootstrapping the q noise terms is achieved by incrementally moving forward q time steps. At each step i, the a posteriori prediction error $e(i-1)$ of the previous step is used as the latest noise term in the observation vector for the current step. Below we illustrate the bootstrapping mechanism.

At startup, based on the Gaussian noise distribution assumption, all unknown, past noise terms, $e(t-1),\ \ldots,\ e(t-q)$, in an ARMA(p,q) model are initialized to zero. As a result, we have:

$$e(t) = y(t) - a_0 - a_1 y(t-1)\ \ldots - a_p y(t-p)$$

To compute the prediction error for the next time step $e(t+1)$, one uses the a posteriori prediction error $e(t)$. Similarly, to compute the prediction error for two time steps ahead $e(t+2)$, one uses $e(t+1)$ and $e(t)$. This bootstrapping procedure repeats for every observation, using the most recent prediction error as the latest noise term. More explicitly,

$$\begin{aligned}
e(t) &= y(t) - a_0 - a_1 y(t-1)\ \cdots\ - a_p y(t-p) \\
e(t+1) &= y(t+1) - a_0 - a_1 y(t)\ \cdots\ - a_p y(t-p+1) - b_1 e(t) \\
e(t+2) &= y(t+2) - a_0 - a_1 y(t+1)\ \cdots\ - a_p y(t-p+2) - b_1 e(t+1) - b_2 e(t) \\
&\cdots \\
e(t+q) &= y(t+q) - a_0 - a_1 y(t+q-1)\ \cdots\ - a_p y(t+q-p) - b_1 e(t+q-1)\ \cdots - \\
&\quad b_{q-1} e(t+1) - b_q e(t)
\end{aligned}$$

3.4.5 Implementation

Outlined in Figure 3.2, our implementation for the ELS parameter estimation includes three phases.

- In phase one, initial conditions for recursion are set following Astrom's suggestions [2]. The parameter vector is initialized to null, $\gamma(0) = 0$. The inverted covariance[2] matrix R is primed with an identity matrix, having very large values on the main diagonal: $R^{-1}(0) =$ Identity matrix $\times 10^6$.

[2] Covariances measure how much I/O request interarrival times vary with respect to each other.

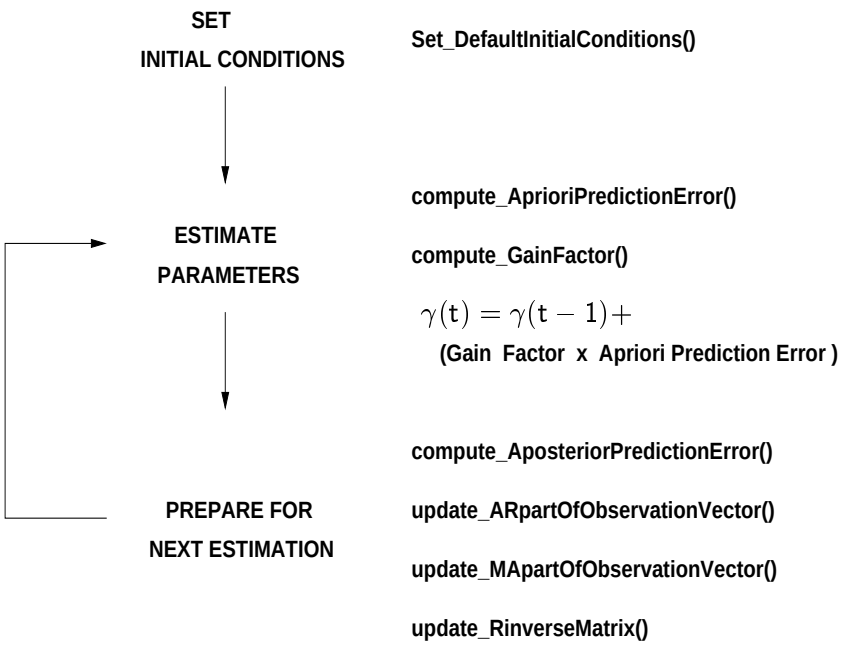

Figure 3.2: Outline of the Implemention of ELS

- In phase two, new parameters in the vector $\gamma(t)$ are estimated based on the a priori prediction error and the gain factor. The a priori prediction error is obtained through Equation 3.3. The gain factor is recursively calculated from the previous gain, the observation vector $\varphi(t)$, and the inverted covariance matrix $R^{-1}(t)$.

- Phase three updates all related structures and variables to prepare for the next round of estimation. First, the a posteriori prediction error in Equation 3.6 is computed as part of noise bootstrapping. Then, to update the autoregressive part of the observation vector $\varphi(t)$, the current observation $y(t)$ is shifted in, removing the oldest observation. Similarly, the newly bootstrapped noise term is shifted into the moving average part, deleting the oldest noise term. Finally, the inverted covariance matrix, $R^{-1}(t)$, is recursively updated from the previous $R^{-1}(t-1)$ matrix and the current gain factor.

Phases two and three are executed for every new I/O request. Below is an example that illustrates the three phases in the implementation, showing how ELS can be applied to estimate parameters recursively.

Example

Suppose we have a short, synthetic series of ten I/O request interarrival times, fitted with an ARMA(1,1) model

$$y(t) = a_0 + a_1 y(t-1) + e(t) + b_1 e(t-1)$$

Interarrival times for the series, expressed in milliseconds, are arranged from left to right, with the most recent interarrival time on the far right:

$$49, \quad 52, \quad 57, \quad 62, \quad 67, \quad 65, \quad 60, \quad 56, \quad 48, \quad 53$$

Below are snapshots of the three phases in the implementation of the extended least squares algorithm, applied to the series above.

1. **Phase 1 - Set Initial Conditions**

   - Because there is only one autoregressive element, the observation vector $\varphi(t-1)$ is primed with only the first interarrival time, 49. Similarly, the single moving average element, representing the past noise term, is assumed to be zero (mean of the assumed Gaussian distribution). The constant 1 in the first entry is associated with the parameter a_0. Hence,

   $$\varphi(t-1) = \begin{bmatrix} 1 \\ 49 \\ 0 \end{bmatrix}$$

   - The inverted covariance matrix is initialized with very large values on the main diagonal:

   $$\begin{bmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{bmatrix}$$

   It is a symmetric matrix with a dimension of 3, determined by the total number of autoregressive and noise elements plus one for the parameter a_0.

2. **Phase 2 - Estimate Parameters**. Parameter estimation starts with the second interarrival time, 52.

- The a priori prediction error is computed as the difference between the new interarrival time, 52, and the model predicted value. The latter is the product of the transpose of observation vector $\varphi^T(t-1) = [1,\ 49,\ 0]$ and the previously estimated parameter vector $\gamma(t-1)$, which was initialized to zero:

$$\text{A priori Prediction Error} = 52 - [1,\ 49,\ 0] \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 52$$

- The gain factor is calculated based on the observation vector $\varphi(t-1)$ and the inverted covariance matrix $R^{-1}(t-2)$ using the equation below. Intuitively, the denominator in the equation represents a correction by which the old gain is adjusted. It approaches 1 as the algorithm converges, closing the gap between the old and new gains.

$$\text{Gain Factor} = \frac{R^{-1}(t-2)\varphi(t-1)}{1 + \varphi^T(t-1)R^{-1}(t-2)\varphi(t-1)} \tag{3.7}$$

Hence,

$$\text{Gain Factor} = \frac{\begin{bmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{bmatrix} \begin{bmatrix} 1 \\ 49 \\ 0 \end{bmatrix}}{1 + [1,\ 49,\ 0] \begin{bmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{bmatrix} \begin{bmatrix} 1 \\ 49 \\ 0 \end{bmatrix}} = \begin{bmatrix} 0.0004 \\ 0.0204 \\ 0 \end{bmatrix}$$

- Finally, model parameters in $\gamma(t)$ are recursively estimated from the previous parameters in $\gamma(t-1)$, using the a priori prediction error and the gain factor:

$$\gamma(t) = \gamma(t-1) + (\text{Gain Factor} * \text{A priori Prediction Error})$$

$$\gamma(t) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.0004 \\ 0.0204 \\ 0 \end{bmatrix} \times 52 = \begin{bmatrix} 0.0208 \\ 1.0608 \\ 0 \end{bmatrix}$$

3. **Phase 3 - Prepare for Next Estimation**. The observation vector, the prediction errors, and the covariance matrix are all updated to prepare for parameter estimation when the next I/O request arrives.

   - First, the a posteriori prediction error is computed as the difference between the current interarrival time, 52, and the latest predicted value, the product of the transpose of the observation vector $\varphi(t-1)$ and the new parameter vector $\gamma(t)$.

$$\text{A posteriori Prediction Error} = 52 - [1,\ 49,\ 0] \begin{bmatrix} 0.0208 \\ 1.0608 \\ 0 \end{bmatrix} = 0$$

   - The inverted covariance matrix $R^{-1}(t-1)$ is recursively updated from the previous covariance matrix $R^{-1}(t-2)$ and the observation vector $\varphi(t-1)$. It is computed based on the recursive least squares equation, using the gain factor to avoid matrix inversion:

$$R^{-1}(t-1) = R^{-1}(t-2) \ - \ \text{Gain Factor} \times \ \varphi^T(t-1)\ R^{-1}(t-2) \qquad (3.8)$$

$$\begin{bmatrix} 999600 & -20400 & 0 \\ -20400 & 400 & 0 \\ 0 & 0 & 10^6 \end{bmatrix} = \begin{bmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{bmatrix} - \begin{bmatrix} 0.0004 \\ 0.0204 \\ 0 \end{bmatrix} [1,\ 49,\ 0] \begin{bmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{bmatrix}$$

   - Finally, the current interarrival time, 52, is shifted into the autoregressive portion of the observation vector $\varphi(t-1)$. Similarly the a posteriori prediction error, 0, is shifted into the moving average portion, ready for the next estimation:

$$\varphi(t-1) = \begin{bmatrix} 1 \\ 52 \\ 0 \end{bmatrix}$$

Phases 2 and 3 are reiterated for each remaining interarrival time in the series (57, 62, 67 ... 53), recursively estimating new parameter vectors, which will be used to predict future requests.

3.5 Recursive Differencer/Integrator for Non-Stationary and Seasonal Series

Many scientific applications have I/O temporal access patterns that are more complex than can be represented by stationary processes alone; the ESCAT series is an example. We need to expand the ELS algorithm to accommodate non-stationary and seasonal processes.

Several transformation techniques can render these processes stationary such that ELS and prediction techniques developed for stationary processes can be reused. Differencing is one commonly used transformation for time series with stable variances. Series with non-stable variances can be modified via other transformations, e.g., logarithmic and fractional (the Box-Cox power transformations) [6], to limit the final variances within constant bounds. As noted earlier, ARIMA differencing also assumes that various sections of a non-stationary series behave alike, except for their local means or trends. The same assumption applies to ARIMA seasonal series, where similar patterns repeat periodically every season, as shown in the ESCAT example.

The scope of this thesis is limited to investigating differencing transformations alone. For these transformations, we develop a *recursive algorithm capable of efficiently executing at run time differencing/integration operations for every new interarrival time.* Our recursive framework can also be extended to accommodate other types transformations (e.g., the class of power transformations). We will discuss in more detail the implications of these transformations in Chapter §10.

3.5.1 Fundamental Concepts on Differencing

Differencing is a transformation aimed at stabilizing the means of non-stationary and seasonal series. Here, we propose using the *difference interval* to distinguish two types of differencing in ARIMA models. This simple idea allows us to design a unified algorithm for both non-stationary and seasonal processes, greatly reducing implementation complexity.

- **Regular Differencing** for non-stationary and *non-seasonal* time series. The *difference interval is one.* Consecutive observations in the original series are subtracted to produce the transformed series: $d(t) = y(t) - y(t-1)$. For example, consider a non-stationary series of increasing request interarrival times, expressed in milliseconds:

70.3, 100.5, 130.2, 160.7, 190.5, 220.2, 250.4

Differencing *consecutive interarrival times* gives a stationary series fluctuating around a mean of 30:

	100.5	130.2	160.7	189.5	220.2	250.4
-	70.3	100.5	130.2	160.7	189.5	220.2
	30.2	29.7	30.5	28.8	30.7	30.2

- **Seasonal Differencing** for seasonal (periodic) time series. The *difference interval S is greater than one*. Each element $D(t)$ in the transformed series is the difference between two observations separated by an interval S, *the season length*: $D(t) = y(t) - y(t - S)$. For example, consider the synthetic series in Figure 3.3 for a non-stationary and seasonal I/O arrival pattern. Every circled fifth interarrival time (50, 53.1, 56.4) delineates one season.

  Taking the differences between data points separated by a season length 5 gives a stationary series fluctuating around 3:

	6.3	9.7	12.3	15.1	53.1	9.4	12.5	15.5	18.2	56.4
-	3.2	6.3	9.5	12.1	50.0	6.3	9.7	12.3	15.1	53.1
	3.1	3.4	2.8	3.0	3.1	3.1	2.8	3.2	3.1	3.3

In the two examples above, differencing transformations are applied only once to the entire series to stabilize their means. However, in general, there are many other non-stationary and seasonal access patterns that need to be differenced more than once to reach stationarity.

The following example illustrates a series with more complex interarrival time patterns, requiring two levels of differencing transformations. Figure 3.4 shows the steps used to difference a synthetic interarrival time series. The series at the top exhibit non-stationary behavior, characterized by increasing interarrival times, with spikes seasonally interspersed at regular intervals of length 5, indicated by the circled data points.

Two levels of differencing are performed: one regular, one seasonal. After applying *regular differencing*, the transformed series (the second series in Figure 3.4) still contains seasonal components (9.8, 10.2, 10.4), although the differenced data within the seasons (1.3, 1.2, 0.9, 1.0 $\|$ 1.2, 1.1, 1.0, 1.3 $\cdots$) reaches statistical equilibrium. Hence, *seasonal differencing* the transformed series with

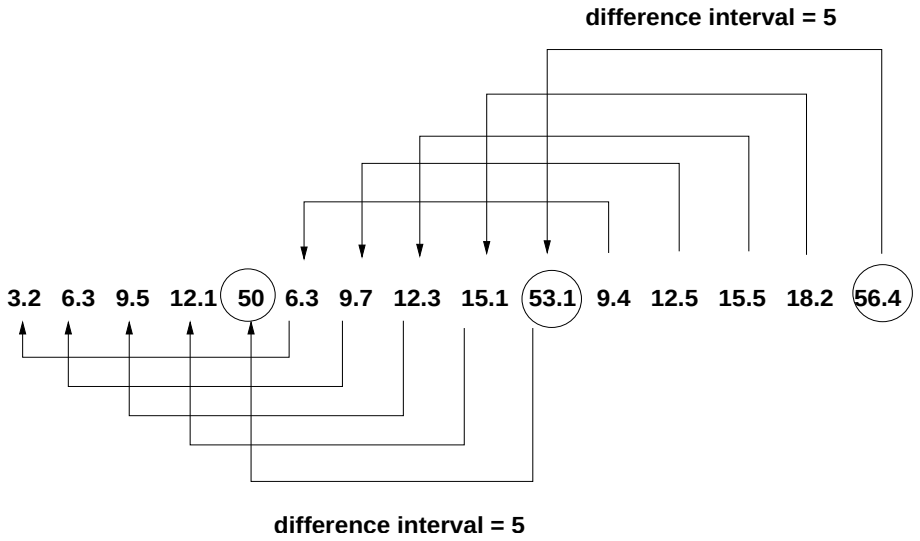

Figure 3.3: Differencing a Seasonal Series

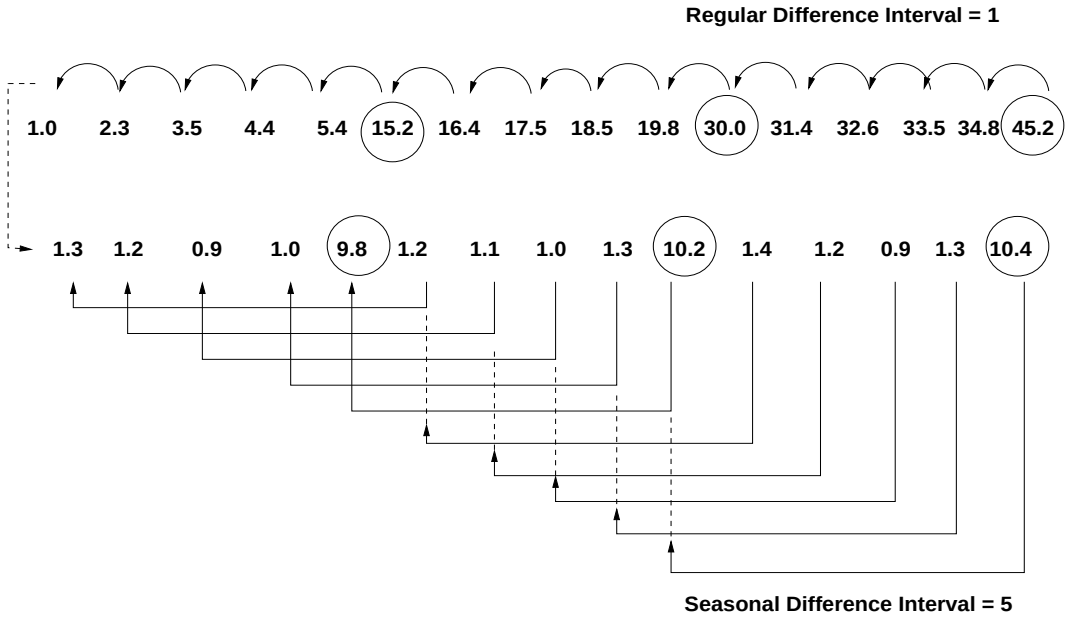

Figure 3.4: Two Levels of Differencing

an interval 5 is necessary. After the second transformation, the final series becomes stationary, fluctuating around a mean of 0.1:

	1.2	1.1	1.0	1.3	10.2	1.4	1.2	0.9	1.3	10.4
-	1.3	1.2	0.9	1.0	9.8	1.2	1.1	1.0	1.3	10.2
	-0.1	-0.1	0.1	0.3	0.4	0.2	0.1	-0.1	0.0	0.2

3.5.2 General ARIMA Models

$ARIMA(p, d, q)$ Model for Non-Stationary Processes

One can represent non-stationary processes by extending the ARMA(p,q) model to incorporate information on the differencing transformations, resulting in the ARIMA(p,d,q) model. This model specifies that after d *regular differencing* levels, the final stationary series will have p autoregressive terms and q moving average terms.

$ARIMA(p, d, q) \times (P, D, Q)_S$ Model for Seasonal Processes

To represent seasonal processes, Box and Jenkins further modified the $ARIMA(p, d, q)$ model, appending to the non-stationary model a seasonal component $(P, D, Q)_S$. This component specifies that after D *seasonal differencing* levels with a season length S, the correlation structure among those observations that are separated by an interval S is stationary, with P autoregressive terms and Q moving average terms.

Viewing the model $ARIMA(p, d, q) \times (P, D, Q)_S$ in its entirety – combining the non-stationary and seasonal components – one can have a more insightful interpretation. The non-stationary component (p, d, q) characterizes the correlations between observations *within a particular season*; the seasonal component $(P, D, Q)_S$ quantifies the correlations between observations of *successive seasons*.

The model $ARIMA(p, d, q) \times (P, D, Q)_S$ is also known as the *general ARIMA* model because it can represent all three types of processes (stationary, non-stationary, seasonal) and their combinations. Here, d and D are assumed to be integers. Supplying different numbers to the various parameters in the model gives different representations. For example, a stationary process can be described as $ARIMA(p, 0, q) \times (0, 0, 0)$, equivalent to an $ARMA(p, q)$ model.

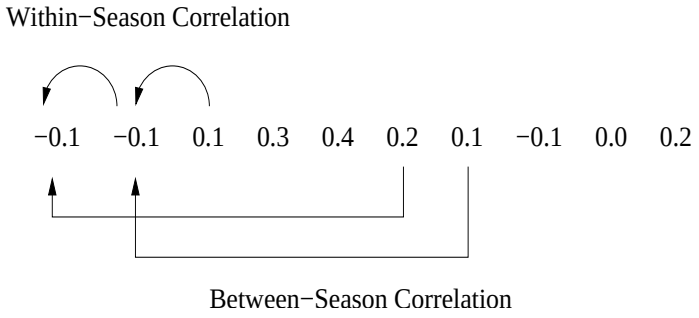

Figure 3.5: Two Types of Correlations

To illustrate the general ARIMA model, we continue exploring the seasonal differencing example in Figure 3.4. The example series requires two levels of differencing, one regular and one seasonal with a season length of 5 (i.e., $d = D = 1, S = 5$). Figure 3.5 shows two types of dependencies in the final differenced series: within a season and between seasons.

The model's non-seasonal component describes the within-season correlation (i.e., successive observations within a season). The seasonal component explains the between-season correlation (i.e., observations separated by 5 lags).

Now, suppose the non-seasonal component is a pure autoregressive AR(1) process ($p = 1, q = 0$), and the seasonal component is a pure moving average MA(1) process ($P = 0, Q = 1$). The general model, representing the example series, is thus $ARIMA(1, 0, 0) \times (0, 1, 1)_5$.

3.5.3 Cascading Property

The interarrival time pattern in Figure 3.4 also discloses an important property inherent in differencing transformations: the cascading property. It shows that results obtained from one level of differencing can be chained to the next level, for as many levels as required, to achieve stationarity. For example, regular differences from the original series in Figure 3.4 are cascaded to the second level to be modified by seasonal transformations.

To support a wide variety of non-stationary and seasonal I/O arrival patterns, ARIMA general models allow an arbitrary number of regular ($d \geq 1$) and seasonal ($D \geq 1$) differencing levels. Transforming these series can be accommodated by cascading the differencing operations $d + D$ times.

In the next section, we explore the cascading property to design the recursive differencing/

integration algorithm.

3.5.4 Recursive Differencer/Integrator (RDI)

To characterize non-stationary and seasonal arrival processes requiring an arbitrary number of differencing levels, we designed a recursive differencer/integrator to efficiently manage the sequence of cascaded operations. The design, illustrated in Figure 3.6, is founded on a *chain structure of RDIs*. Each RDI is responsible for handling one differencing level. The role of the chain is to facilitate the task of forwarding/returning differencing/integration results between adjacent RDIs. For $d + D$ differencing levels, an equal number of RDIs are dynamically created and added to the chain. We reuse the series in Figure 3.4 to build an example chain. Because this series needs to be regularly differenced once and seasonally differenced once, the cascaded sequence includes one regular and one seasonal RDI.

Each RDI has two components: a *season length* S to determine the differencing interval, and an *input queue* to store differencing results produced by the previous RDI. It performs two major tasks:

- **Differencing**: Each RDI saves in its input queue every new observation, or the difference result produced by a preceding RDI. This new value is subtracted from a previously queued value, separated by S time steps. The result is forwarded to the next RDI, where the same procedure is repeated.

  In the example illustrated in Figure 3.6, regular differencing is performed by the first RDI. This regular RDI saves the observed interarrival time in its input queue and takes the differences between consecutive observations because the difference interval S is one. Results (e.g., $1.3 = 2.3 - 1.0$) are then given to the second RDI in the chain to continue seasonal differencing.

  Because the difference interval for the second RDI is 5, five observations are accumulated in the input queue. The arrival of the sixth difference triggers seasonal differencing. Values separated by 5 time steps are subtracted for each new arrival, e.g., $1.2 - 1.3 = -0.1$. The final differences produce a stationary series, the parameters of which can be estimated by ELS.

- **Integration**: Because forecasts for future interarrival times are computed for the final differ-

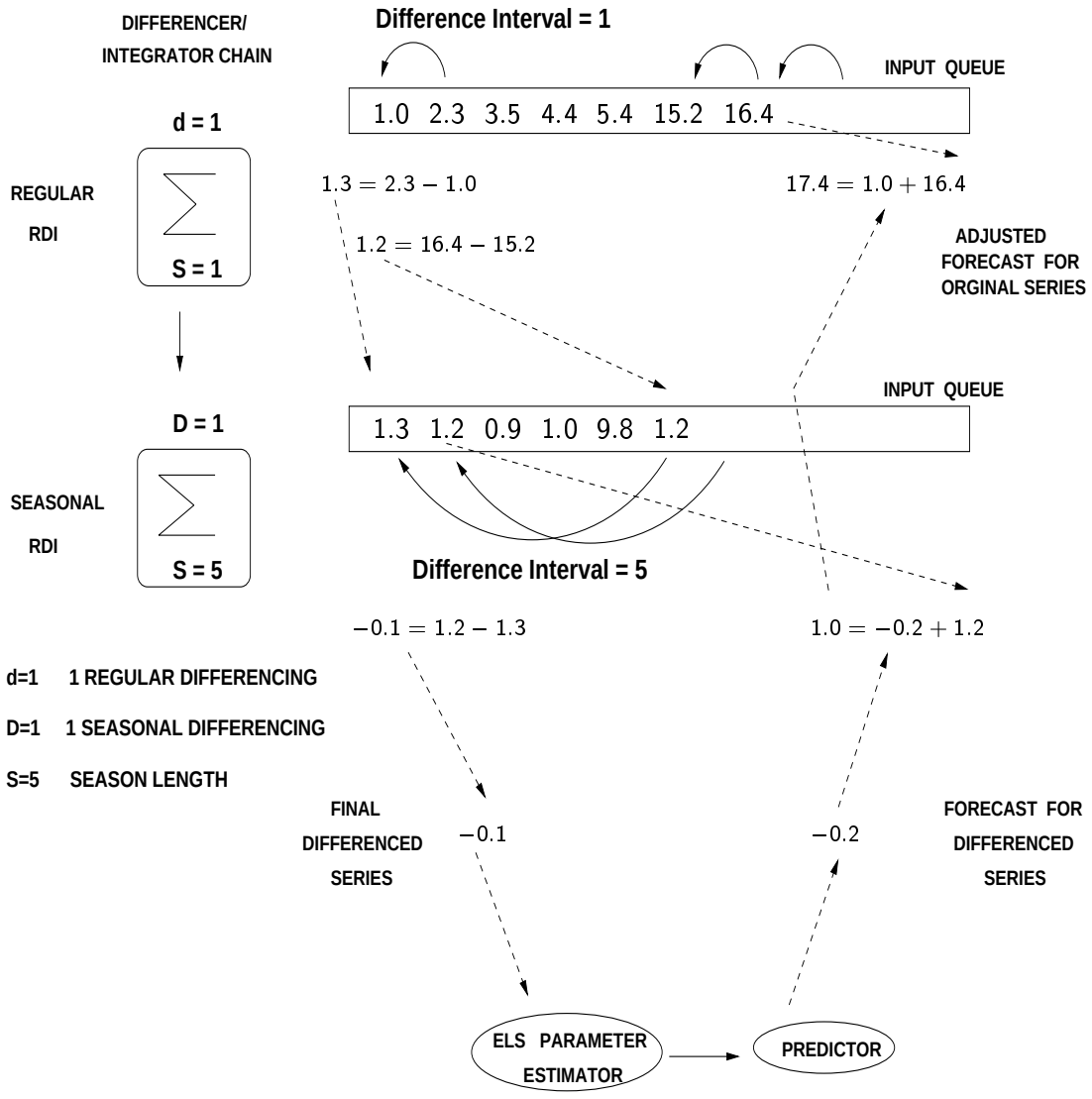

Figure 3.6: Recursive Differencer/Integrator Structure

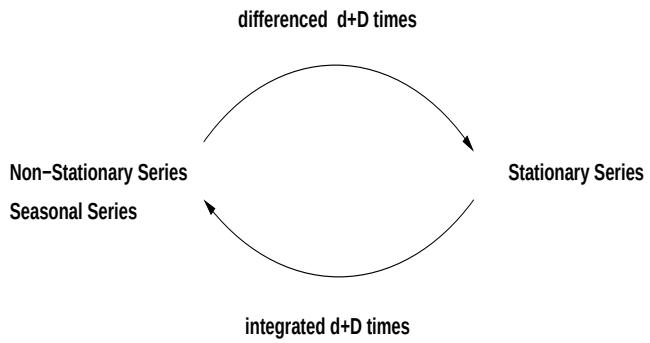

Figure 3.7: Duality between Differencing and Integration

enced stationary series, differences that were removed earlier must be re-integrated (summed) to recover forecasts for the original non-stationary or seasonal series. This duality between differencing and integration is illustrated in Figure 3.7.

The RDI can efficiently manage the integration process by simply traversing the chain structure upward to dequeue the appropriate difference results. This is possible because these differences were stored earlier in the input queues at every differencing level. In each RDI, forecasts computed for the transformed series are adjusted to reverse (re-integrate) the differences. Figure 3.6 shows these adjustments, ascending the RDI chain to the top.

To illustrate, assume that the one-step ahead forecast, generated by the final difference result, is -0.2 (the bottom right hand side of Figure 3.6). The seasonal RDI performs integration by adding to this value the difference 1.2, located right next to the actually removed difference 1.3. This special measure is crucial because 1.2 is associated with the one-step ahead forecast (shown by the second arrow on the second input queue), and not 1.3, which is associated with the old observation.

Similarly, continuing the traversal up the RDI chain, the newly integrated forecast 1.0, produced by the seasonal RDI, is now given to the first RDI to undergo integration for a second time: $1.0 + 16.4 = 17.4$. Again, the adjusted amount is 16.4 (not 15.2) because it is the difference associated with the one-step ahead forecast.

We implemented the differencing/integration algorithm in a RDI object along with two methods:

one for the differencing transformation and the other for the integration reversal. These methods are recursively applied at each differencing level for every new observation to achieve online modeling and prediction for non-stationary and seasonal processes.

3.5.5 Computation Cost

The three main types of operations executed in a RDI are differencing, enqueue, and dequeue. The RDI overhead for processing each interarrival time is the sum of the costs of the operations above, multipled by the total number of differencing levels $d + D$.

$$\text{Cost/Observation} = (C_{diff} + C_{enq} + C_{deq}) \times (d + D)$$

Hence, the RDI incurs computation costs proportional to the total number of read requests. Because the queue contents are merely real numbers, one can reduce the enqueuing/dequeuing costs by using arrays to implement queues.

3.6 N-Step Ahead Predictor

Our ultimate purpose of modeling I/O requests is to gain insight into future I/O behaviors via predictions. By tracking and adaptively re-estimating model parameters at run time, we hope to project meaningful forecasts from the recent past under time-varying system conditions.

In this section, we will explain the rationale used to compute the one-step ahead forecast and the technique to bootstrap n steps ahead from the one-step ahead forecast for stationary, non-stationary, and seasonal processes.

3.6.1 One-Step Ahead Forecast for Stationary Arrival Processes

Based on the criterion of *minimizing mean square forecast errors*, a one-step ahead forecast, starting from t, is the *conditional expectation* of the next observation arrival at time $t+1$, conditioned upon all previous arrivals, available up to and including the forecast origin at time t [7]. Thus, given an ARMA(p,q) model for interarrival times:

$$y(t+1) = a_0 + a_1 y(t) + \cdots + a_p y(t-p+1) + e(t+1) + b_1 e(t) + \cdots + b_q e(t-q+1)$$

the one-step ahead forecast, denoted by $\hat{y}(t+1)$, is

$$\hat{y}(t+1) = E[y(t+1) \mid y(t), y(t-1) \cdots]$$

Because all quantities in the model, except $e(t+1)$, are known at time t, and the Gaussian noise terms are assumed to be normally distributed $N(0, \sigma^2)$, we can substitute $e(t+1)$ with its expected value, which is zero at time t. As a result, a one-step ahead forecast is simply a linear combination of past interarrival times and past noise terms, weighted by estimated parameters a_i and b_i. The noise terms are bootstrapped from past prediction errors, as described in §3.4.4 for the extended least squares algorithm. Hence,

$$\hat{y}(t+1) = a_0 + a_1 y(t) + \cdots + a_p y(t-p+1) + b_1 e(t) + \cdots + b_q e(t-q+1)$$

3.6.2 N-Step Ahead Forecasts for Stationary Arrival Processes

Starting from the one-step ahead forecasts for stationary processes, the n-step ahead forecasts can be bootstrapped by incrementally stepping forward n time steps. Recall that, in the first equation for $\hat{y}(t+1)$, the first zero represents the value of $e(t+1)$:

$$\begin{aligned}
\hat{y}(t+1) &= a_0 + a_1 y(t) + \quad \cdots \quad + a_p y(t-p+1) + 0 + b_1 e(t) + \cdots + b_q e(t-q+1) \\
\hat{y}(t+2) &= a_0 + a_1 \hat{y}(t+1) + \cdots + a_p y(t-p+2) + 0 + b_1\ 0 \quad + \cdots + b_q e(t-q+2) \\
\hat{y}(t+3) &= a_0 + a_1 \hat{y}(t+2) + a_2 \hat{y}(t+1) + \cdots + a_p y(t-p+3) + 0 + b_1 0 + b_2 0 + \cdots + b_q e(t-q+2) \\
&\cdots \\
\hat{y}(t+n) &= a_0 + a_1 \hat{y}(t+n) + \cdots + a_p \hat{y}(t-p+n) + 0 + b_1 0 \quad + \cdots + b_q e(t-q+n)
\end{aligned}$$

At each time step $t+j$, $j \geq 1$,

- to compute the *autoregressive portion* of the next forecast $\hat{y}(t+j)$, previously forecast values $\hat{y}(t+j-1), \quad \hat{y}(t+j-2)$ are used as expected values of future interarrival times.

- to compute the *moving average portion*, future Gaussian noise terms $e(t+j), e(t+j+1)\ldots$ are substituted with their expected mean of zero.

3.6.3 N-Step Ahead Forecasts for Non-Stationary and Seasonal Arrival Processes

To avoid late prefetches when I/O requests arrive in bursts at high rates, long range forecasts are usually necessary, allowing multiple blocks to be prefetched. Forecasting n steps ahead for non-stationary and seasonal models includes two stages.

- First, one needs to find the n-step ahead forecasts for the *final transformed stationary* series, as outlined in §3.6.2. This gives: $\hat{y}(t+1), \quad \hat{y}(t+2), \quad \ldots \quad \hat{y}(t+n)$

- Next, for each forecast produced in stage I, one *applies the integration procedure* as outlined in §3.5.4. This procedure is illustrated on the right side of Figure 3.6. The key idea is to iteratively revert the differences removed during earlier transformations, traversing the chain of recursive differencers/integrators (RDI) in reverse order. These differences, located in the input queue of each RDI, can be found S time steps away. Recall that S = season length for a seasonal RDI and $S = 1$ for a regular RDI. The last integration at the top of the chain produces the final forecasts for the *original non-stationary and seasonal series.*

  Returning to the example in Figure 3.6, let us consider the one-step ahead forecast for the final transformed series $\hat{y}(t+1)$, assumed to be -0.2. Differences to be integrated are 1.2 for the seasonal RDI (located 5 time steps away) and 16.4 for the regular RDI (located one time step away). Adding these differences to $\hat{y}(t+1)$ gives the one-step ahead forecast $\hat{y\prime}(t+1)$ for the non-stationary and seasonal series:

$$\hat{y\prime}(t+1) = \hat{y}(t+1) + 1.2 + 16.4 = -0.2 + 1.2 + 16.4 = 17.4$$

  Now, suppose that the two-step ahead forecast $\hat{y}(t+2)$ for the transformed stationary series is -0.8. From Figure 3.6, differences associated with this forecast are 0.9 (5 steps away) for

the seasonal RDI and 17.4 (one step away) for the regular RDI. Adding these differences to $\hat{y}(t+2)$ gives the two-step ahead forecast $\hat{y\prime}(t+2)$ for the original series:

$$\hat{y\prime}(t+2) = \hat{y}(t+2) + 0.9 + 17.4 = -0.8 + 0.9 + 17.4 = 17.5$$

A similar integration procedure is used to obtain the remaining forecasts $\hat{y\prime}(t+3), \ \ldots \ \hat{y\prime}(t+n)$.

Note that, at the regular RDI, we use the new forecast 17.4 as an estimate of the actual difference (i.e., when the next read request actually arrives). In general, when the number of look-ahead steps n exceeds the differencing interval S, forecasts obtained at each RDI during previous steps are used as estimates of the to-be-integrated differences.

3.7 Summary

In this chapter, we have proposed a dynamic framework for time series modeling and forecasting. This framework can make real-time forecasts for three types of I/O arrival processes: stationary, non-stationary, and seasonal. The purpose for using real-time parameter estimation is to produce up-to-date forecasts for future interarrival times. These forecasts, combined with Markov model predictions of block accesses, allow prefetching for multiple blocks adaptively.

We used the extended least squares algorithm (ELS) to estimate model parameters for stationary processes. We supplemented ELS with a new recursive differencing transformation algorithm to estimate parameters for non-stationary and seasonal processes. We also developed a matching recursive integration algorithm to reverse the differencing transformations. We exploited the integration algorithm to design a forecaster that can predict future interarrival times for multiple time steps ahead, giving the basis for multiple block prefetching.

Parameter estimation and forecasting presented in this chapter are based on the assumption that the structure of an ARIMA model is known in advance. This approach is potentially restrictive, limited to a small set of applications, for which we have already identified a model. New applications must wait. To remove this restriction, we will automate the model structure identification process in the next chapter.

Chapter 4

Automatic Model Structure Identification

Model structure identification is the first stage in ARIMA modeling. One must estimate values for p, d, q and P, D, Q, S of an ARIMA model. Representing the structure of dependencies, these values are inferred by extracting the most significant correlations among observations in a time series.

The classical approach to identification is *offline inspection* of graphs to interpret and cull embedded correlation patterns. Unfortunately, this method has two major difficulties. First, new application codes cannot be immediately analyzed unless models have been identified in advance. Second, if an application changes behavior during program execution, the identified model may become obsolete.

Analyses of scientific applications give evidence of considerable I/O behavior changes, manifested as phases – initialization, read, compute and write – in a program execution. These phases often arise as a result of algorithm changes or program control transfers. For example, in Figure 4.1, there are two phase transitions in the read behavior of the PRISM code, a physics code to simulate fluid dynamics [26]. PRISM read interarrival times change drastically near the beginning (initialization) and the middle of the code execution. For this reason, a model identified via interarrival times collected during the initialization phase might not be usable in the subsequent phases.

In this chapter, we explore techniques to *automate the identification process*, estimating a model structure for I/O interarrival times during program execution. Section §4.1 introduces the Box-Jenkins methodology for offline autocorrelation analysis. Section §4.2 explains our approach to

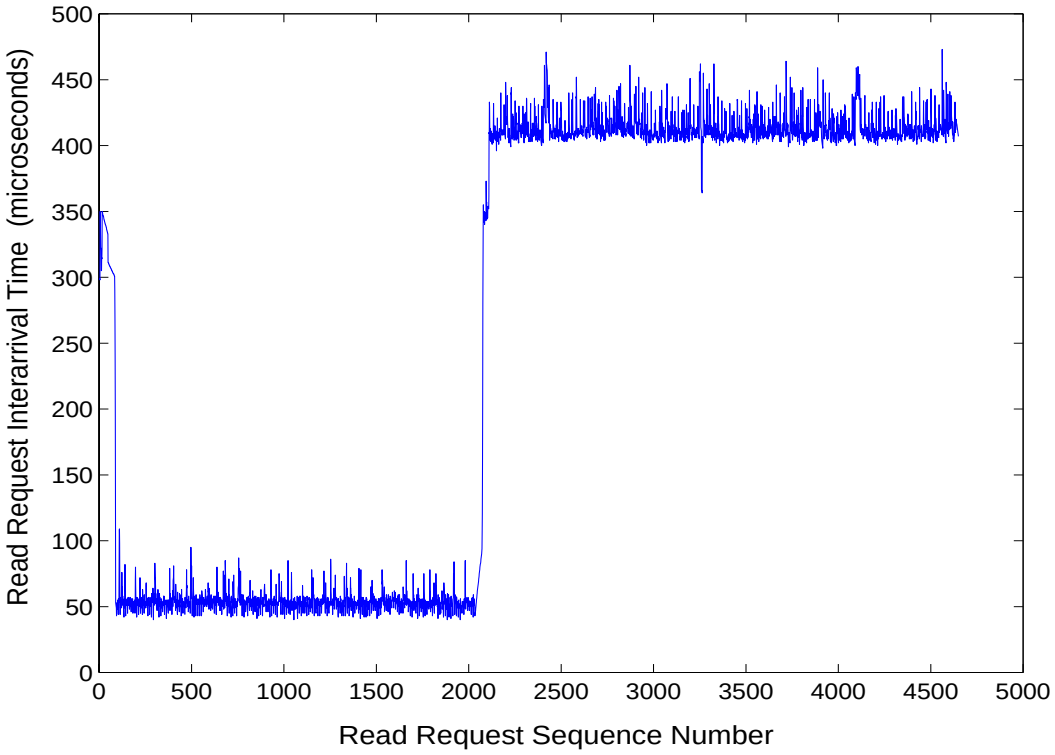

Figure 4.1: Phase Transitions

automating autocorrelation analysis. We designed several tests to assist identifying various patterns exposed by autocorrelations. A wavelet-based enhancement for *automatic detection of phase transitions* will be presented in the next chapter.

4.1 Autocorrelation Analysis

The fundamental graphical tools used by ARIMA to expose and identify data patterns are the autocorrelation function (ACF) and partial autocorrelation function (PACF). They are plots derived from autocovariances to show how much observations in the *same* time series (hence, the prefix auto) vary with respect to each other. Because covariances change with measurement units, measurements taken using different units cannot be compared directly (e.g., interarrival times measured in microseconds versus in seconds). For this reason, autocovariances are usually normalized by the variances to obtain autocorrelations, dimensionless statistics suitable for comparisons.

4.1.1 Autocorrelations

Autocorrelations measure the *linear dependencies* between observation pairs, $y(t)$ and $y(t+k)$, separated by $k = 1, 2, \cdots$ time periods (lags). For a given lag k, interpreting the pairs as realizations of the random variables Y_t and Y_{t+k}, the *autocorrelation at lag k* is defined as the *expectation of the variables' deviations from the mean, normalized by the variance*:

$$\text{autocorrelation at lag } k = \frac{\text{covariance at lag } k}{\text{variance}} = \frac{E(Y_t - \mu)(Y_{t+k} - \mu)}{\sigma^2}$$

This definition assumes that random variables in a stationary series vary about a common mean μ, with a constant variance σ^2. Autocorrelations at different lags have different values.

In practice, one can estimate the autocorrelation at lag k as the *average variation* of all observation pairs, separated by k lags, with respect to the sample mean $\hat{\mu}$, normalized by the sample variance $\hat{\sigma}^2$. For example, given a series with N data points, the autocorrelation r_5 at lag 5 is computed as:

$$r_5 = \frac{\frac{1}{N}\sum_{t=1}^{N-5}(y_t - \hat{\mu})(y_{t+5} - \hat{\mu})}{\hat{\sigma}^2} \quad \text{with}$$

$$\hat{\mu} = \frac{1}{N}\sum_{t=1}^{N}(y_t) \qquad \hat{\sigma}^2 = \frac{1}{N}\sum_{t=1}^{N}(y_t - \hat{\mu})^2$$

An I/O Example

Suppose we have a synthetic series of fifteen[1] read interarrival times, with sample mean of 48.4 and sample variance of 92.6.

42, 59, 35, 66, 37, 58, 49, 63, 52, 45, 36, 50, 43, 39, 52

Figure 4.2 shows the ACF of the series at the first ten lags. The first bar represents autocorrelation at lag 0; it is always 1 because each observation is completely correlated with itself. The second bar, depicting autocorrelation at lag 1, summarizes the average variation between consecutive observations: 42 and 59, 59 and 35, 35 and 66, etc. The same principles apply to autocorrelations

[1] In practice, at least 50 data points are needed for a meaningful interpretation of ACF and PACF [7].

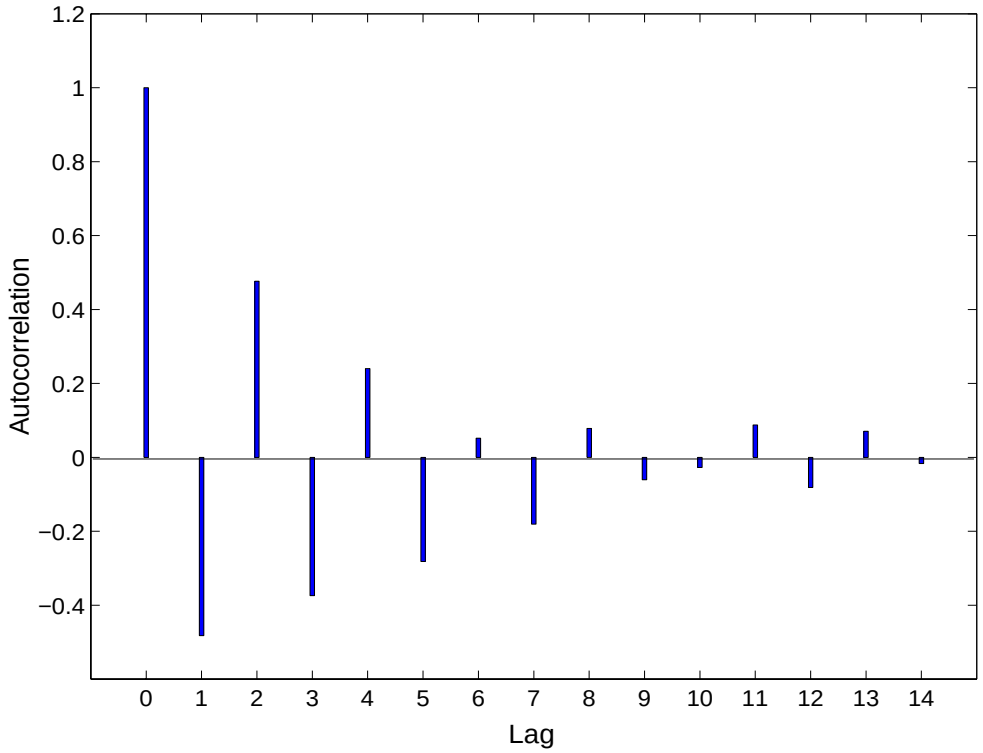

Figure 4.2: ACF of a Synthetic Series

at all other lags. Specifically, autocorrelation at lag 3 computes the average variation of observations that are 3 time steps apart: 42 and 66, 59 and 37, 35 and 58, etc.

The *sign and magnitude* of an autocorrelation specify the *general direction and strength* respectively of relationships between the observation pairs. In our example, the negative autocorrelation (-0.48) at lag 1 indicates that, in general, when one interarrival time increases, the next would decrease, and vice versa. On the other hand, the positive autocorrelation at lag 2 suggests that interarrival times separated by two lags tend to increase/decrease together. Overall, the ACF displays an alternating (sinusoidal) pattern that gradually decays to zero.

4.1.2 Partial Autocorrelations

In addition to measuring the correlations between pairs $y(t)$ and $y(t+k)$, partial autocorrelations take into account the *effects of intervening observations* $y(t+1), \cdots, y(t+k-1)$. Denoted as C_{kk}, the partial autocorrelation at lag k can be estimated by the coefficient associated with $y(t)$ in a linear combination shown below. The combination can be interpreted as a linear regression to

compute the effects of $y(t)$ and all the intermediate values on $y(t+k)$. Thus, C_{kk} represents the *contribution by* $y(t)$ *to the total effect.*

$$y(t+k) \;=\; C_{k1}\; y(t+k-1) + C_{k2}\; y(t+k-2) + \cdots + C_{kk-1}\; y(t+1) + C_{kk}\; y(t) + e(t)$$

As noted in Chapter §3, solving the regression equation via ordinary least squares is computationally intensive, because the *regression coefficients* C_{kj} have to be computed for every lag k (the first subscript of C_{kj}) and for every index j (the second subscript), running from 1 to k.

A less expensive solution was developed by Durbin [19] to *recursively approximate the regression coefficients* for ARIMA stationary series, using the autocorrelations r_k and the regression coefficients of previous lags. Below we illustrate Durbin's method for the first three lags.

- Lag 1: Initially, the partial autocorrelation is the same as the autocorrelation because there are no intermediate values between two adjacent observations: $C_{11} = r_1$

- Lag 2: Two coefficients C_{22} and C_{21} are computed based on the autocorrelations r_2 and r_1, along with the previous partial autocorrelation C_{11}.

$$\begin{aligned} C_{22} &= \frac{r_2 - C_{11}\; r_1}{1 - C_{11}\; r_1} \\ C_{21} &= C_{11} - C_{22}\; C_{11} \end{aligned}$$

- Lag 3: Similarly, three coefficients $C_{33},\; C_{32}$ and C_{31} are computed based on the previous autocorrelations $r_3,\; r_2,\; r_1$ together with the coefficients computed for lag 2: $C_{22},\; C_{21}$.

$$\begin{aligned} C_{33} &= \frac{r_3 - C_{21}\; r_2 - C_{22}\; r_1}{1 - C_{22}\; r_2 - C_{21}\; r_1} \\ C_{32} &= C_{21} - C_{33}\; c_{22} \\ C_{31} &= C_{22} - C_{33}\; c_{21} \end{aligned}$$

As the number of lags increases, the number of coefficients grows commensurately. Fortunately, Durbin's method allows recursive computations using previous results. In practice, for a series of N observations, Box and Jenkins suggested computing autocorrelations and partial autocorrelations for $N/4$ lags, based on a minimum of $N = 50$.

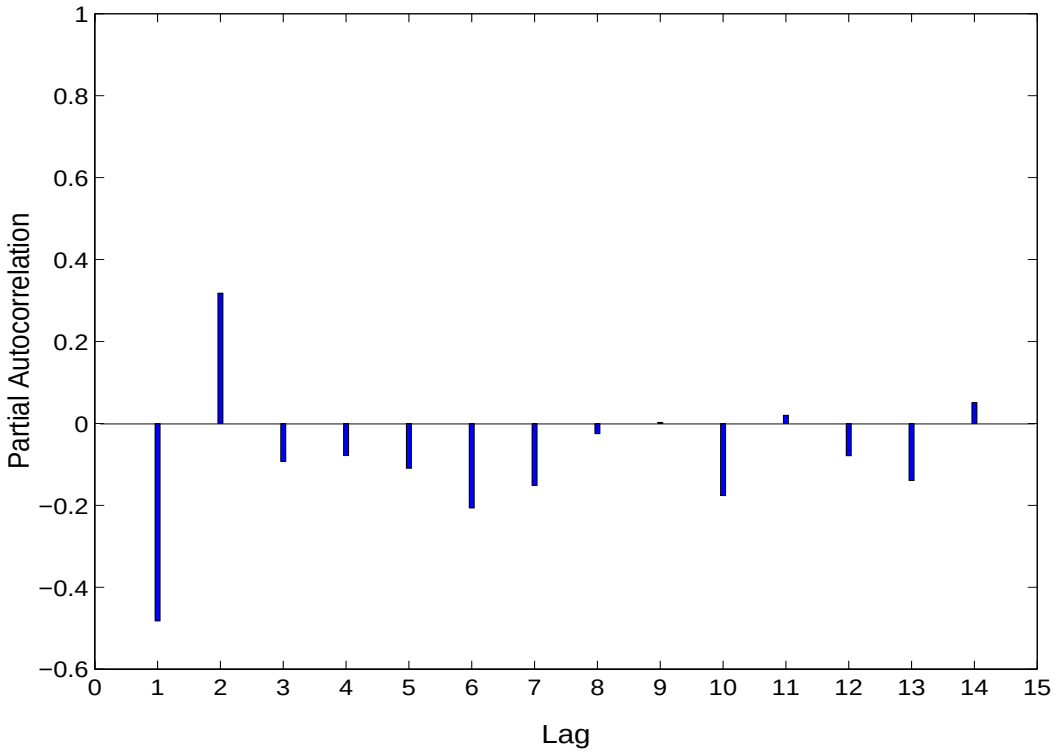

Figure 4.3: PACF of a Synthetic Series

Returning to the I/O example used earlier to illustrate the ACF, Figure 4.3 shows that the PACF is dominated by two large spikes in the first two lags. At lag 1, the partial autocorrelation is simply the autocorrelation. Its negative value suggests strong negative correlations between interarrival times that are adjacent to each other. Similarly, for interarrival times that are one lag apart (i.e., separated by a single intermediate observation), the partial autocorrelation at lag 2 remains significant even when the effect of the intermediate interarrival times have been accounted for. However, after lag 2, all remaining partial autocorrelations become insignificant.

In summary, the ACF and PACF of a time series have different patterns with different characteristics. The former measures the linear dependencies between observation pairs. The latter measures the partial contributions to the linear dependencies. ARIMA exploits these differences to identify a model structure for a series.

4.1.3 Pattern Characteristics in ACFs and PACFs of Stationary Processes

ARIMA uses broad characteristics in ACFs and PACFs to associate a time series with a particular type of process – autoregressive AR(p), moving average MA(q), or mixed ARMA(p,q). Table 4.1 summarizes the key properties in ACFs and PACFs to assist distinguishing the different model types graphically. It shows that most patterns are dominated by two behaviors: *exponential/sinusoidal decay* or *significant spikes with abrupt cutoff*.

Specifically, autoregressive AR(p) processes have ACFs that decay to zero exponentially or sinusoidally, while their PACFs have significant spikes cut off to zero abruptly. The number of spikes gives a value for p. Moving average MA(q) processes, on the other hand, have the opposite characteristics: ACFs with large spikes cut off after q lags, but PACFs rapidly decay to zero. Mixed ARMA(p,q) processes have decay patterns in both ACFs and PACFs.

Based on their experiments, Box and Jenkins observed that, in practice, the values for p, q are usually limited to $0, 1$, or 2 for most series. This knowledge provides a useful basis for matching patterns in estimated ACFs and PACFs with those of the five most common processes:

- AR(1), AR(2) – 1 and 2 spikes in PACFs respectively. Our example synthetic series, having two spikes in its PACF, can thus be modeled as an AR(2) process.
- MA(1), MA(2) – 1 and 2 spikes in ACFs respectively.
- ARMA(1,l) – exponential/sinusoidal decay pattern in both ACFs and PACFs.

4.1.4 Model Identification Methodology

The identification procedure in ARIMA includes four major steps. Step 1 determines the number of *regular* differencing levels required to expose the season length. Step 2 finds the *seasonal* differencing

Process	ACF	PACF
AR(p)	exponential or sinusoidal decay to zero	spikes cut off to zero after lag p
MA(q)	spikes cut off to zero after lag q	exponential or sinusoidal decay to zero
ARMA(p,q)	exponential or sinusoidal decay to zero	exponential or sinusoidal decay to zero

Table 4.1: Broad Characteristics in ACFs and PACFs of Stationary Processes

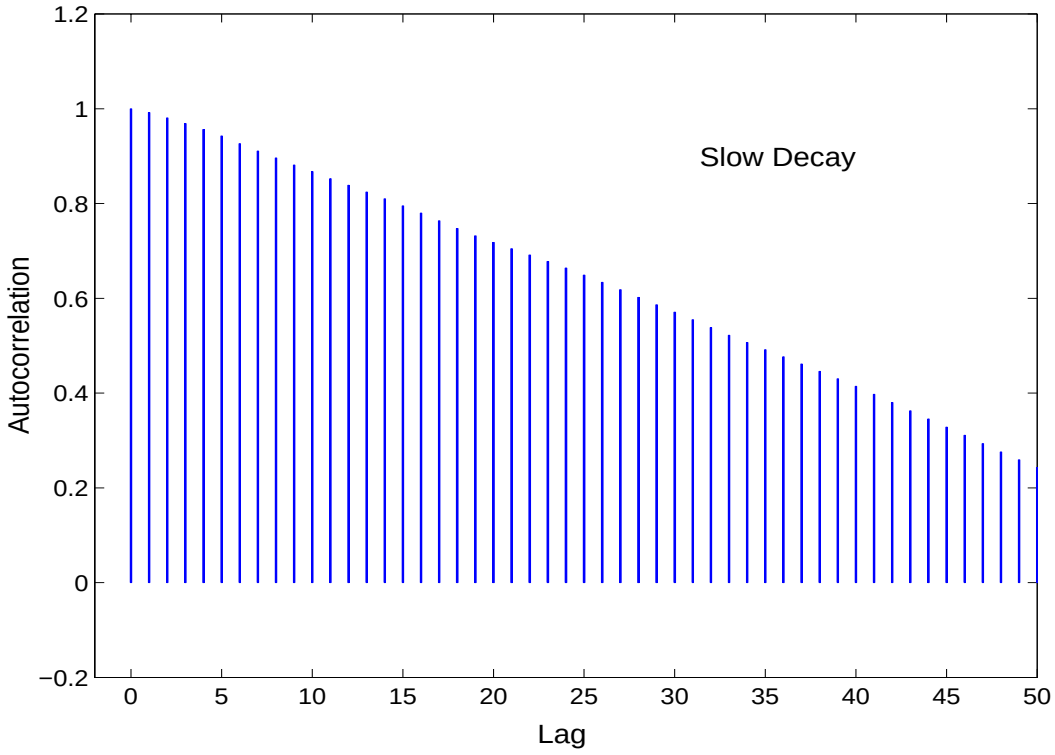

Figure 4.4: PRISM ACF

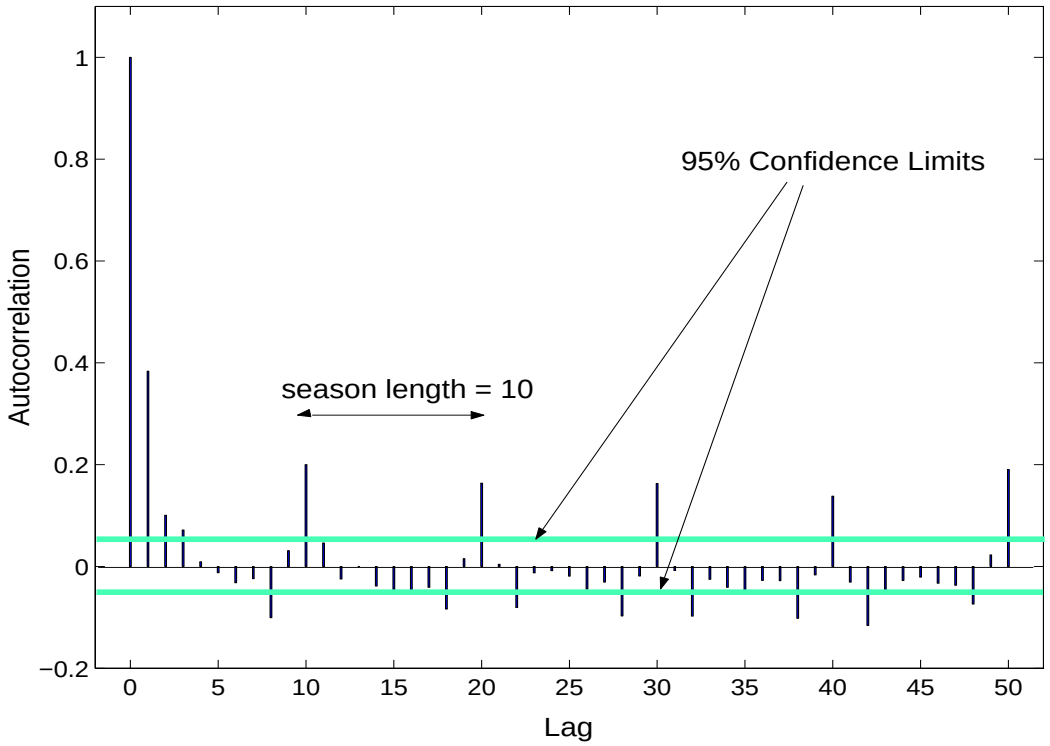

Figure 4.5: PRISM ACF after One Regular Differencing

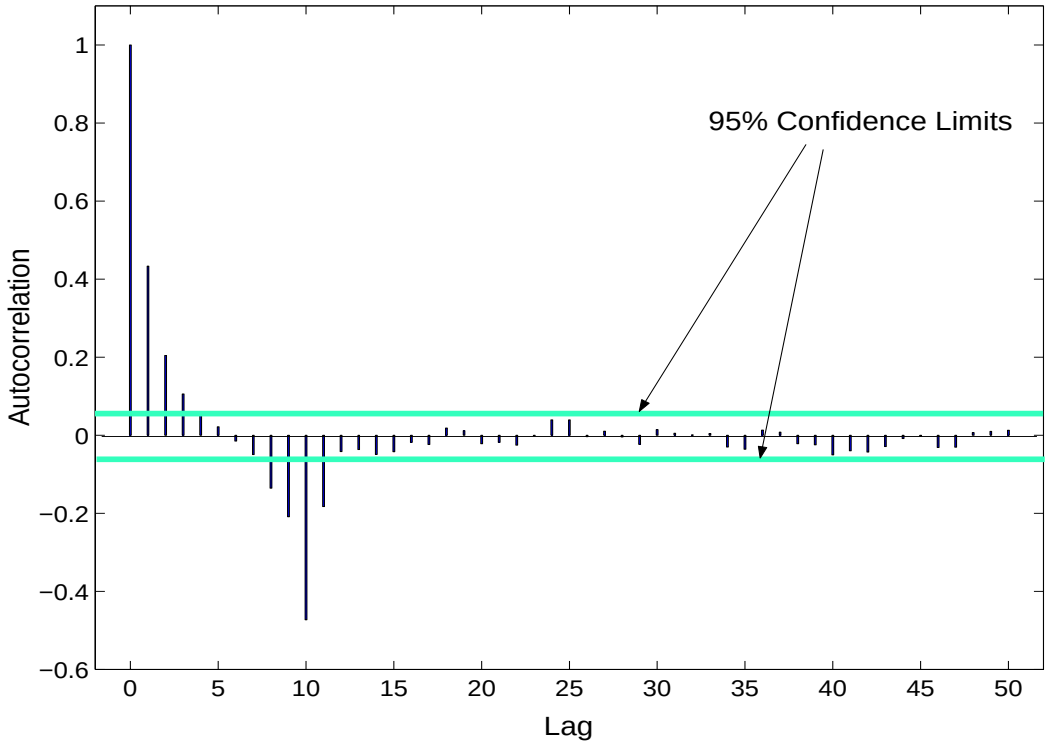

Figure 4.6: PRISM ACF after Seasonal Differencing

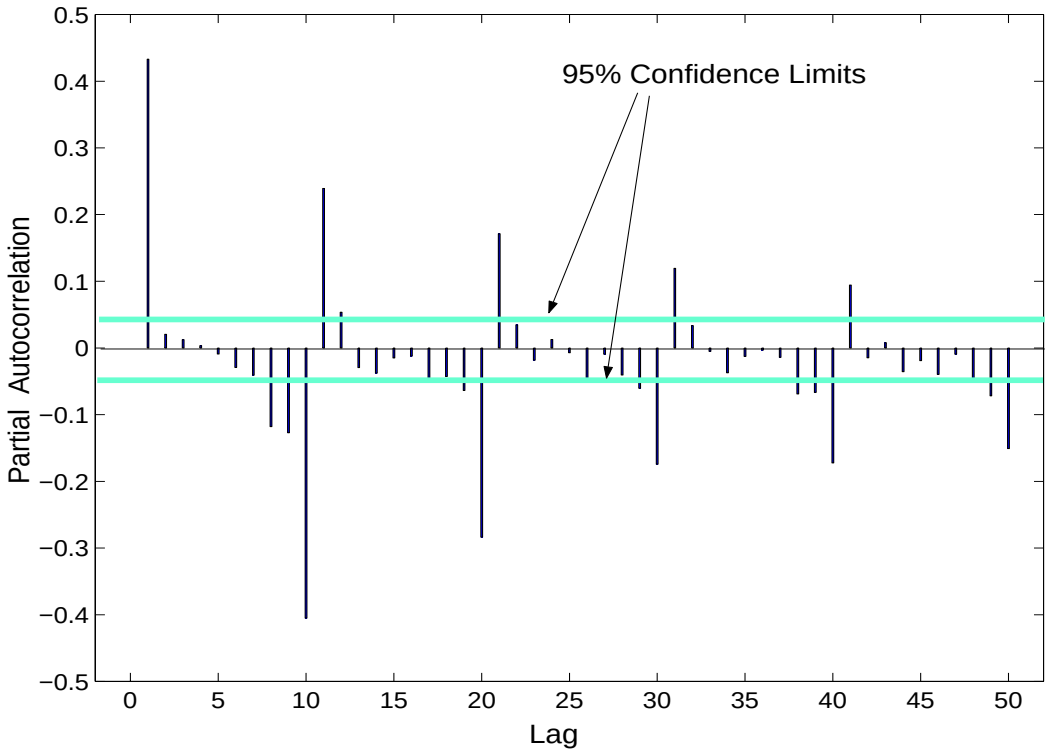

Figure 4.7: PRISM PACF after Seasonal Differencing

levels necessary to remove seasonality. Steps 3 and 4 estimate values for P, Q in the seasonal component, and p, q in the non-seasonal component respectively.

1. *Regular differencing to remove non-stationarity*: When the ACF of a series contains *slow decay patterns*, the series is differenced several times until there is no sign of slow decay. The number of times required, d (differencing levels), is often less than 2 [7]. Figure 4.4 plots the ACF of the PRISM read interarrival time series for the first fifty lags. It clearly exhibits very slow decay, extending far beyond the first ten lags, giving strong evidence of non-stationarity.

   If the series is seasonal, the ACF of the transformed series exhibits spikes periodically at the season boundaries. The distances between the spikes give the season length S.

   For the PRISM series, we focus on data after the first transition to discard transient values generated during program initialization. Figure 4.5 shows PRISM's ACF after having been differenced once. The horizontal band about zero represents approximately two times the estimated standard deviations or 95% confidence limits. Significant spikes extend above the band, repeating at regular intervals of ten requests, suggesting a season length of 10.

2. *Seasonal Differencing to remove seasonality*: The original series is differenced D times (often once is sufficient), using the newly found season length. The ACF of the transformation should be cleared of periodic patterns. In Figure 4.6, the estimated ACF of the seasonally differenced series is free of periodicity.

3. *Finding pattern for the seasonal component*: In the final stationary series, ACF and PACF patterns at the *season boundaries* would be broadly similar to the standard patterns summarized in Table 4.1. PRISM's ACF in Figure 4.6 has a large spike at lag 10 below the x axis, while its PACF decays to zero, alternating between lags 10, 20, etc. As a result, the seasonal component in PRISM can be modeled as an MA(1) process, with $P = 0$ and $Q = 1$.

4. *Finding pattern for the non-seasonal component*: The procedure used in Step 3 is applied to examining patterns at the beginning of ACFs and PACFs for the *lags within a season.* PRISM's ACF shows exponential decay, while its PACF displays a single spike at lag 1, suggesting an AR(1) process for its non-seasonal component, with $p = 1$ and $q = 0$.

Finally, combining results from the four steps above gives an ARIMA model $(p, d, q) \times (P, D, Q)_S$. For PRISM, the model is $(1, 0, 0) \times (0, 1, 1)_{10}$.

4.2 Automation of Autocorrelation Analysis

The model identification procedure we have just described is a visual, labor intensive process – it needs to be automated to serve a wide range of applications. Our objectives are to allow dynamic detection of correlation structures in application I/O interarrival times, without requiring in-depth knowledge of correlation analysis, or storing possibly obsolete models, especially after program modifications. Automatic characterization of I/O behaviors can generate more accurate predictions for I/O prefetching.

Our approach to automation targets a) identifying the significant correlations in ACFs and PACFs, using statistical confidence intervals, and b) extracting data patterns from these correlations. Our analysis uses samples of read interarrival times collected at run time, instead of the entire series from previous executions.

4.2.1 Identification of Significant Correlations

Intuitively, observations that are close together are more likely to be interdependent than those that are far apart, except at season boundaries. As a result, a stationary series' autocorrelations and partial autocorrelations are effectively reduced to zero at large lags. Barlett developed equations to approximate the standard deviations of ACF and PACF [3] for various lags of a stationary series of size N.

$$\hat{\sigma}[r_k] = \sqrt{\frac{1}{N} \times (1 \ + \ 2 * \sum_{j=1}^{k-1} r_j^2)} = \sqrt{\frac{1}{N} \times (1 \ + \ 2(r_1^2 + r_2^2 + \cdots + r_{k-1}^2))} \tag{4.1}$$

$$\hat{\sigma}[C_{kk}] = \sqrt{\frac{1}{N}} \tag{4.2}$$

Equation 4.1 estimates the standard deviation of ACF at lag k, involving all previously estimated autocorrelations. Its value eventually converges as autocorrelations become increasingly small at large lags. In contrast, the standard deviations for PACFs, estimated in Equation 4.2, are constant

High Frequency Locations	0	1	2	8	10	20	30	40	50	60	68	70	80	90
Level-1 Distances		1	1	6	2	10	10	10	10	10	8	2	10	10

Table 4.2: An Example of Level-1 Distances

and inversely proportional to the size of the series.

One can automatically isolate the significant correlations by selecting those that exceed the confidence intervals for ACFs and PACFs. Revisiting PRISM's ACF in Figure 4.5, the 95% confidence interval, about twice the largest standard deviation, is the horizontal band centered at zero.

4.2.2 Automatic Detection of Seasons Via ACF

To expose the existence of seasons, we developed two tests performed on the *locations (lag numbers)* of the significant correlations. These correlations, also known as *high frequency signals*, are automatically extracted from the ACF of the *regularly differenced* series.

- *The Level-1 (L-1) Distance Test*:

  We compute the distances between successive locations of the high frequencies. If there are several occurrences of the same distances, the most common distance will be chosen as the season length, according to a majority rule.

  To illustrate, Table 4.2 lists the first 14 locations of the high frequencies in PRISM's ACF, automatically detected using Barlett's 98% confidence limits. The ACF, shown in Figure 4.8, was computed on the differenced series of a sample of 700 read interarrival times, taken after the initialization period.

  There are seven occurrences of L-1 distances with length 10, two with length 2, one with length 6, and 1 with length 8. The high frequencies at the first few lags $(0, 1, 2)$ are due to the large correlations between read requests that are close to each other. All distances of length 1 are ignored as they cannot be associated with any season.

  As a result, the distance of 10 has a majority of 7/11 or 64%, and is chosen as the season length. However, if the majority test fails, we continue with the next test.

- *The Level-2 (L-2) Distance Test*:

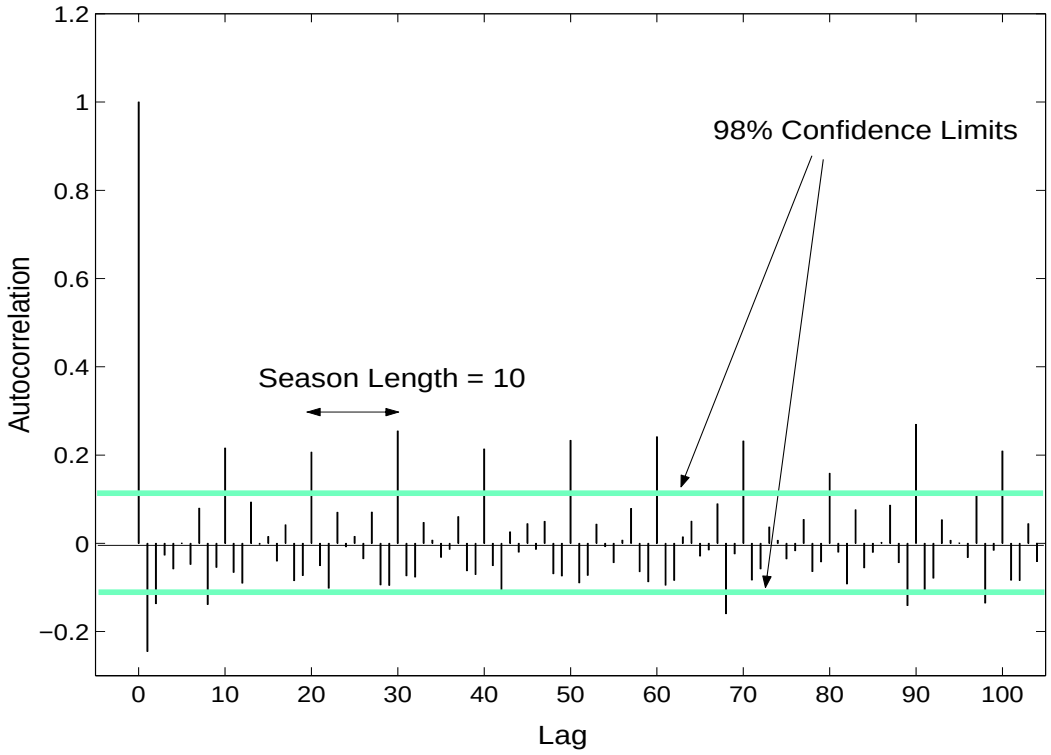

Figure 4.8: Season Detection Using PRISM ACF

While the L-1 distance test is used to detect *regular seasons*, the L-2 distance test is designed to detect *incremental seasons* (the length S of each season increases incrementally, e.g., $S = 1$, then 2, then 3 etc.). The season increments can be identified by simply taking the differences of the L-1 distances themselves. Again, a majority test is applied to the results to choose the most common occurrence.

For example, Table 4.3 shows a sequence of high frequency locations in the ACF of a synthetic series, after it has been differenced once. No duplicate values are found in the L-1 distances with the exception of 1. However, the L-2 distances are dominated by the value 15. The

High Frequency Locations	0	1	2	3	15	45	90	150	225	315	420
Level-1 Distances		1	1	1	12	30	45	60	75	90	105
Level-2 Distances			0	0	11	18	15	15	15	15	15

Table 4.3: An Example of Level-2 Distances

starting season length can be found from the L-1 distances, at the same position (5) as the first occurrence of the increment 15.

When both L-1 distance and L-2 distance tests fail, the series will be modeled as non-seasonal.

4.2.3 Automatic Differentiation of Decay Patterns in ACF and PACF

We observe that most patterns in ACF and PACF, including those in Table 4.1, fall into one of three categories: *slow decay* for non-stationary series, *exponential/sinusoidal* decay, and *abrupt* cutoff for stationary series. Sinusoidal patterns can be placed in the same category as exponential patterns because they can be transformed into exponential when only the magnitudes (absolute values) of the correlations matter.

4.2.3.1 Lag Count Test

If the correlation patterns are well-behaved – monotonically decreasing for slow or exponential decay – differentiating the three decay types can be achieved by simply tracking the *locations (lags) of the exposed high frequencies*. Below is a pseudo-code for the lag count test, where MIN_LAGS are usually chosen to be 2 and MAX_LAGS 10, based on the criteria of the five most common processes: $AR(1), AR(2), MA(1), MA(2)$ and $ARMA(1,1)$, in combination with our experiments with I/O traces.

```
if (  high frequencies are monotonically decreasing )
{
    if       (  high frequencies stay   within MIN_LAGS  )  then
                    pattern = abrupt cut-off
    else  if (  high frequencies extend beyond MAX_LAGS  )  then
                    pattern = slow decay
    else            pattern = exponential decay
}
```

However, difficulties arise when there are occasional outliers, often caused by sampling errors or side effects of large neighboring correlations. For example, in Figure 4.9, the overall pattern of the high frequency signals is exponential decay, with decreasing magnitudes, except for two outliers at lags 5 and 8.

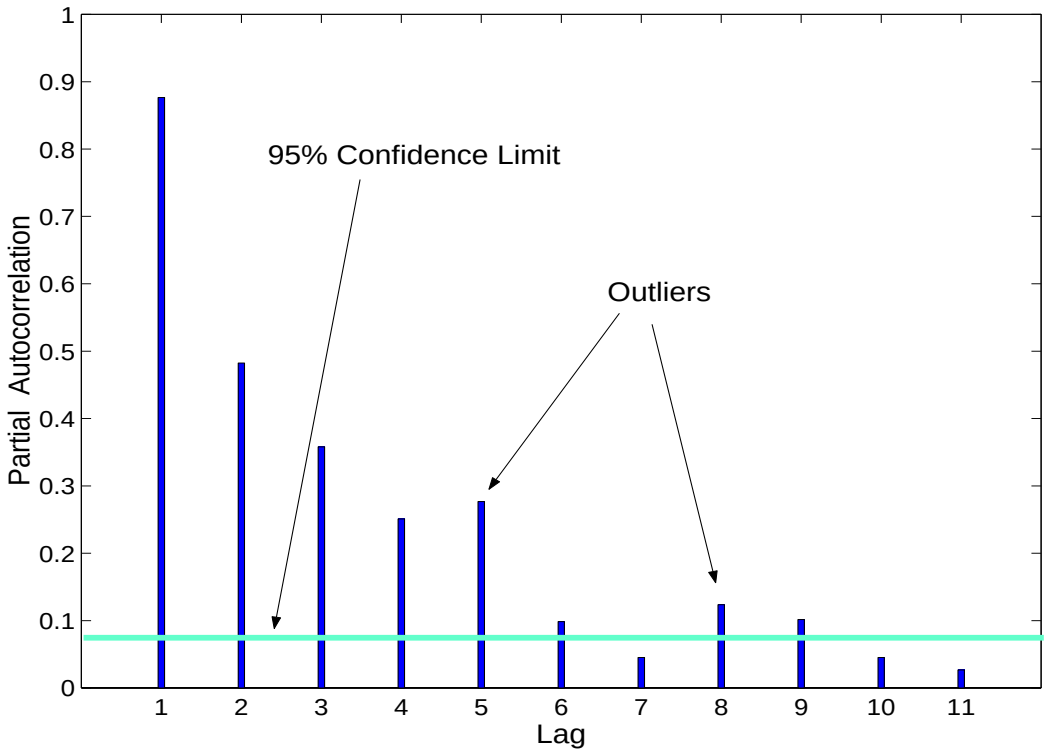

Figure 4.9: An Example PACF with Outliers

4.2.3.2 Average Rate of Change Test

To provide a more robust differentiation of decay patterns, we propose supplementing the lag count test with the *average rate of change (Δ) in the magnitudes of the high frequencies* over all lags k. The average rate of change offers more flexibility in capturing the overall pattern without being severely impeded by a few perturbations.

$$\Delta = \frac{1}{M}\sum_{k=0}^{M}\frac{|r_k| - |r_{k+1}|}{|r_k|} \qquad \text{M = number of the last high frequency lag}$$

Subsequently, Δ is compared against a set of thresholds to determine a particular decay pattern. What follows is an example test designed for I/O interarrival time series, with thresholds chosen based on our analyses of I/O behavior using I/O traces. Returning to our example PACF (Figure 4.9), the average decay rate is around 21%.

```
if        ( Δ > 65% )    then  pattern = abrupt cutoff
else if   ( Δ < 10% )    then  pattern = slow decay
else                           pattern = exponential decay
```

4.2.3.3 Broad Characteristics Matching

Finally, combining results from the lag count and average rate of change tests, the pseudo-code below uses the broad characteristics listed in Table 4.1 to make decisions about the model structure. When both ACF and PACF have exponential decay patterns, we choose the simple ARMA(1,1) model to avoid over-parameterization, following the criterion of the 5 most common processes.

```
if  (  high frequencies exist in ACF or PACF )
{
     if       ( ACF pattern  = abrupt cut-off at lag q )          then
                   model = pure MA(q)
     else if  ( PACF pattern = abrupt cut-off at lag p )          then
                   model = pure AR(p)
     else if  ( both ACF and PACF patterns = exponential decay )  then
                   model = mixed ARMA(1,1)
}
else
   // data has no correlation
   p = q = 0;
```

Occasionally, regular or seasonal differencing can *decorrelate* data completely, making all correlations in the non-seasonal or seasonal component of the final transformed series insignificant. When this situation occurs, $p = q = 0$.

4.2.4 Implementation

Our approach to automatic identification of an ARIMA model focuses on a) automatic extraction of significant correlations and b) automatic pattern recognition. Techniques developed earlier for

these two key ideas form the core of our implementation of a model identification system. The schematic of our implementation, illustrated in Figure 4.10, includes three key components:

1. *The season detector via ACF* searches for seasonal autocorrelations in I/O interarrival times. It uses the level-1 and level-2 distance tests to determine the existence of seasons and their lengths.

2. *The model identifier for non-seasonal series* must first check for slow decay patterns. If the series is non-stationary (slow decay exists), regular differencing transformation is applied. Then the lag count test and average rate of change test are executed on the ACF and PACF of the transformed series to find suitable structures (p, d, q).

3. *The model identifier for seasonal series* distinguishes fixed-length from incremental-length seasons. The *fixed season model identifier* analyzes ACFs and PACFs to approximate structures for the season component $(P, D, Q)_S$ and the non-seasonal component (p, d, q). On the other hand, the *incremental season mediator*, guided by the season length and increment, splits the series along season boundaries into a major and a minor series, which are separately identified. In the next section, we will describe seasonal model identification in more detail.

4.2.5 Identification of Model Structures for Seasonal Series

4.2.5.1 Fixed Season Model Identifier

When the detected season length is fixed, the interarrival time series is seasonally differenced with the season length to remove all periodic signals. High frequencies in the ACF and PACF of this differenced series are analyzed via the lag count test, the average rate of change test, and the broad characteristics matching procedure. The key issue is to *decide where these tests should be applied*:

- *For the non-seasonal component*, only the high frequencies *within the first season* are analyzed, excluding those that are near the season boundaries because side effects from seasons can inflate autocorrelations. The identification procedure is similar to that for non-seasonal series, described in §4.2.4, except that it operates on *selected portions* as opposed to the *entire* series.

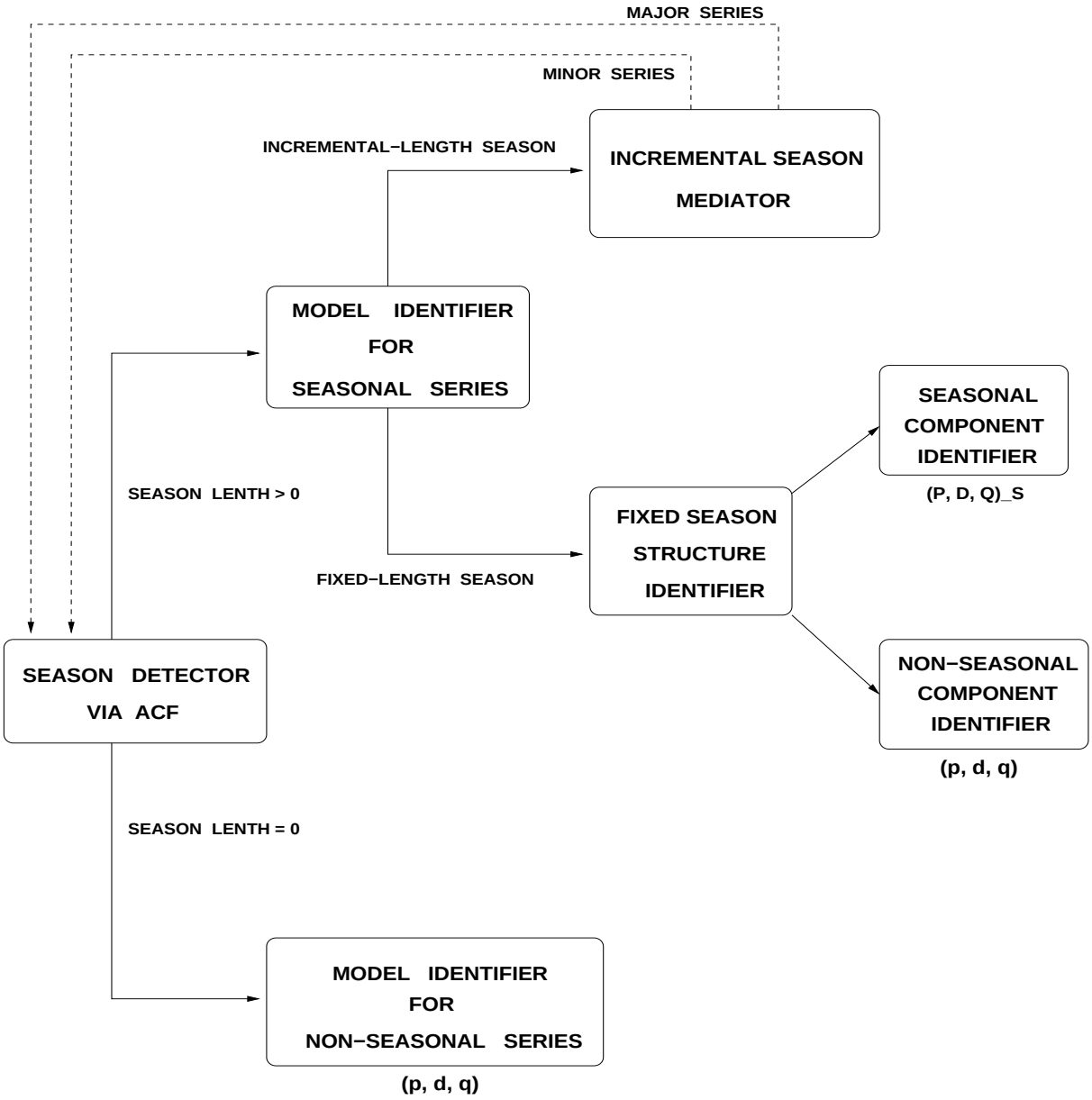

Figure 4.10: Automatic Model Identification Architecture

- *For the seasonal component*, identification is based on examining high frequencies located at the *season boundaries*, i.e., at lags S, $2S$, $3S$, etc.

Below, we illustrate the identification procedure applied to the PRISM series, for which a season length of 10 was automatically detected (see §4.2.2). Tests were performed on the absolute values of the ACF and PACF of the seasonally differenced series. Only one differencing transformation ($D = 1$) was required to clear away *both* non-stationary and seasonal behavior.

- *Seasonal component:* Autocorrelations at lags associated with the *season boundaries* (10, 20, 30, $\cdots$) were culled from the ACF and PACF of the PRISM seasonally differenced series. In the extracted PACF shown on Figure 4.12, the presence of an outlier at lag 80 disrupted the monotonically decreasing behavior and made the lag count test fail. However, the average rate of change test successfully identified an exponential decay pattern with an average decay rate of 32%.

  The abrupt cut-off at lag 10 of the extracted ACF in Figure 4.11 can be promptly identified by the lag count test. This cutoff matches the broad characteristics of a pure moving average MA(1) process, yielding an estimated structure $(P, D, Q)_S = (0, 1, 1)_{10}$.

- *Non-seasonal component:* Automodeler performed the lag count test on the ACF and PACF at lags located *within the first season* (as opposed to season boundaries). The test successfully recognized an exponential decay pattern from the four strictly decreasing, significant autocorrelations shown in Figure 4.13. It was also successful in detecting an abrupt cut-off pattern in the PACF, marked by a single spike at lag 1 in Figure 4.14. The combined patterns match those of a pure autoregressive AR(1) process. Consequently, the model's non-seasonal component is estimated to have a structure $(p, d, q) = (1, 0, 0)$.

4.2.5.2 Incremental Season Mediator

The identification techniques designed for series with *fixed-length seasons* are also applicable to incremental season series when the latter are modeled via two subseries: the major series has all the interarrival times that occur *at the season boundaries*, whereas the minor series includes all the interarrival times *within the seasons*. The seasons are increased incrementally, starting from the

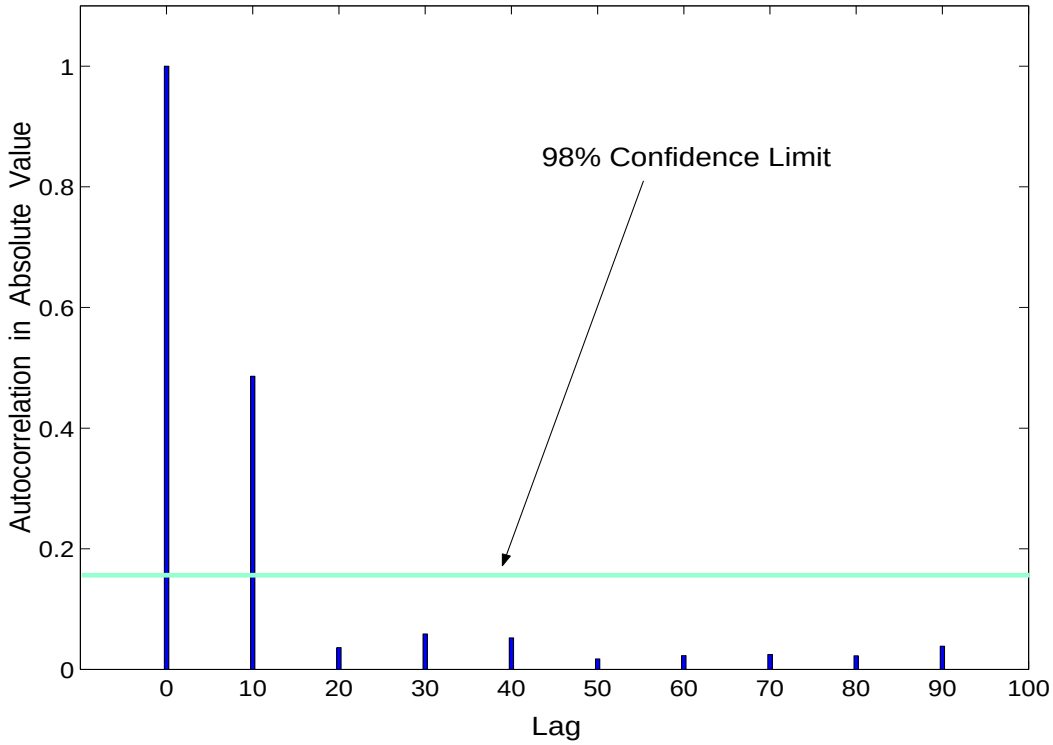

Figure 4.11: PRISM ACF Selected at Season Boundaries

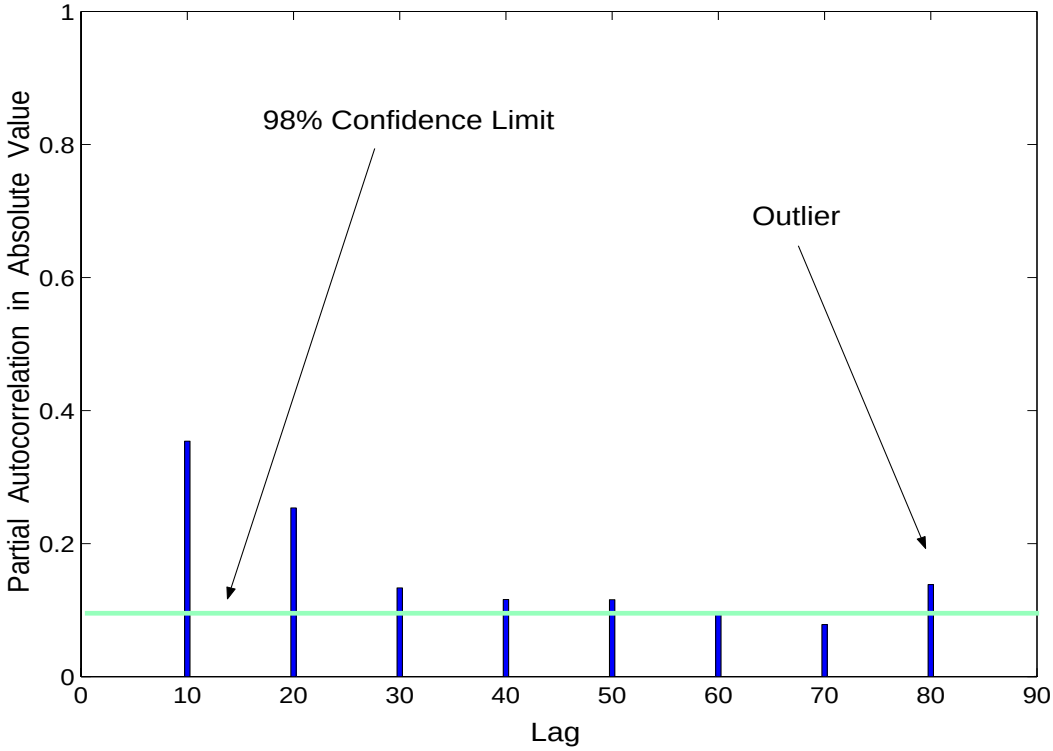

Figure 4.12: PRISM PACF Selected at Season Boundaries

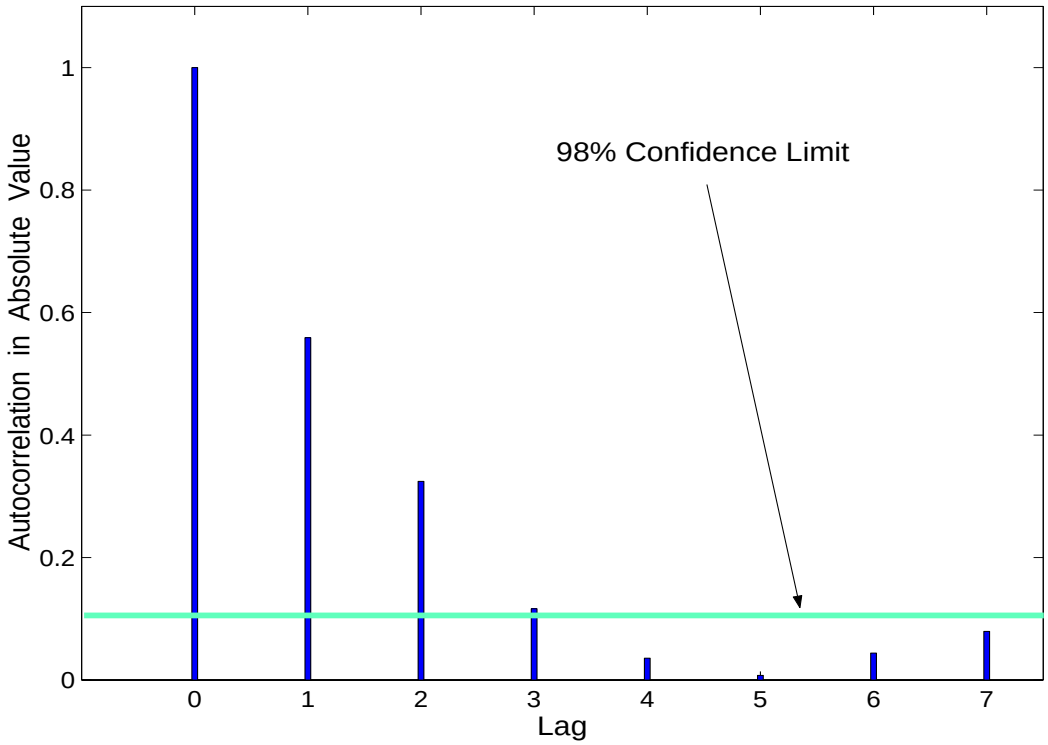

Figure 4.13: PRISM ACF within the First Season

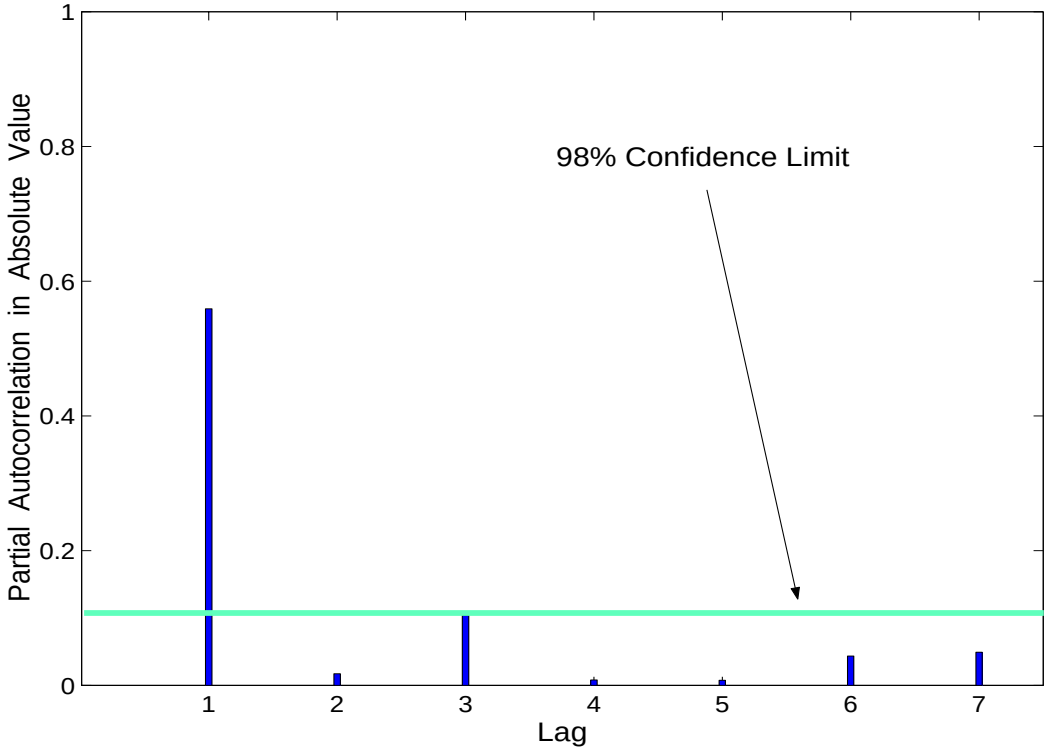

Figure 4.14: PRISM PACF within the First Season

length S, with an increment detected by the L-2 distance test. For example, if the increment is 2, then the boundary locations would be S, $S+2S = 3S$, $3S+2S = 5S$, etc. For I/O interarrival times in scientific codes, the major series is usually non-seasonal, while the minor series has fixed-length seasons.

Later, in the chapter on validating Automodeler, we will examine further experiments with different I/O behaviors to verify our proposed methods for automatic identification of ARIMA models.

4.3 Summary

In summary, we have designed a model identification system that can automatically estimate an ARIMA structure to approximate the significant correlations in I/O interarrival times. We explore several pattern recognition techniques to isolate the broad characteristics in I/O behavior. This characterization, based on the five most common ARIMA processes, is aimed at simplifying the identification process, trading accuracy for overhead reduction.

In I/O prefetching, we believe that models for read interarrival times do not have to be very accurate, as long as the overall behavior and the associated correlations are captured. For example, there is no large difference between 300 and 400 microseconds for the predicted interarrival times when we are using one-second windows to perform multiple-block prefetching.

Chapter 5

Automatic Detection of Abrupt Phase Transitions via Haar Wavelet

Detection of abrupt phase transitions can provide essential information about possible access pattern changes, a potential cause of inaccurate predictions. Although autocorrelation analysis provides extremely valuable tools for identifying important stationary, non-stationary, and periodic access patterns, it cannot efficiently identify occasional, irregular phase transitions. *Interferences among transitions and seasonal dependencies can perturb autocorrelation patterns*, making them very difficult to recognize.

For this reason, we investigate the application of wavelet transforms to efficiently detect phase transitions and rapidly identify seasons in bursty I/O access patterns. Wavelet transforms can be computed in linear time [56], minimizing identification overhead.

We begin with a brief introduction to the Haar wavelet in §5.1. Then we explain the motivation on using wavelet detail coefficients to identify high frequencies in a time series. Section §5.2 presents the Jackknife method that allows confidence intervals to be computed for the mean of the detail coefficients. Finally, in §5.3 and §5.4, we describe experiments to validate our proposed techniques.

5.1 Haar Wavelet

From the perspective of time series, the Haar wavelet transform decomposes a series into a set of averages and differences, known as average coefficients and detail coefficients respectively. The decomposition is based on the simple averaging and differencing operations below, performed on non-overlapping pairs of consecutive observations [58].

Resolution	Average Coefficients	Detail Coefficients
8	10 12 11 7 9 11 8 12	
4	11 9 10 10	-2 4 -2 -4
2	10 10	-2 0
1	10	0

Table 5.1: Haar Wavelet Decomposition into Average and Detail Coefficients

$$\begin{aligned} \text{average coefficients} \quad c_i &= \frac{y_i + y_{i+1}}{2} \\ \text{detail coefficients} \quad d_i &= y_i - y_{i+1} \qquad \text{for} \quad i = 1,\ 3,\ 5\ \cdots \end{aligned}$$

The *averaging operation acts as a lowpass filter*, smoothing signals in a series and removing high frequency signals. Averaging approximates signals' general behavior. Conversely, the *differencing operation acts as a highpass filter*, exposing high frequency signals in a series and removing the averages [57, 56].

Table 5.1 illustrates the Haar wavelet transformations applied to a series of 8 observations. The highest resolution, 8, corresponds to the original series with 8 elements. Pairwise averaging and differencing result in 4 averages and 4 detail coefficients in the next resolution, 4. Signals at the higher resolution can be easily reconstructed from the coefficients when we observe:

$$\begin{aligned} 2c_i + d_i &= 2y_i \\ y_i &= \frac{2c_i + d_i}{2} \end{aligned}$$

Recursively decomposing the coefficients at each lower resolution eventually leads to the single average coefficient 10, which represents the overall average (DC component) of the series. The final wavelet transform consists of this average and all detail coefficients at all resolutions:

$$[\ 10,\ \ 0,\ \ -2,\ \ 0,\ \ -2,\ \ 4,\ \ -2,\ \ -4\ ]$$

If perfect signal reconstruction is not required, one can take advantage of these recursive transformations to compress data at different resolutions – discarding the detail coefficients that are small relative to the overall average. Each lower resolution reduces data volume from previous resolution by half.

Returning to our example, at resolution 4, if -2 is deemed to be small and thus discarded, the signal can be compressed to 3 values: [10, 4, -4]. Further compression is possible when one settles for even lower resolutions. Hence, at resolutions 1 and 2, the Haar transform only contains the average coefficient, 10, when the detail coefficient, -2, is ignored.

When interarrival times in a series do not have large variations, the detail coefficients tend to be small. One can use detail coefficients to reveal correlations between neighbors in a series. If the correlations are high, then y_i and y_{i+1} would have similar values and their differences would be small. Conversely, if the neighbors are only weakly correlated, their differences would be large, creating sharp impulses among the relatively small differences in the correlated pairs.

Abrupt changes caused by bursty I/O (e.g. ESCAT) and phase transitions (e.g. PRISM) disrupt regular I/O behavior and create sharp edges at the boundaries. These sharp edges manifest in the detail coefficients with very large magnitudes amidst numerous small coefficients. Occurrences of these large coefficients are periodic in bursty I/O patterns, but only occasional in phase transitions.

Below is a synthetic series that illustrates a phase transition. The series is bi-modal – the first 9 observations hover around 23000 microseconds, but the remainding 9 around 5300 microseconds. $\lfloor log_2(9+9) \rfloor$ gives 4 possible resolutions, hence 4 differencing levels. However, only one differencing level is sufficient to disclose a single, large detail coefficient. All other coefficients are much smaller. The observation associated with the large coefficient marks the beginning of a phase transition, changing from a higher to a lower mean.

SERIES $\longrightarrow$ 23244, 23203, 23080, 23096, 22964, 23010, 23025, 22994, **240048**,
5244, 5322, 5502, 5261, 5086, 5225, 5352, 5301, 5476

	23244	23080	22964	23025	240048	5322	5261	5225	5301
–	23203	23096	23010	22994	5244	5502	5086	5352	5476
	41	-16	-46	31	234804	-180	175	-127	-175

Table 5.2: Haar Detail Coefficients after One Differencing Level

5.2 Confidence Interval Computations via Jackknife

Constructing a *confidence interval for the mean of the detail coefficients* is challenging: how do we express our confidence in the results for the detected phase transitions when we do not know the distribution of the detail coefficients to estimate their variances.

The Jackknife method, so named because of its universality, is a statistical technique that allows estimating confidence intervals for the parameters of any given sample with an unknown distribution [67]. The rationale is to *split the sample into subsamples in such a way as to produce randomization effects* (i.e., as if the subsamples were drawn from randomized experiments so they can be considered as independent random samples). The goal is to have *many subsamples, each with a large number of observations*, such that the subsamples are spread far enough apart to minimize statistical dependencies.

Below we illustrate the Jackknife procedure, applied to approximating a confidence interval for the mean of our sample of 9 wavelet detail coefficients.

41 -16 -46 31 234804 -180 175 -127 -175

5.2.1 The Jackknife Procedure

- Compute the *overall mean* $\bar{D} = 26056.33$.
- To expose variations within the sample, *divide it into subsamples*, each time shifting forward a window of 8 (sample size - 1) coefficients, starting from the second coefficient and wrapping around the sample until we have all 9 entries as shown in Table 5.3. This technique avoids arbitrary splitting of the sample while maintaining a sufficiently large subsample size.
- Because the subsamples above are extracted from coefficients that are close to each other, they need to be spread out far apart before they can be considered independent. This spreading effect can be induced by *magnifying* 8 *times the deviations* $(\bar{D} - \bar{D}_i)$ of the subsample means $\bar{D}_i$ from the overall mean $\bar{D}$ to give the adjusted subsample means $\bar{D}'_i$:

Subsample	Detail Coefficient								Subsample Mean
D_1	-16	-46	31	234804	-180	175	-127	-175	$\bar{D}_1 = 29308.25$
D_2	-46	31	234804	-180	175	-127	-175	41	$\bar{D}_2 = 29315.38$
D_3	31	234804	-180	175	-127	-175	41	-16	$\bar{D}_3 = 29319.12$
				...					
D_9	41	-16	-46	31	234804	-180	175	-127	$\bar{D}_9 = 29335.25$

Table 5.3: Systematic Splitting of Detail Coefficients into SubSamples

$$\begin{aligned}
\bar{D}_1' &= \bar{D} + 8 \times (\bar{D} - \bar{D}_1) = 26056.33 + [8 \times (26056.33 - 29308.25)] = 40.97 \\
\bar{D}_2' &= \bar{D} + 8 \times (\bar{D} - \bar{D}_2) = 26056.33 + [8 \times (26056.33 - 29315.38)] = -16.07 \\
&\cdots \\
\bar{D}_9' &= \bar{D} + 8 \times (\bar{D} - \bar{D}_9) = 26056.33 + [8 \times (26056.33 - 29335.25)] = -175.03
\end{aligned}$$

- Finally, the mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$ of the adjusted subsample means are used to compute the confidence interval.

$$\hat{\mu} = \frac{\sum_{i=1}^{9} \bar{D}_i{}'}{9} = 26056.33 \qquad \hat{\sigma} = \sqrt{\frac{1}{8} \sum_{i=1}^{9} \left(\bar{D}_i{}' - \hat{\mu}\right)^2} = 78280.45$$

The 95% confidence interval based on the 9 subsamples (i.e., 8 degrees of freedom) is:

$$CI = \hat{\mu} \pm 2.31 \frac{\hat{\sigma}}{\sqrt{(9)}} = [-34219.62, \quad 86332.29]$$

All detail coefficients in the sample fall within the confidence interval, except 234804 which is about 9 times the value of $\frac{\hat{\sigma}}{\sqrt{(9)}} = 26093.48$. For this reason, one can statistically infer, with 95% confidence, that the 234804 is highly likely to be at a transition point.

5.2.2 Pseudo-Jackknife

If the sample is long, the Jackknife method may be computationally intensive. A less sophisticated, but also less expensive variant is to simply divide the samples into fewer subsamples and continue the jackknife procedure without adding the adjustments [67].

For our example, the pseudo-Jackknife computations are shown in Table 5.4 using 3 non-overlapping subsamples. The mean and standard deviation of the subsample means are $\hat{\mu} = 26056.33$ and $\hat{\sigma} = 45173.62$. The 95% confidence interval is $[-86091.97,\ 138240.64]$, wider than that obtained via pure Jackknife, $[-34219.62,\ 86332.29]$, but still useful. The high frequency of 234804 is about 9 times larger than the standard deviation of the detail coefficients' mean, $\frac{\hat{\sigma}}{\sqrt{3}} = 26081$.

Subsample	Detail Coefficient			Mean
D_1	41	-16	-46	$\bar{D}_1$ = -7
D_2	31	234804	-180	$\bar{D}_2$ = 78218.33
D_3	175	-127	-175	$\bar{D}_3$ = -42.33

Table 5.4: Pseudo-Jackknife Computations

5.3 Detection of Phase Transitions in PRISM

To demonstrate the ability of the Haar wavelet decomposition in detecting phase transitions, we conducted experiments on the PRISM read interarrival time series. Each sample contains 300 observations. Our approach includes the following major steps:

- Haar transform the sample to obtain the detail coefficients.
- Use the Jackknife method to estimate the confidence interval for the mean of the detail coefficients. Filter those coefficients that exceed the confidence interval. Because we are only searching for high frequency components, we can simplify computations by using the absolute values of the coefficients to compare with the upper confidence limit.
- Execute the L-1 and L-2 tests on the filtered coefficients to detect periodicity. Successful tests indicate that these high frequencies correspond to season boundaries rather than phase

transitions.

- If the L-1 and L-2 tests fail, we locate the coefficient with the maximum magnitude and compute the means of the subsamples on both sides of this maximum. If these local means are substantially different, e.g., one is at least twice as large as the other, then the maximum is most likely to be a phase transition, because of the change in overall behavior before and after it.

5.3.1 PRISM Phase Transition I

Figure 5.1 plots the detail coefficients for the first 300 observations. The total number of coefficients, 150, is half of the sample size because of the pairwise Haar differencing transformations. The upper confidence limit of around 8 was computed via the pseudo-Jackknife method to reduce processing costs. Those spikes that extend beyond the confidence limit correspond to PRISM's high frequency signals. The spike with a maximum coefficient of 39.6 in the top graph represents the abrupt edge at observation 88 in the bottom graph. The sample means on the left and right side of this edge are approximately 319 and 55 respectively – a ratio of 5.8 to 1, pinpointing an increase in the arrival rate of PRISM's read requests after the transition.

5.3.2 PRISM Phase Transition II

The second phase transition manifested as a series of abrupt changes, where PRISM interarrival times increase in a stepwise manner, from approximately 55 to 100, then to 350, and finally to around 400 (see Figure 5.4). These changes result in a sequence of spikes in the detail coefficients as shown in Figure 5.3. The largest spike, with a magnitude of 30.4, is associated with the longest abrupt edge at observation 2077. The subsequent smaller increases induce smaller spikes. The mean ($\simeq 395$) of the series data, located on the right of the maximum spike, is approximately 6.7 times that on the left ($\simeq 59$), indicating a slowdown in PRISM's read request arrivals after the transition.

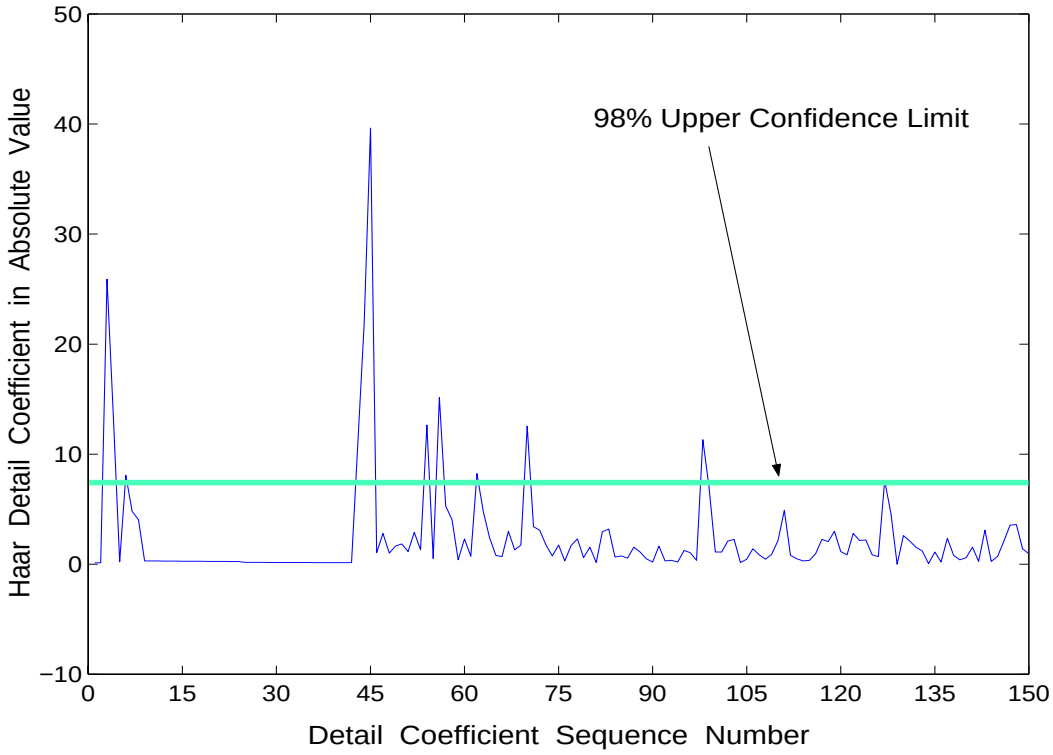

Figure 5.1: Haar Detail Coefficients for PRISM – Phase Transition I

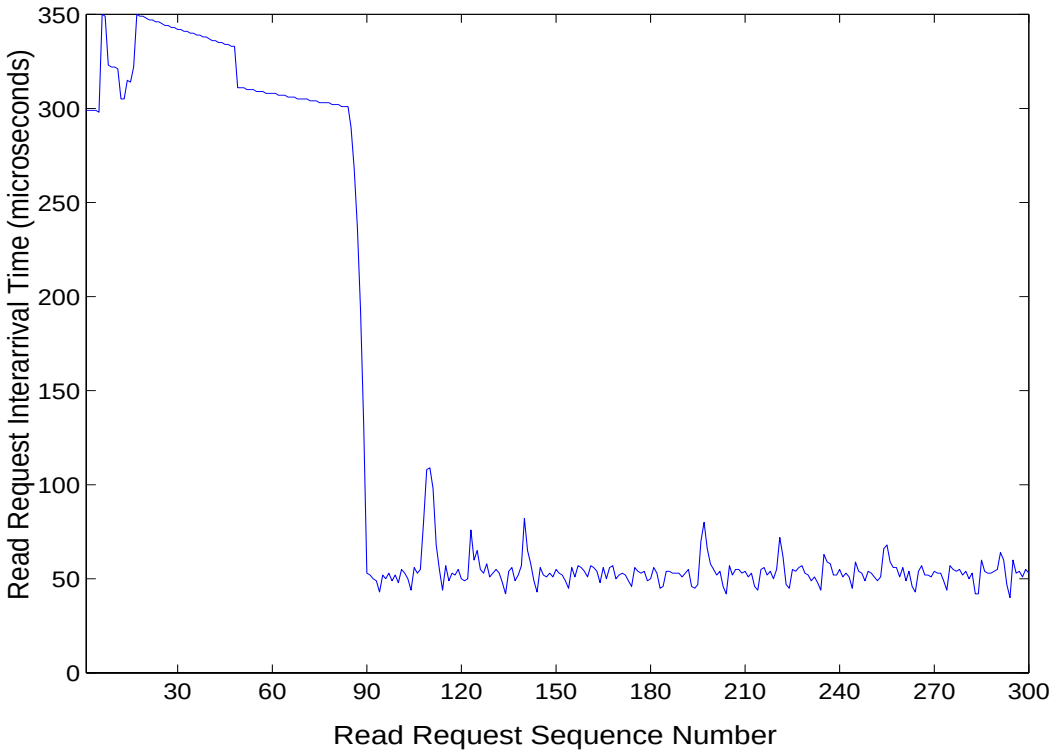

Figure 5.2: PRISM Series – Phase Transition I

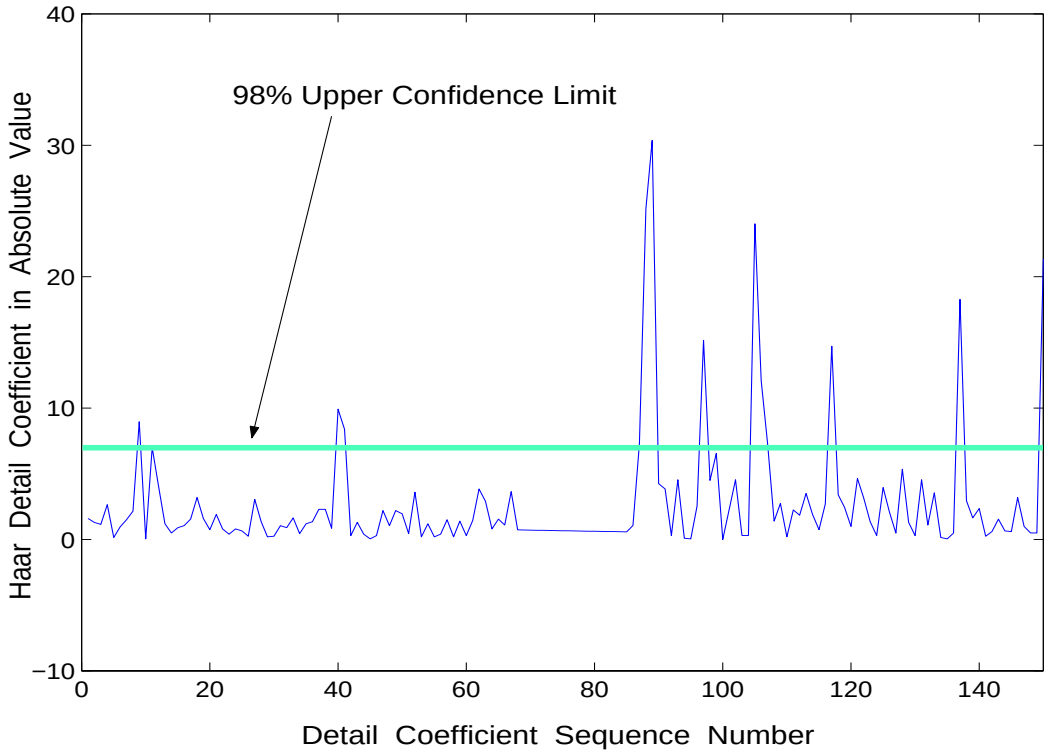

Figure 5.3: Haar Detail Coefficients for PRISM – Phase Transition II

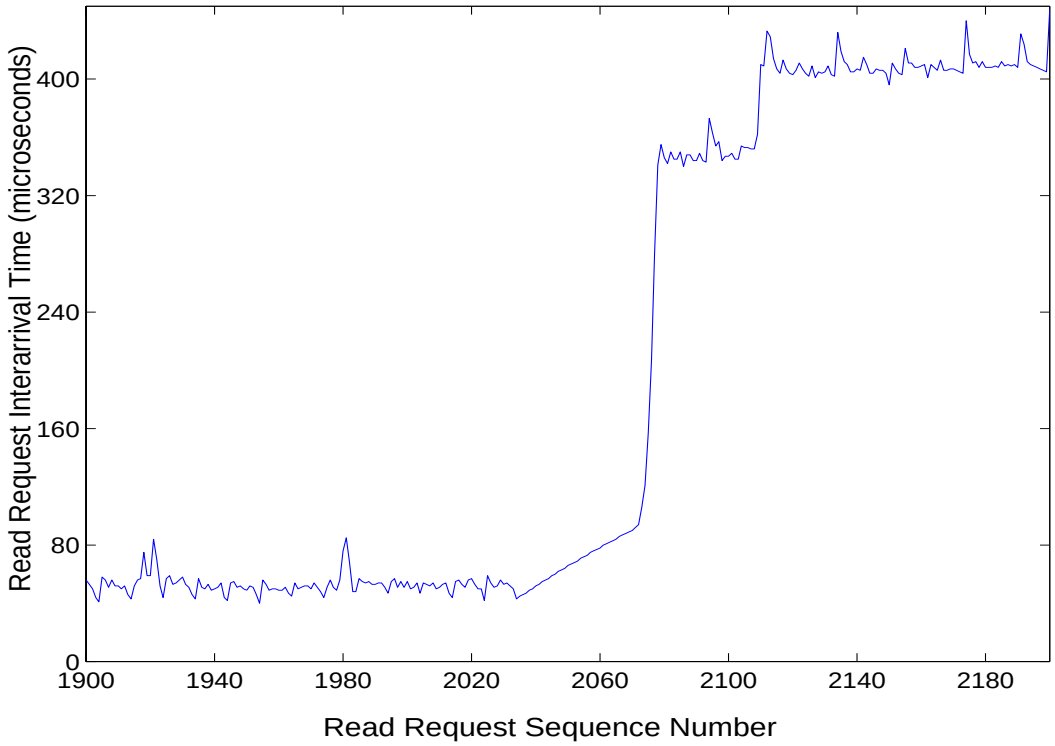

Figure 5.4: PRISM Series – Phase Transition II

5.4 Detection of Seasonal patterns in ESCAT

As noted earlier, the ESCAT read interarrival time series is characterized by bursts of short read arrivals alternating with long computation intervals. This behavior results in periodic patterns separated by very sharp edges that reflect abrupt changes.

To show the ability of the Haar wavelet transform to identify such bursty and seasonal behavior, we extended Automodeler to include a module that uses wavelets to detect seasonal patterns. In our implementation, outlined in Figure 5.5, the wavelet technique was chosen as the preferred method for season identification over the autocorrelation technique because it is less computationally intensive. Only when it fails is season detection via autocorrelation invoked.

Figure 5.6 plots the 350 Haar detail coefficients for the first 700 observations of ESCAT. The sharp edges in ESCAT, illustrated in Figure 5.7, are transformed into spikes, spaced regularly across the detail coefficient plot. Their magnitudes are much larger than the upper confidence interval of 4600, as shown in Figure 5.8.

The L-1 distance test performed on the spikes succeeded in identifying a season length of 64, which is equivalent to 128 (64×2) observations. The identification process through Haar wavelet is fast: it only takes linear time as opposed to quadratic time required by autocorrelation analysis.

5.5 Summary

In this chapter, we presented online techniques to detect abrupt phase transitions in I/O interarrival times. Such detection is important because it pinpoints an imminent potential for I/O behavior change.

We decomposed a given series via the Haar wavelet transform to obtain its detail coefficients. These coefficients expose high frequency signals while remove averages. We then explored the Jackknife method to compute a confidence interval for the coefficients' mean. High frequencies that exceed the confidence interval are used to establish the existence of seasons and phase transitions. To reduce computation overhead for long series, we investigated the pseudo-Jackknife method which yields a wider, but still useful, confidence interval.

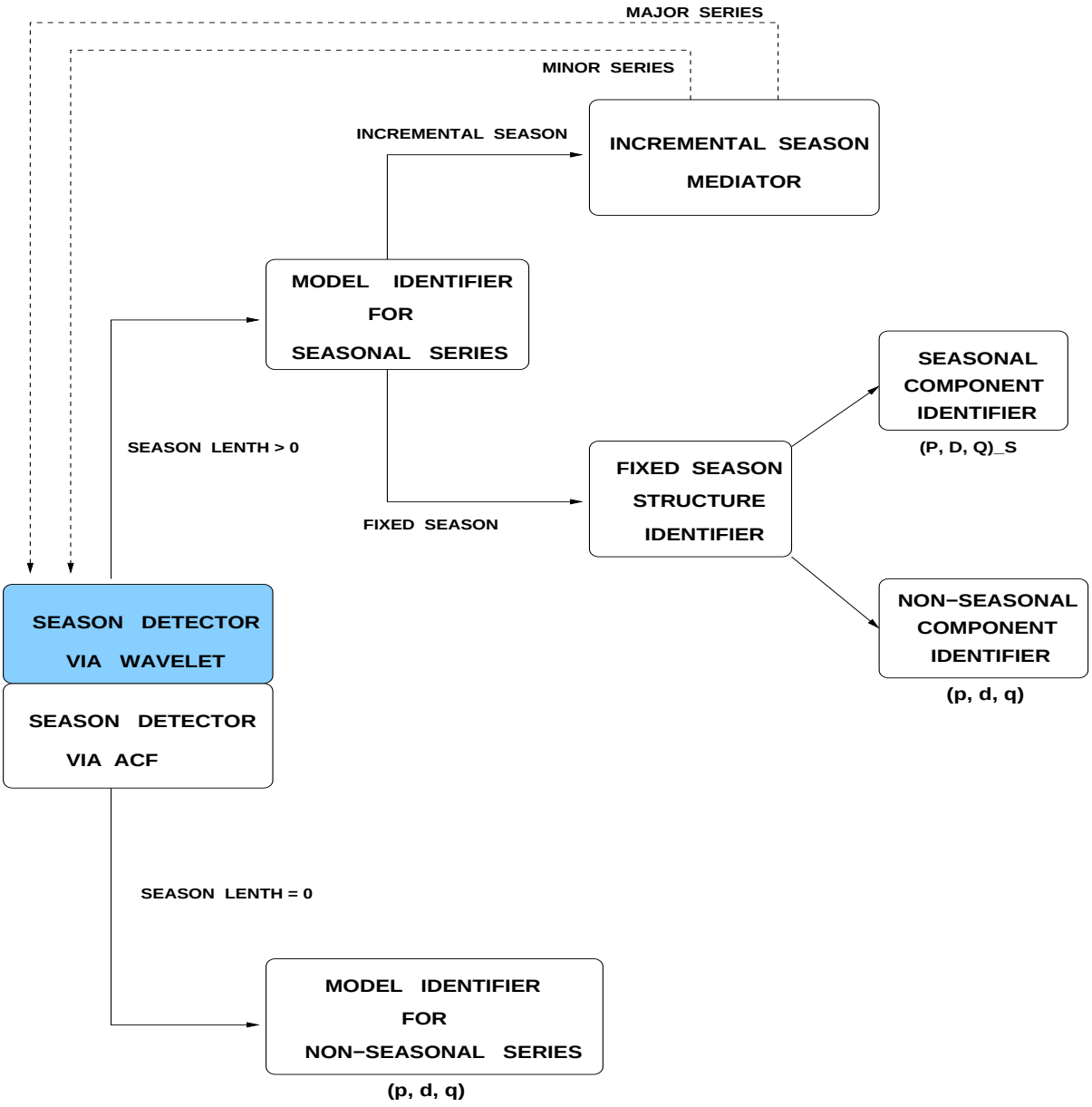

Figure 5.5: Automatic Model Identification Architecture Extended with Wavelet Transform

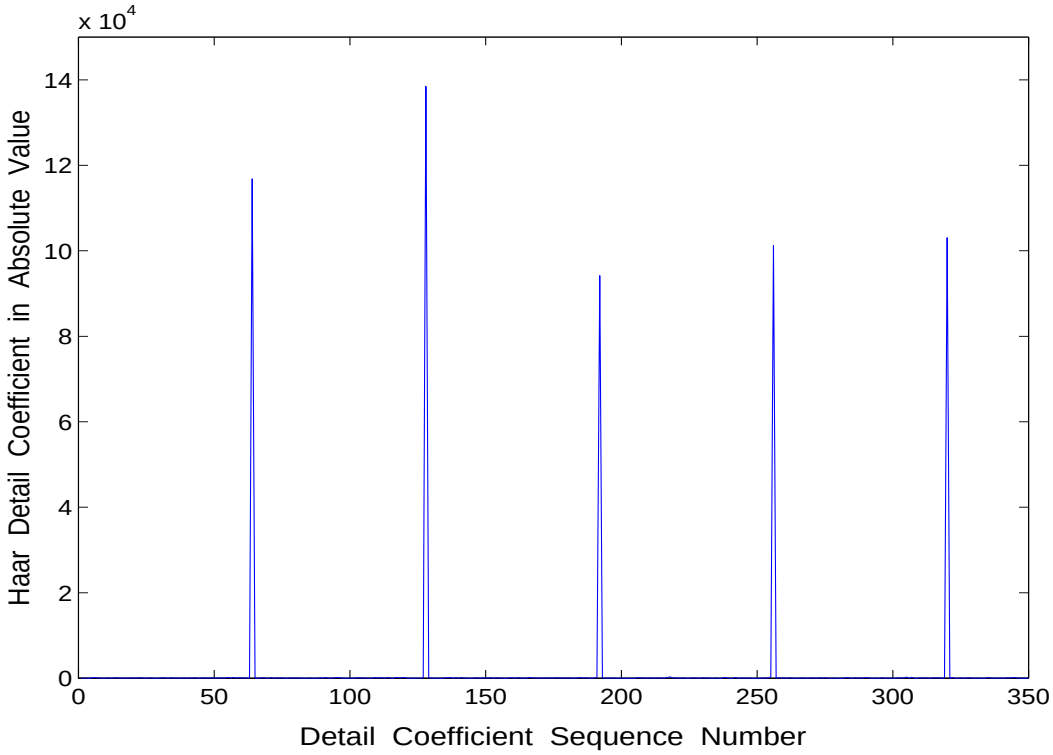

Figure 5.6: ESCAT Detail Coefficients from the Haar Wavelet Transform

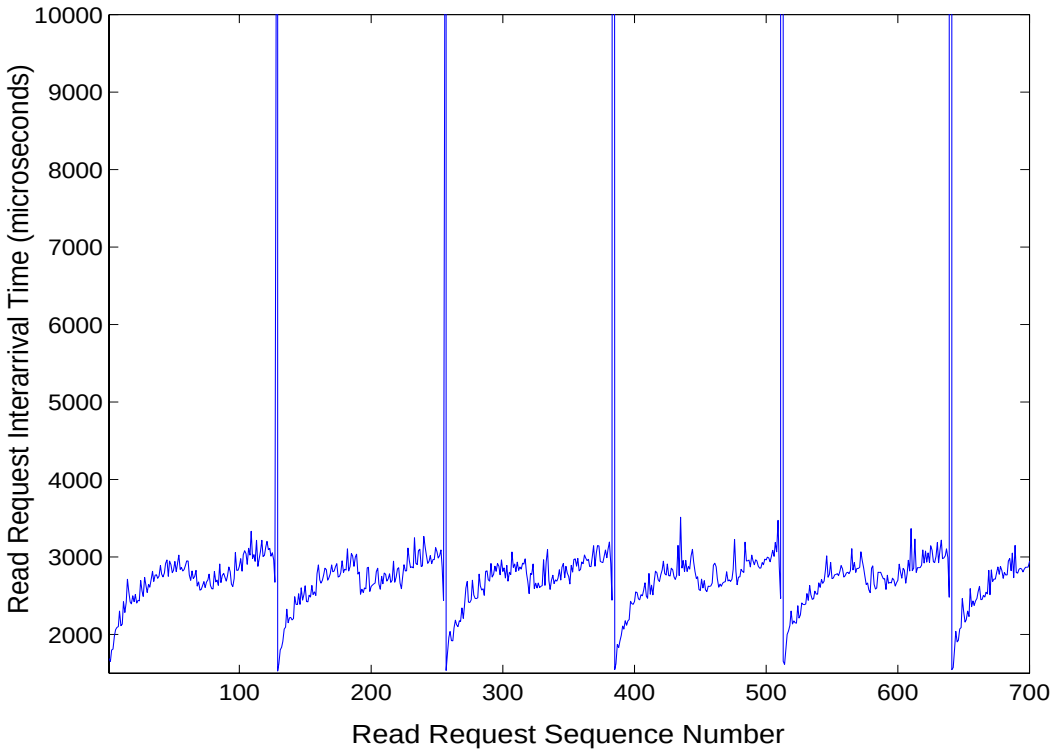

Figure 5.7: ESCAT Series - Sharp Edges at Season Boundaries

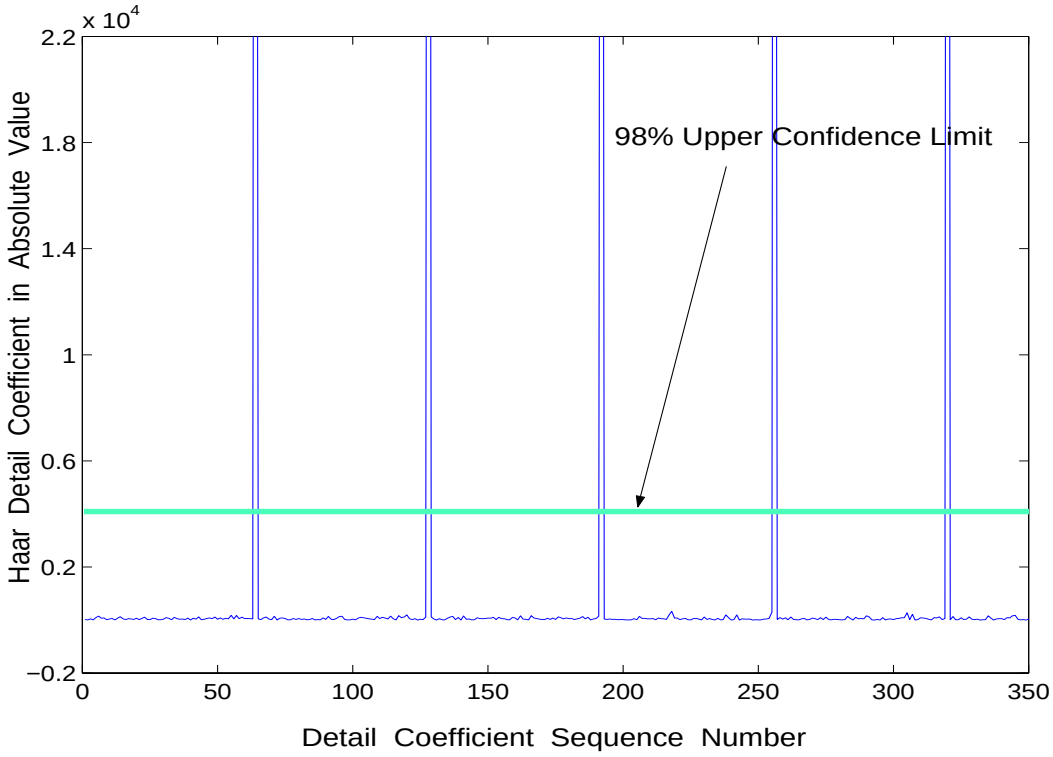

Figure 5.8: ESCAT Detail Coefficients with Upper Confidence Limit

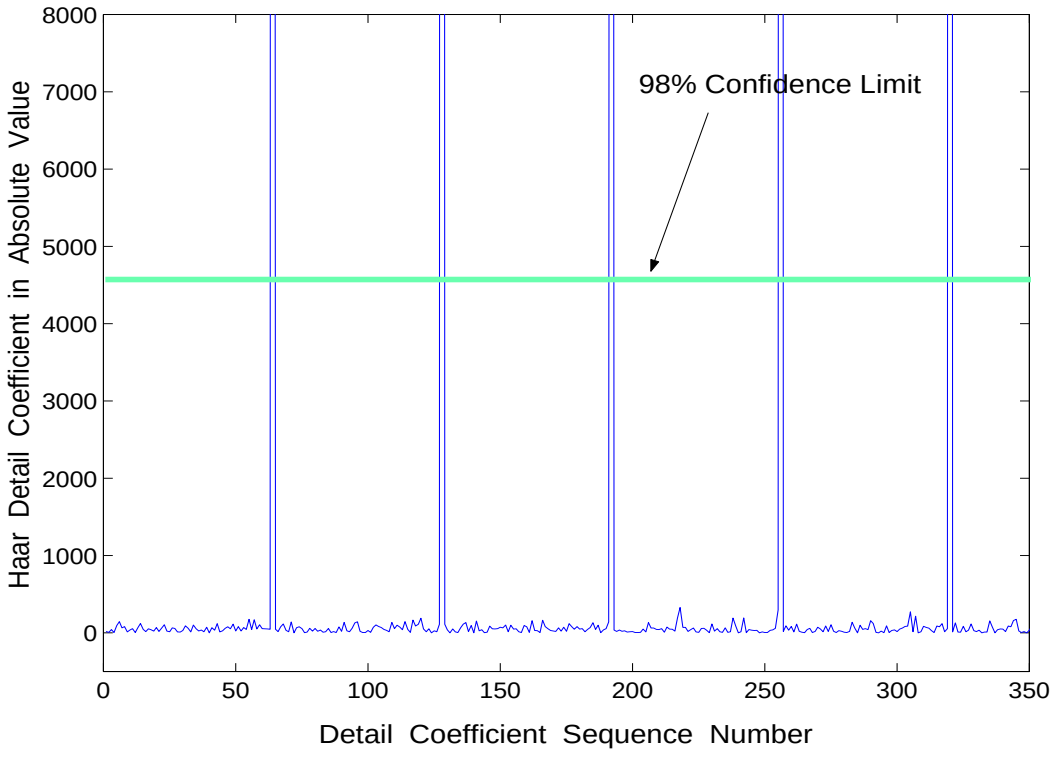

Figure 5.9: ESCAT Detail Coefficients Viewed at Close Range

Chapter 6

Adaptive Prefetching in PPFS2

Having described how to build online models and generate forecasts for I/O interarrival times, in this chapter, we explain how to use online forecasts to adaptively prefetch read requests. For workloads that have non-sequential, periodic and bursty I/O patterns, prefetching guided by online forecasts can provide better performance than sequential lookahead, as the amount of data prefetched can be dynamically adjusted to applications' access patterns and resource usages in the supporting environment, e.g., the disk load.

First, in §6.1, we introduce PPFS2 – a portable parallel file system that provides a flexible environment for prefetching experiments. Next, in §6.2, we present an adaptive prefetcher, a system that integrates the ARIMA and Markov modelers in PPFS2. Prefetches are scheduled based on the models' predictions and the disk service times. The prefetcher's implementation is briefly described in §6.3.

6.1 PPFS2 – a File System with Adaptive Resource Control

PPFS2 [51] is a user-level parallel I/O library designed to provide an experimental testbed for high performance file system research on clustered PCs. An overview of PPFS2's architecture is shown in Figure 6.1. Equipped with Autopilot's [47] performance sensors, decision procedures, and policy actuators, PPFS2 offers a flexible, adaptive control infrastructure for investigating real-time adaptive file system control policies.

Autopilot sensors and actuators give PPFS2 the ability to adaptively control I/O resources distributed across heterogeneous environments. Via remote sensors, applications' I/O performance

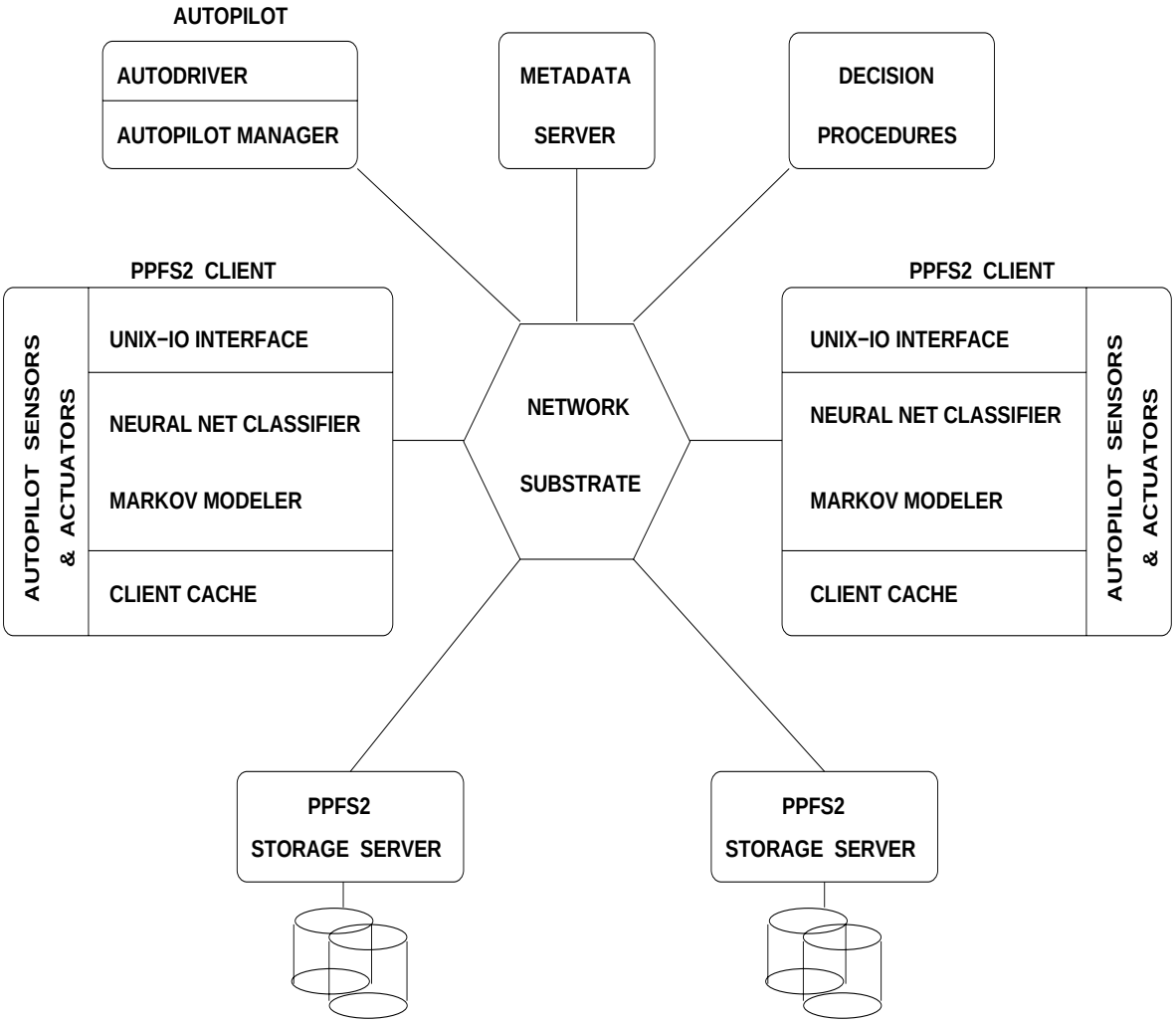

Figure 6.1: PPFS2 Architecture

data can be captured at run time from various PPFS2 components, without requiring application code modifications. These real-time sensor-gathered data are then examined by decision procedures to determine suitable file control policies dynamically. On the other hand, via Autopilot actuators, policies can be remotely implemented by directing control actions toward target resources. For example, when the file cache configuration decision procedure detects small read/write request sizes from sensor data, it instructs the cache reconfiguration actuator to dynamically decrease the cache block size.

6.1.1 PPFS2 Base Architecture

The PPFS2 adaptive resource control infrastructure is built atop a base architecture with three main components: application clients, storage servers, and a metadata server. Application clients initiate I/O requests through a Unix I/O interface. Physical I/O can be issued either to the clients' local disks or remotely to the storage servers' disks. At file open, the metadata server provides information about the location and layout of the file on local and/or remote storage servers. Each storage server, in turn, manages data in its local disks and communicates with application clients through the network.

PPFS2 exploits locality in I/O requests and builds user-level cache structures to allow caching operations in the application address space, minimizing expensive Unix file system call overhead. Associated with each opened file is a cache of fixed size blocks. Each block contains state information as well as actual data. All read/write requests are satisfied either from the blocks in clients' own caches or from storage server caches. Cache misses are resolved by invoking functions of the underlying Unix file system to perform disk I/O. Through Autopilot actuators, caches can be dynamically reconfigured for various parameters such as cache size and cache block size to match an application's characteristics.

6.1.2 Qualitative Access Pattern Classification

To allow automatic, adaptive selection and tuning of file system policies during program execution, a neural net driven classifier [37] provides online identification and classification of I/O access patterns. Classification results provide qualitative descriptions of file accesses, such as sequential,

strided, or random. Depending on the description outcomes, decision procedures intervene to select appropriate control policies. For example, MRU cache replacement policy may be selected when the classifier detects a sequential access pattern.

6.1.3 Quantitative Access Pattern Prediction via Markov Models

To complement qualitative classification, a probabilistic Markov modeling system [41] provides online, quantitative predictions of future file block accesses. Depending on I/O behavior, Markov models can be created online or offline. Online models are suitable for data dependent access patterns because they allow dynamic tracking of block accesses. Reciprocally, offline models, built from previous program executions, capture the transition probabilities of all past references for an entire execution. They are appropriate for patterns that depend on long access history but vary little across executions. Three online prediction strategies can be used to accommodate various access patterns:

- The *greedy method* generates forecasts by finding the *'most likely transition at each state'*. As a result, the selection of states is dominated by the highest transition probability from the *initial* state, even when there exist other paths (sequence of blocks) with aggregate transition probabilities higher than the one chosen. Forecasts are produced until the total transition probability of the selected path exceeds some predetermined limit, or until the number of predictions reaches a preset cutoff length.

- The *path method* finds the sequence of blocks that has the *highest total* transition probability and predicts based on this sequence. It attains better prediction accuracy for those paths that start with low transition probabilities. However, it is more costly as the state branching factor – number of transitions emerging from a state – becomes large.

- The *amortized method* computes the *state occupancy probability vectors* over the *entire execution* of the application. Each vector, representing one state, contains the transition probabilities of moving from that state to its immediate neighboring states. In the occupancy vector of the current state, the entry with maximum probability is chosen to determine the next state in the path. Unlike the other two approaches, the amortized method performs well

when recurrent access patterns are dominant because blocks that are visited multiple times will have a high transition probability in the occupancy vector.

In our prefetching system, we use the PPFS2 Markov models to quantitatively determine which file blocks to prefetch. Markov models of block accesses are built online and predictions for future blocks are created dynamically from the models via the greedy method. Markov model experiments conducted on selected I/O traces from a suite of scientific applications indicated that, for most traces, the greedy method performs better than the other two [41].

6.2 Adaptive Prefetching System

The design of our adaptive prefetcher, illustrated in Figure 6.2, is centered around two concurrent I/O activities: *demand fetch and prefetch*. Prefetched blocks are placed in the PPFS2 cache in anticipation of future arrivals. If correct blocks are predicted and brought into the cache before being requested (not late), demand fetches can be satisfied directly from cache hits, avoiding I/O stalls. Otherwise, cache misses will occur and will be resolved by initiating disk reads on demand.

The adaptive prefetcher includes two components: the prefetch schedule builder and the prefetch issuer; the latter assumes the task of initiating disk I/O for the scheduled predicted blocks.

6.2.1 Prefetch Schedule Builder

The schedule builder accepts three inputs: disk service time estimates, predictions for data blocks from the Markov modeler, and predictions for block interarrival times from the ARIMA modeler. Blocks are obtained by grouping read requests according to a predetermined block size. Interarrival times for these blocks are measured as the elapsed times between arrivals of requests for two distinct blocks. Disk service time is estimated from monitored time, waiting for disk I/O.

To generate prefetch schedules, we slightly modify the Forestall algorithm [60, 31]. This algorithm is the result of several comparative studies, made by Tomkins *et al.*, on different hint-based prefetch schemes. It combines Patterson's conservative and Cao's aggressive prefetching methods [44, 9].

- Assuming zero disk queuing delay (infinite number of disks running in parallel), Patterson

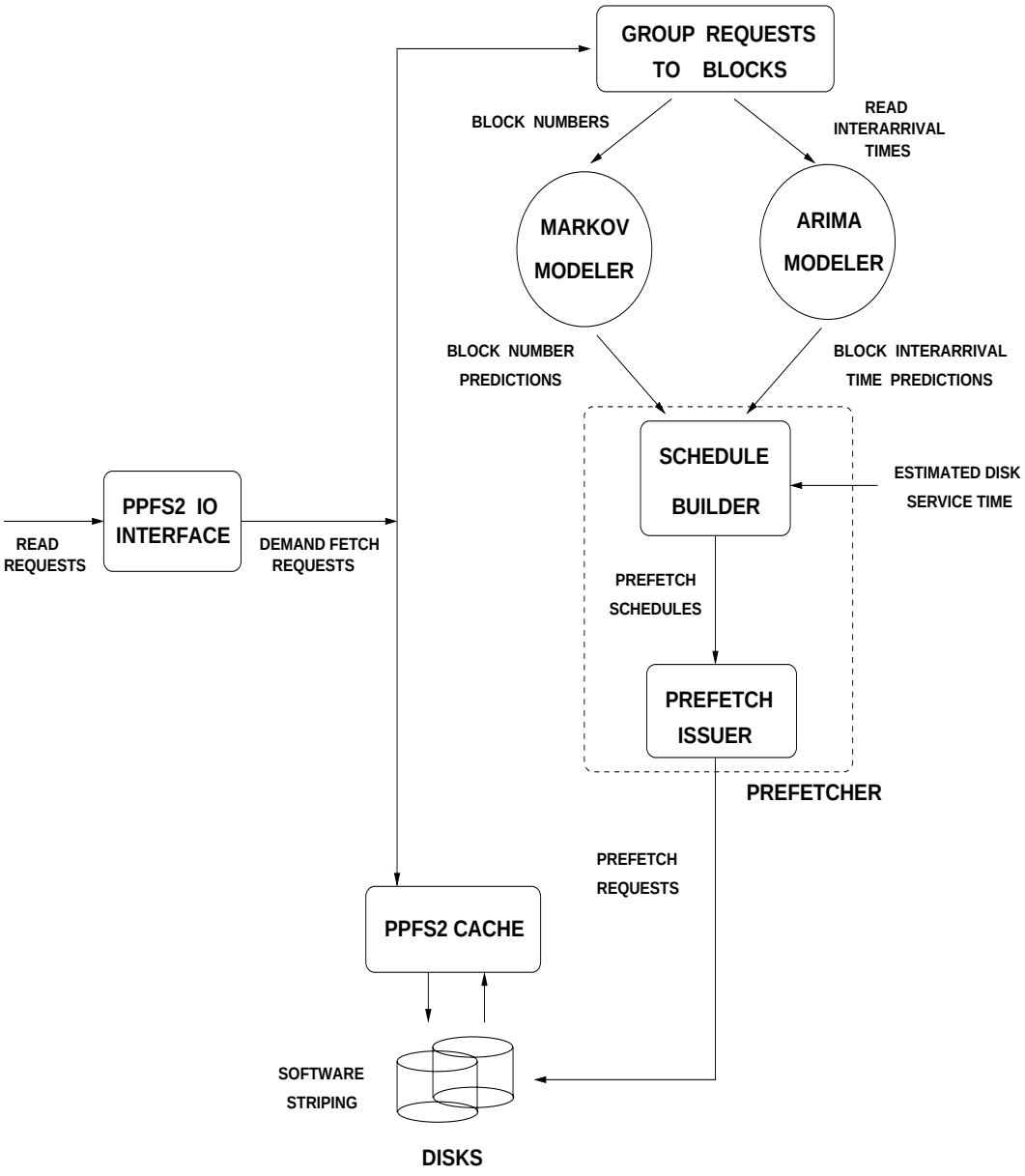

Figure 6.2: Adaptive Prefetcher Structure and Interacting Components

proposed to conservatively prefetch a fixed number of blocks, defined as the prefetch distance. However, in many systems where disk queuing delays exist, the assumption is violated. The number of blocks to be prefetched could be under-estimated, resulting in late prefetches.

- Conversely, Cao proposed to aggressively prefetch as much as possible. Prefetches are initiated as soon as a disk is idle, with highest priority given to blocks having the earliest deadlines. This method can over-estimate the number of prefetches, flooding the file cache with premature blocks and causing unnecessary cache block replacements.

To reach an acceptable compromise, the Forestall algorithm uses Patterson's fixed prefetch distance if the disk service rate is faster than or equal to the request arrival rate. Otherwise, prefetch distances are adjusted based on constraints in request interarrival times and variations in disk service times. The Forestall algorithm has been shown to achieve good performance via extensive simulation studies [31, 60].

We apply the following Forestall equation to detect disks with queuing delays, defined by Tomkins in [60] as *constrained disks*:

$$\sum_{i \mid d} isInCache(block\ i) \times T_{Disk} \quad \geq \quad \sum_{i \mid d} T_{CPU_i}$$

$i \mid d$ = for a block i to be fetched from a disk d

T_{Disk} = disk service time for blocks at instant t

T_{CPU_i} = interarrival time for block i

$isInCache(i)$ = boolean function indicating cache hit/miss

Intuitively, a disk is constrained when data blocks arrive faster than they can be served. The key idea behind the equation is to *determine whether there is sufficient time for the disk to prefetch predicted blocks when requests for these blocks arrive.* This is achieved by comparing the total disk time spent to fetch those predicted blocks that miss the cache with their predicted interarrival times. We have two cases:

- **Constrained disk**. If the total predicted interarrival time is smaller than the associated total disk service time, the disk will be over-utilized and will incur I/O stalls. Prefetching these blocks should be initiated immediately to minimize further delays. Prefetching any

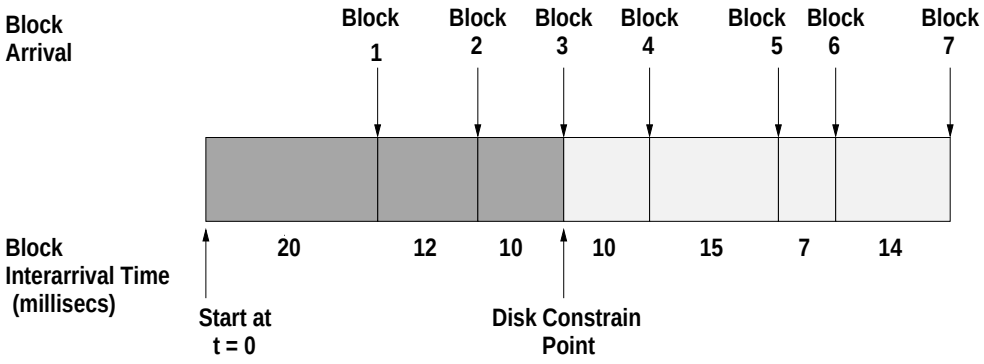

Figure 6.3: Detection of a Constrained Disk

Block	Interarrival Time	Disk Service Time
1	20	15
2	12	15
3	10	15
Total	42	45

additional block will not benefit performance because the block access will stall. As a result, a prefetch schedule is built to contain only those blocks up to and including the constraint point.

Figure 6.3 provides an example to illustrate disk constraint detection. Consider the first three blocks in the stream of predicted blocks, assuming a disk service time of 15 milliseconds per block. Starting from time $t = 0$, the first block will arrive at time $t = 20$ milliseconds. As 20 is greater than disk service time 15, there will be enough time to prefetch block 1. For the second block, although its interarrival time, 12, is smaller than disk service time, it still can arrive before needed if prefetching starts at time zero. This is possible because the total predicted interarrival time for the first two blocks is $20 + 12 = 32$, larger than the total disk service time of 30. However, block 3 will arrive late even if we start prefetching at time zero, as the total predicted interarrival time, 42, is smaller than the total disk time required, 45. The disk will become constrained when serving block 3 – the constraint point. Hence, the prefetch schedule contains only the first three blocks to satisfy requests with the earliest deadlines. The remaining blocks will be on later schedules.

- **Unconstrained disk**. If the total predicted interarrival time is larger than the total disk

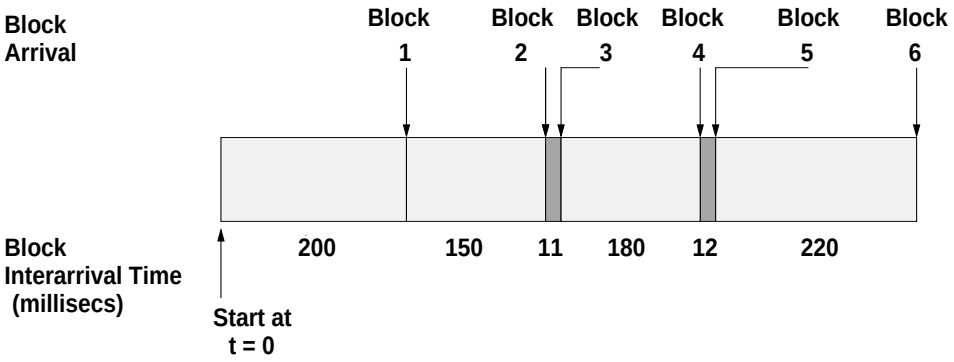

Figure 6.4: Unconstrained Disk

	Interarrival Time		Disk Service Time	
Block	Individual	Cumulative	Individual	Cumulative
1	200	200	15	15
2	150	350	15	30
3	11	361	15	45
4	180	542	15	60
5	12	553	15	75
6	220	773	15	90
Total	773		90	

service time, the disk will not be over-utilized and will keep pace with request arrivals. Here, we deviate from the Forestall algorithm, which uses Patterson's *fixed* prefetch distance [44], computed as the ratio of the disk service time and the minimum request interarrival time (i.e., cache hit time). Instead, we use a *variable* prefetch distance that consists of approximately one second's worth of predicted blocks (i.e. a group of blocks with total predicted interarrival time summing to about one second). *Variable prefetch distances take into account application I/O intensity even when a disk is unconstrained.* For example, long interarrival times require fewer blocks to be prefetched, hence shorter prefetch distances.

Prefetching using the criterion of a one-second window for unconstrained disks may cause premature prefetches. However, we note that the number of prefetched blocks is bounded: these blocks can arrive early by at most one second, limiting the probability of flooding the file cache. In addition, assuming accurate predictions, blocks prefetched into the cache during the previous second will have been consumed by the application and returned to the free buffer pool when the next second arrives.

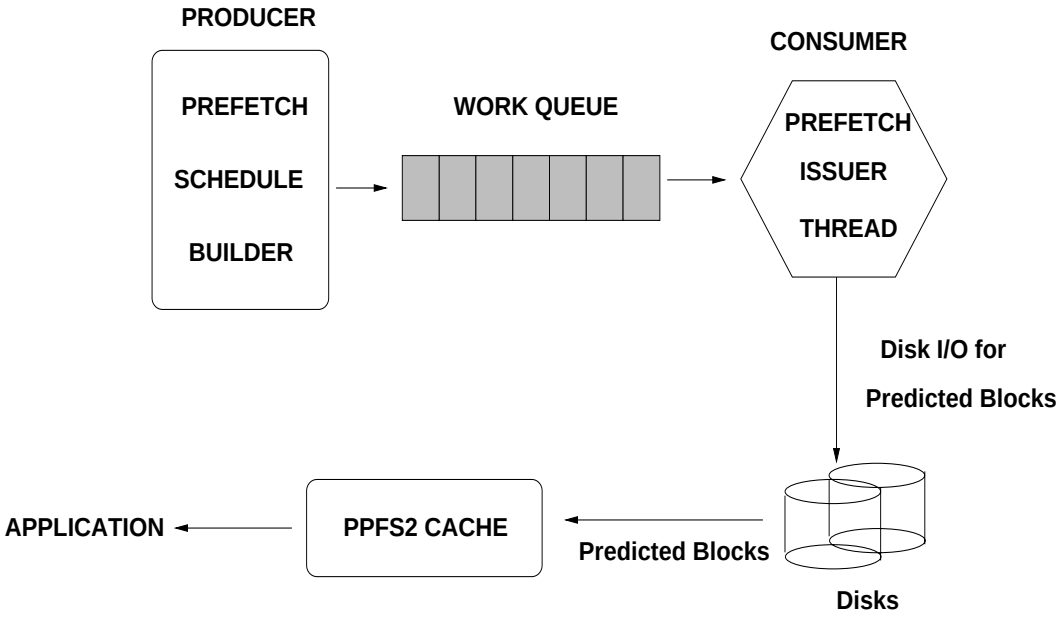

Figure 6.5: Prefetcher's Implementation

Figure 6.4 illustrates an example of building prefetch schedules for an unconstrained disk. Again, we assume a disk service time of 15 milliseconds per block. Counting from $t = 0$, the cumulative interarrival time for every block in the stream of predicted blocks is larger than the corresponding cumulative disk service time. The first 3 blocks have a cumulative predicted interarrival time of 361, greater than the disk service time counterpart of 45. Considering the entire stream of blocks, the total predicted interarrival time of 773 far exceeds the total disk service time of 90, despite the presence of a few, occasional blocks with interarrival times smaller than the disk service times (e.g. blocks 3 and 5). As a result, a prefetch schedule is built to include all the blocks in the stream, having cumulative predicted interarrival time less than one second.

6.3 Implementation

Based on the producer-consumer model, we implemented the prefetcher using a First-in First-out work queue to coordinate the flow of schedules between the schedule builder and the prefetch issuer. Figure 6.5 summarizes the implementation's main data path, starting from the producer placing schedules in the work queue, to the issuer thread that initiates I/O requests to disks, and finally to the application that receives prefetched blocks from the PPFS2 cache.

The state of the PPFS2 cache is continually updated as existing blocks are evicted and replaced by the new prefetched blocks. In our implementation, cache replacement is controlled by two pointers kept at each end of the cache block list. Blocks that are newly prefetched will be directed to the *non-evicted end* of the list to avoid premature eviction. Blocks that are not reused will be moved to the evicted end, allowing obsolete blocks to be gradually pushed out.

Depending on the outcome of a demand fetch request – hit or miss, three scenarios are possible:

- In the case of a cache hit, demand fetch can be immediately satisfied by prefetched data without I/O stalls. This situation occurs when we have accurate predictions in both the space and time domains, prefetching the right blocks at the right times.
- However, if a cache hit is pending, i.e., demand read requests match the blocks for which prefetches have been issued but not yet completed. Prefetches will be late and applications will stall for I/O completion.
- If there is a cache miss, no block in the cache can satisfy the request because of erroneous predictions in either the space or time domain. PPFS2 issues disk reads at once to bring in data on demand.

These scenarios suggest that the number of cache misses can be used to evaluate a prefetching system's performance. A large number would signal the system's inability to predict accurately and prefetch in time.

6.4 Summary

In this chapter, we described algorithms and techniques to realize an adaptive prefetching system, built atop the PPFS2 experimental testbed. We explained how to integrate interarrival time predictions from ARIMA models and file block predictions from Markov models to dynamically build prefetch schedules that can adapt to changes in applications' I/O access patterns in both spatial and temporal domains. We also proposed a simple cache block replacement policy that can avoid premature eviction of new prefetched blocks while allowing stale blocks to be gradually removed from the cache.

Chapter 7

ARIMA Automodeler Experiments

Because incorrect forecasts from erroneous models can severely degrade the performance of a prefetching system, we need to validate Automodeler. Our validation investigates three issues: a) ability to identify models that can approximate I/O behaviors b) prediction accuracy, and c) processing overhead. The latter is an important consideration as it can limit the practical usefulness of Automodeler in improving application I/O performance.

Following an overview of the experiment environment in §7.1, Section §7.2 describes our initial *offline* experiments performed on I/O traces of scientific applications. We selected two representative applications from the Pablo I/O event trace suite, captured using the Pablo performance instrumentation and characterization toolkit [46].

Further validation tests are described in the remainder of this chapter. We conducted three *online* experiments, where the test applications and Automodeler were executed concurrently:

- A synthetic workload containing periodically bursty read interarrival times, modeled using *individual* read requests. The goal is to validate the extended least squares (ELS) and the recursive differencing/integration (RDI) algorithms.
- To minimize processing overhead, the same synthetic workload was modeled using the interarrival times of *blocks* of read requests. Effects of this block downsampling strategy on overhead reduction and prediction accuracy are evaluated.
- Finally, a computational physics code was modeled using block downsampling during program execution. The purpose is to verify that block downsampling does not compromise prediction accuracy in scientific applications.

7.1 Experiment Environment

7.1.1 Hardware Platform

Our experiments were conducted using the Pablo group's 8-node cluster. Each of the eight PCs is a Dell Dimension XPS with Pentium II 266 MHz processor and 128MB main memory. One additional PC is used as a front-end server for the cluster. The remaining machines function as computational nodes and PPFS2 storage servers. All PCs in the cluster are inter-connected with 100 Mb/second, switched, fast Ethernet network.

The cluster has a total of 24 disks with 3 disks per machine. These Western Digital WDE4360 Ultra SCSI 4.3 GB disks are connected to an Ultra Wide SCSI host adapter. The Ultra Wide SCSI bus supplies a maximum aggregate bandwidth of 40 MB/second. The full stroke seek time is 18 milliseconds for each disk and the disk average access latency is 8.4 milliseconds.

7.1.2 Integration with PPFS2

All experiments presented in this section were performed using PPFS2 as an I/O layer. In PPFS2, file blocks are obtained by grouping requests based on a fixed block size. Applications issue I/O requests through a Unix I/O interface that passes the requests to a caching and prefetching engine. When cache misses occur, disk I/O will be issued to the application clients' local disks using Unix file system calls. The interarrival time computed for each new block of requests is given to Automodeler.

Automodeler was implemented in PPFS2 as a group of C++ objects. It used Davies' newmat09 matrix package [16] for matrix manipulations. The implementation provided a simple function call: *timeSeriesModel(interarrival time)*. Based on the first window of interarrival times, Automodeler's identification subsystem constructed a model structure. This structure $(p, d, q) \times (P, D, Q)^S$ specified the number of elements in the autoregressive and moving average components, together with the differencing levels and the season lengths for non-stationary and seasonal data patterns. Once a model was identified, the arrival of every new interarrival time triggered ELS and RDI to recursively estimate parameters during program execution. Subsequently, n-step ahead forecasts for future interarrival times were produced.

7.2 Modeling I/O traces of Scientific Applications

We selected a representative subset from the Pablo I/O trace suite to highlight the access patterns commonly found in scientific applications. In this study, we present the temporal I/O behavior of two codes: a fluid dynamics code and an electron scattering code. We model and analyze the interarrival times of read requests, extracted from the I/O traces, to create one-season ahead predictions.

7.2.1 Fluid Dynamics Code: PRISM

PRISM is an implementation of a 3-D numerical simulation of the Navier Stokes equation to study the dynamics and transport properties of turbulent flows in fluids [26]. PRISM read requests have mixed sequential (for loading file headers) and interleaved patterns (for the remaining data) [53]. Automodeler analyzed the correlations in PRISM read interarrival times and built a model on the fly. Parameters in the model were continually estimated with the arrival of new read requests.

7.2.1.1 Automatic Model Structure Identification

The different tests and procedures used to identify a model for PRISM read interarrival times were explained in detail in §4.2.5. Here, we merely summarize the main results and the important steps taken to achieve the results.

The overall behavior of PRISM read interarrival times was disrupted by two phase transitions – one after program initialization and the other near the middle of program execution. Automodeler began by applying the Haar Wavelet transform to the first sample of 300 observations. It disclosed a maximum high frequency signal at observation 88, shown in Figures 5.1 and 5.2. The sample means on either side of this observation (319 and 55 respectively) differed from each other by at least a ratio of five to one, *signaling a phase transition.* Associated with initialization, this transition was short and transient – it could not be reliably used for model identification.

Hence, Automodeler continued the identification task by analyzing the next set of observations, starting right after the first phase transition. Figure 4.8 displays the ACF estimated for the PRISM series, after it has been regularly differenced once. The L-1 distance test succeeded in detecting a season length of 10 from the significant spikes extending above the upper 98% confidence limit.

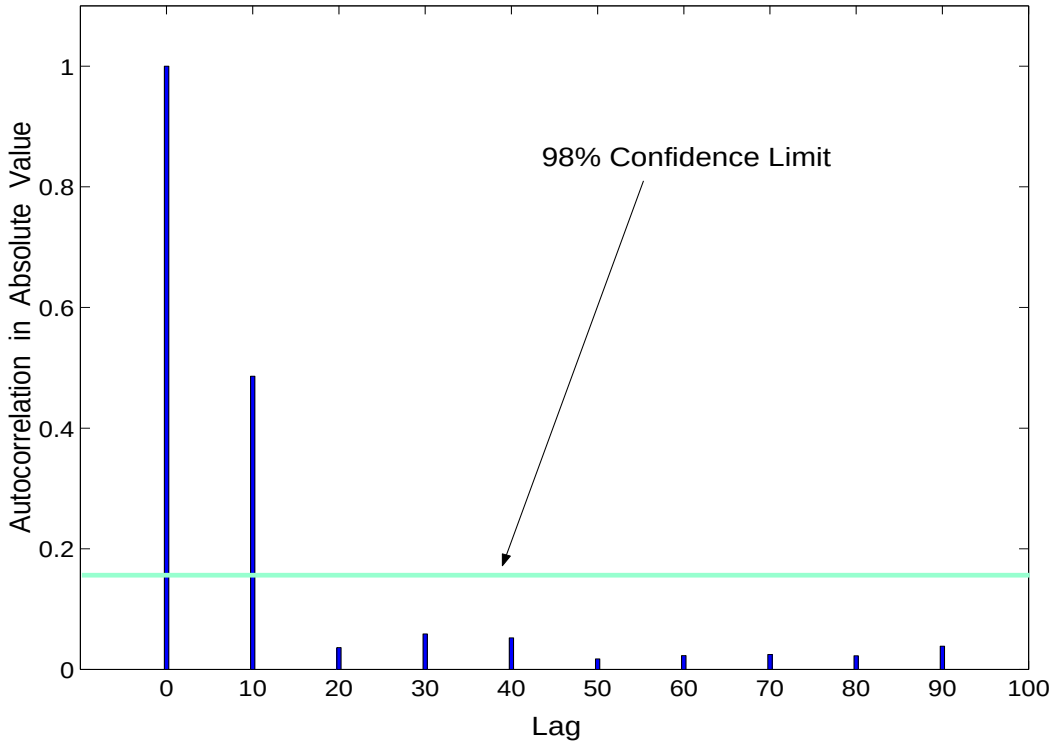

Figure 7.1: PRISM ACF (after Seasonal Differencing) at Season Boundaries

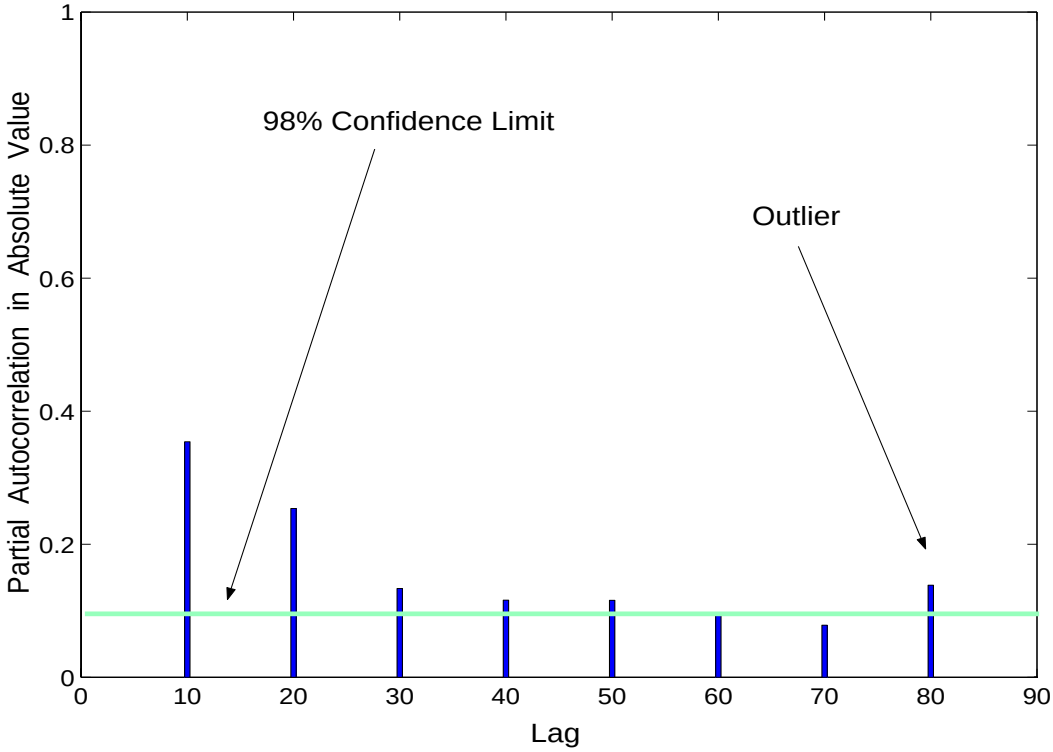

Figure 7.2: PRISM PACF (after Seasonal Differencing) at Season Boundaries

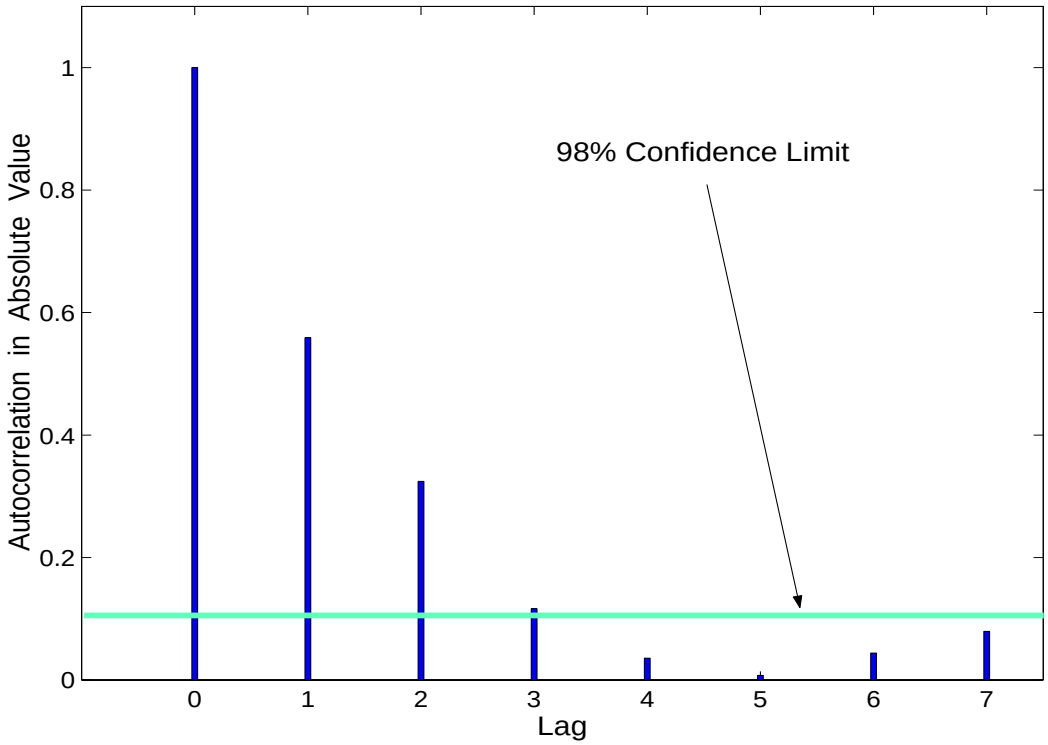

Figure 7.3: PRISM ACF (after Seasonal Differencing) within the First Season

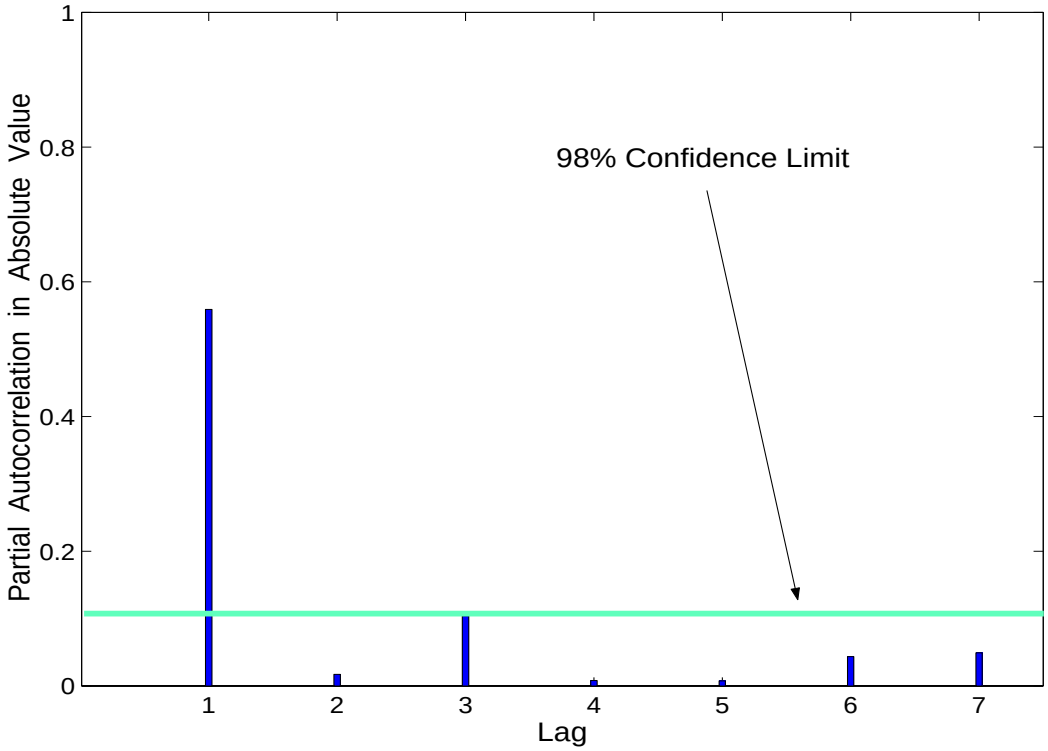

Figure 7.4: PRISM PACF (after Seasonal Differencing) within the First Season

Upon detecting a season, Automodeler differenced the PRISM series with the season length 10 and computed the ACF and PACF of the differenced results. The autocorrelations *located at the season boundaries*, i.e., lags 10, 20, ..., were automatically selected and used to identify the model's seasonal component. For convenience, we replot these selected values (previously shown in Chapter §4) in Figures 7.1 and 7.2. The average rate of change test recognized an exponential decay pattern in the PACF, despite the existence of an outlier. From the ACF, the lag count test isolated a single spike at lag 10, the first seasonal lag after zero. Matching the detected patterns with the broad characteristics in Table 4.1, Automodeler built an ARIMA model structure $(P, D, Q)_S = (0, 1, 1)_{10}$ for the seasonal component.

Similarly, to identify the non-seasonal component, Automodeler selected the autocorrelations *located within the first season* (see Figures 7.3 and 7.4). The identification tests disclosed one autoregressive element, manifested through the exponential decay during the first few lags in the ACF plot, and a single large spike at lag 1 in the PACF plot. The model for the non-seasonal component is thus $(p, d, q) = (1, 0, 0)$.

In short, the aggregate model $(1, 0, 0) \times (0, 1, 1)_{10}$ estimated for PRISM specifies that, in the seasonally differenced PRISM series, each difference value depends only on the previous difference (one autoregressive element), and the previous noise term (one moving average element) separated by ten requests.

7.2.1.2 Results and Analysis

This section describes results when the structure identified above was used to model the *entire* PRISM series. Figure 7.5 compares observed read interarrival times with one-season ahead predictions. The x axis represents read request arrivals. Each arrival was assigned an unique sequence number. The y axis represents read request interarrival times, measured in microseconds. The plot shows three phases with two transitions, one near the beginning and the other near the middle of the code execution. In general, predictions follow closely PRISM's read behavior for 4500 requests. The root mean square prediction error is about 18%. It is computed using prediction error ratios,

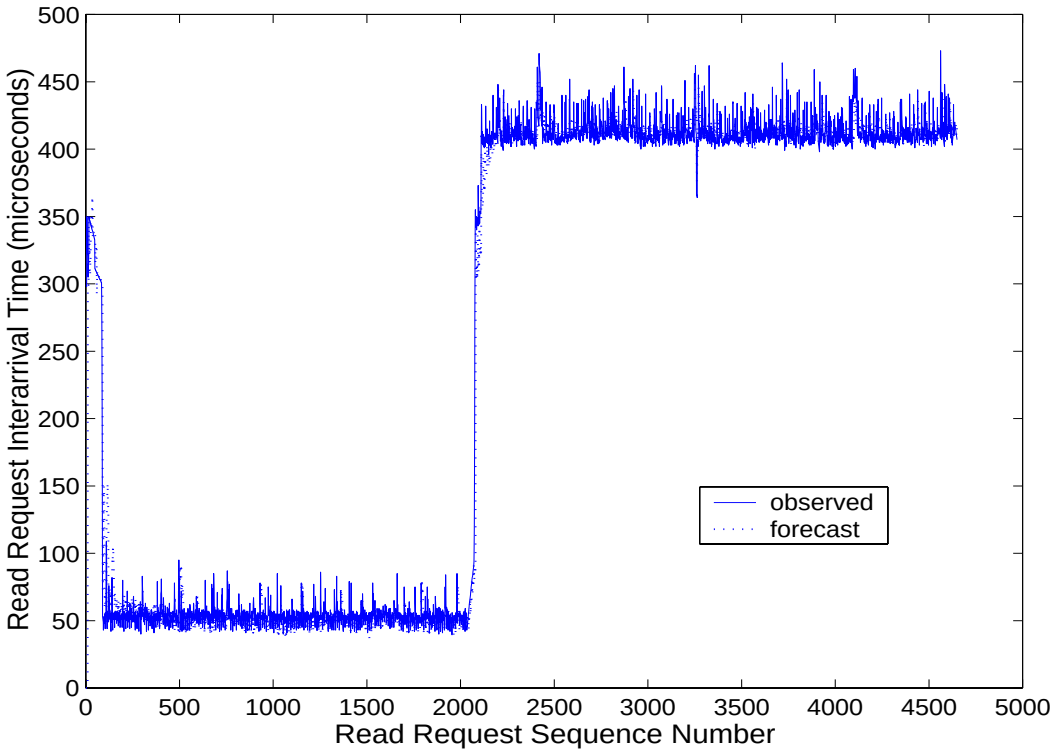

Figure 7.5: PRISM's General Distribution of Interarrival Time Predictions

defined as (forecast - observed) / observed:

$$\text{root mean square prediction error} = \sqrt{\frac{\sum (\text{prediction error ratios})^2}{\text{total number of reads}}}$$

This results in an average prediction accuracy of 82%.

To examine how Automodeler tracks behavior changes at phase transitions, Figure 7.6 displays the first 300 observations of the PRISM series. The first dotted vertical line rising from zero to 300 microseconds represents the first prediction, produced after the initialization period. The second vertical drop, from 300 to about 50, corresponds to the first phase transition. These changes were automatically captured by the model parameters, which were re-estimated for every read request. Because the changes have large magnitudes, estimated parameters converge slowly, resulting in a gradual improvement in prediction accuracy, as gaps between observed and predicted values narrow.

Figure 7.7 provides a detailed snapshot of requests 300 to 400. It shows successful predictions of the PRISM seasonal read pattern, appearing at regular intervals of ten requests, with the exception of two outliers at observations 302 and 382.

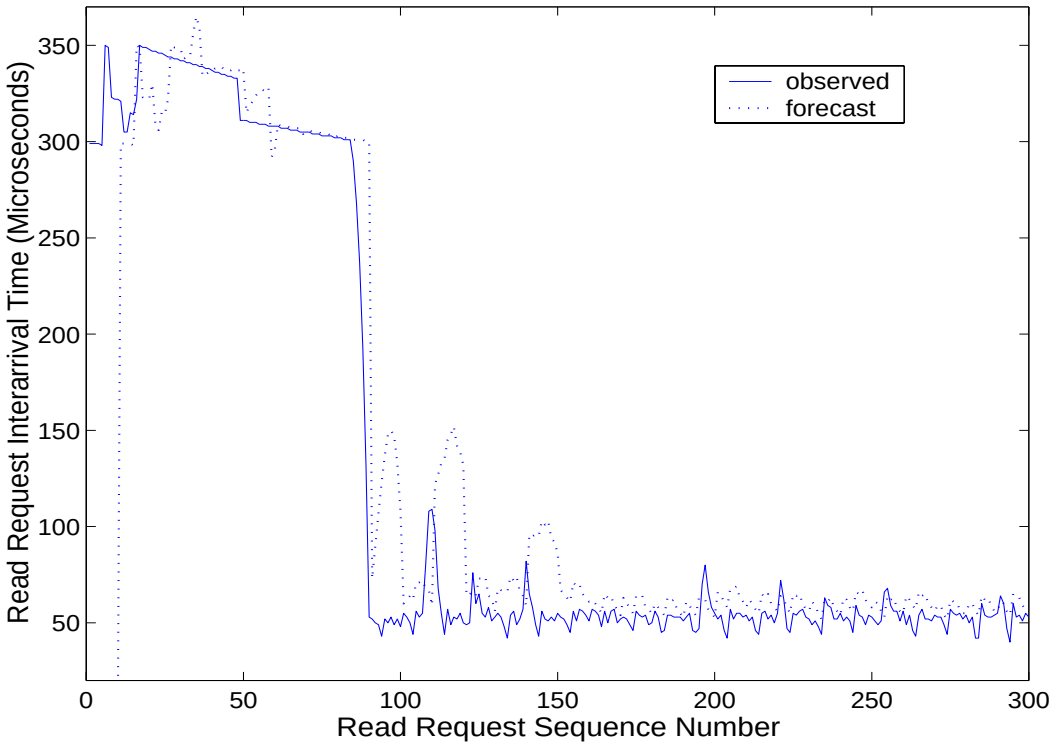

Figure 7.6: PRISM's Interarrival Time Predictions after Phase Transition I

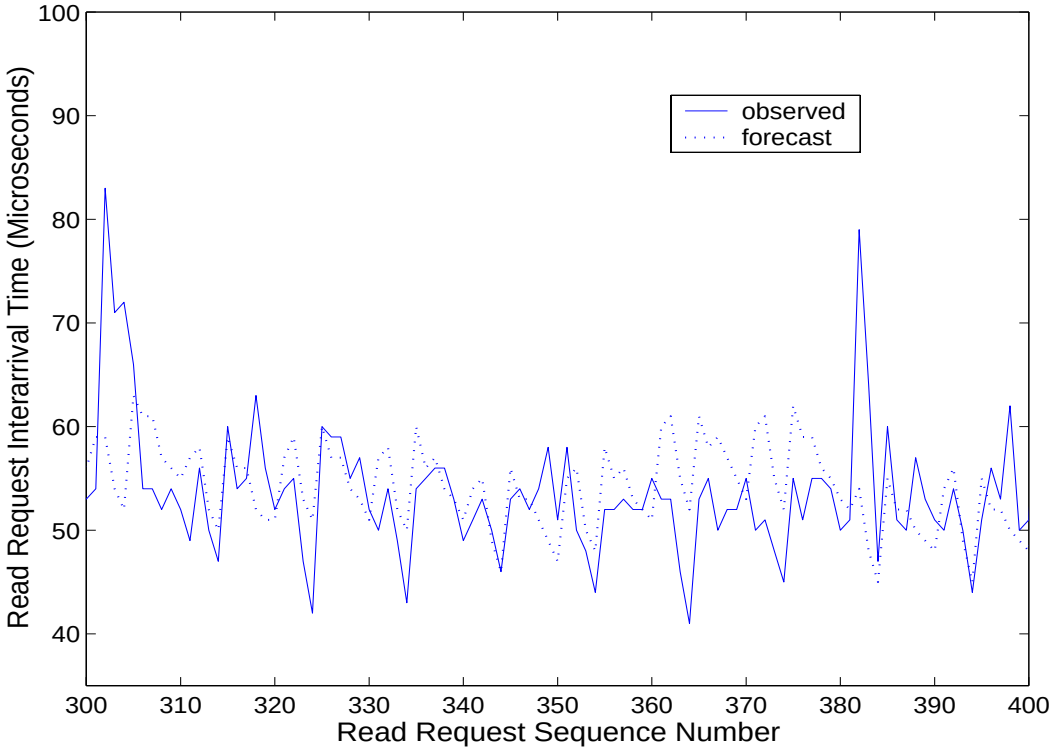

Figure 7.7: Detailed Snapshot of PRISM's Interarrival Time Predictions

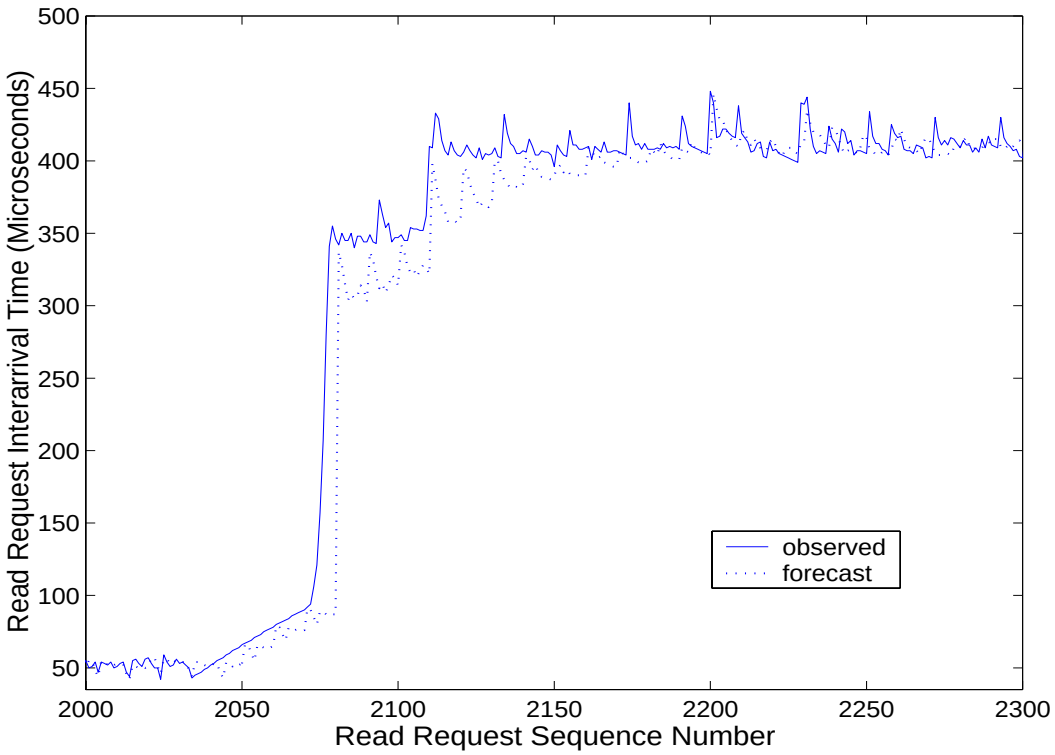

Figure 7.8: PRISM's Interarrival Time Predictions after Phase Transition II

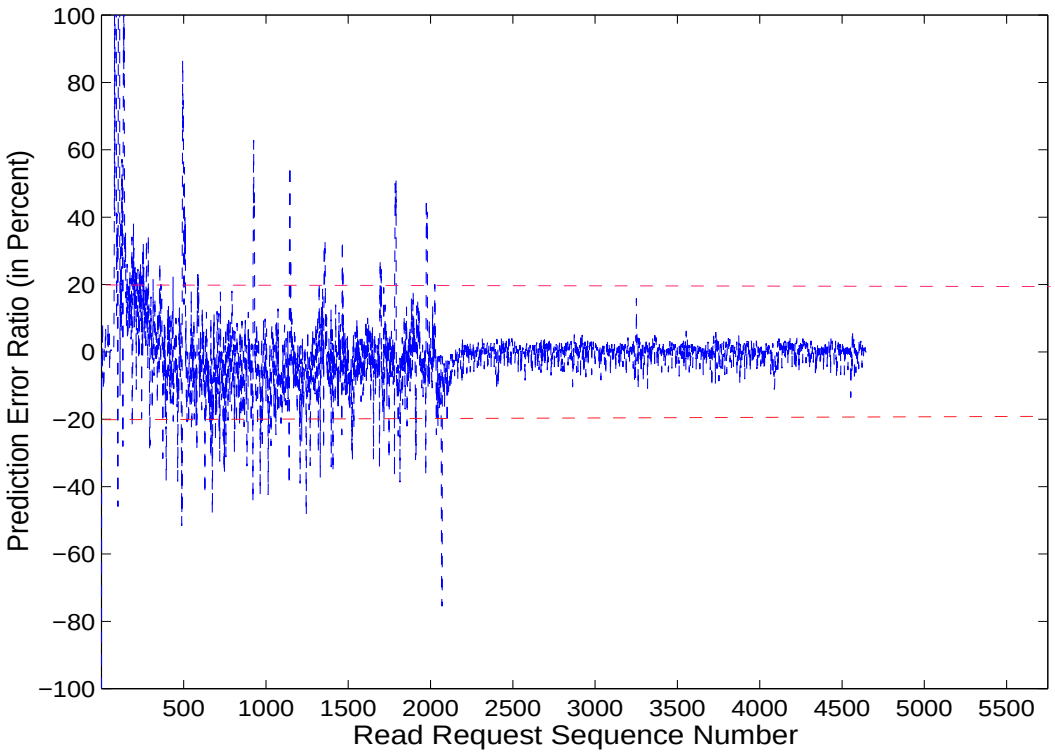

Figure 7.9: PRISM's Prediction Error Ratios

Figure 7.8 illustrates predictions created during the second phase transition, which starts at observation 2040. Read interarrival times slowly increase from 50 to 100 microseconds, followed by a sharp rise from 100 to 350, and a final, gradual increase from 350 to around 420. ARIMA tracks these changes and produces predictions accordingly. Prediction accuracy improves as parameters are changed to gradually adapt to large increases in interarrival times.

Finally, Figure 7.9 plots prediction error ratios in percentages[1] for the entire workload, defined as (prediction - observed) $\times 100$ / observed. Spikes in the plot indicate occurrences of large errors during phase transitions. The majority of errors for the first 2200 observations (before the second transition) fall within $\pm 20\%$ range. However, the second phase has a smaller error range of approximately $\pm 5\%$.

With this experiment, we have preliminarily verified Automodeler's capability to a) automatically identify and build an ARIMA model, b) track phase transitions with related behavior changes, and c) produce forecasts reflecting these changes.

7.2.2 Electron Scattering Code: ESCAT

ESCAT is an implementation of the Schwinger multichannel method to investigate electron-molecule collisions, solving linear systems to obtain the scattering probabilities [66]. Read operations represent 56% of the I/O volume, with roughly equal numbers of small and large size requests, varying from less than 4 KB to more than 256 KB [13]. ESCAT read interarrival times are bursty, where bursts of reads are periodically disrupted by long computation intervals.

7.2.2.1 Automatic Model Structure Identification

Automodeler examined autocorrelations in the first 1280 read interarrival times of ESCAT. Figure 7.10 shows this series' ACF after one regular differencing. The significant correlations were detected at lags $1, 127, 128, 129, 255, 256, 257, 383, 384, 385, 511, 512, 513, 639, 640, 641, 767, 768$ and 896. As these correlations were large, they induced aliasing effects in their neighbors, manifested as clustered values [127 : 128 : 129], [255 : 256 : 257], ... To remove these effects, Automodeler simply chose

[1] Ratios, computed relative to the magnitudes of observed values, give better error assessment than absolute prediction errors (prediction - observed) for interarrival times: an absolute error of 1 microsecond for an observation of 1 microsecond results in an error ratio of 100%, even though 1 microsecond is a small number.

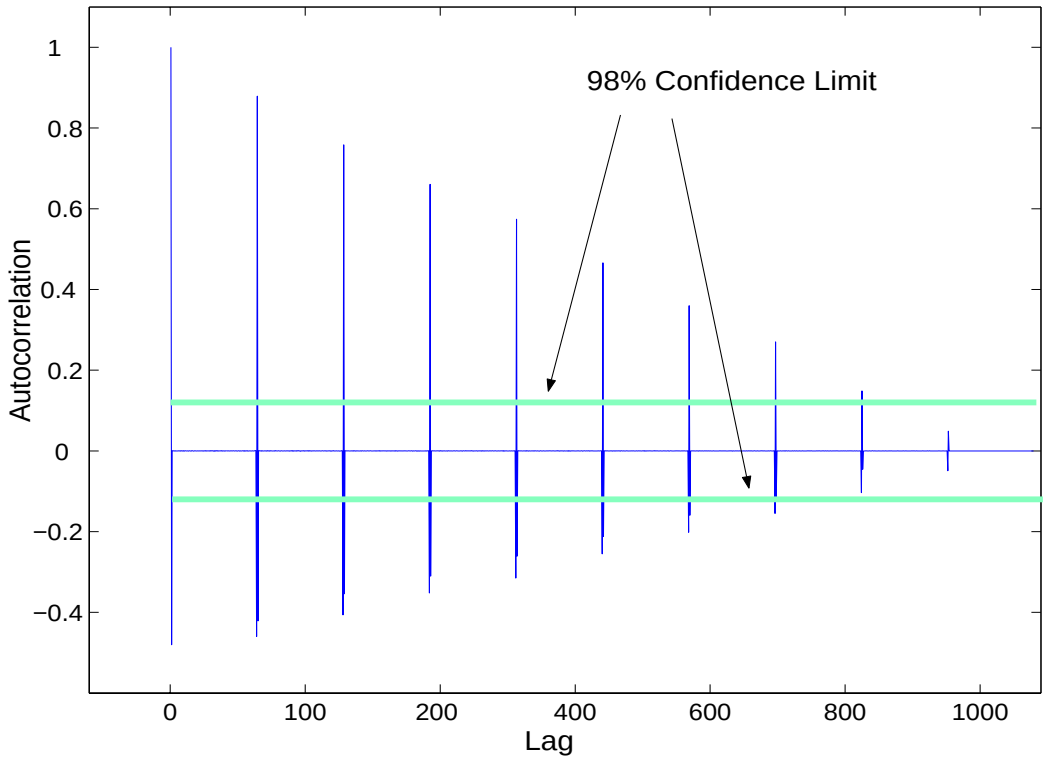

Figure 7.10: ESCAT ACF after One Regular Differencing

the last value in each cluster. It then performed the L-1 distance test on the chosen signals (e.g., $129, 257, 385$) and other non-clustered significant signals (1 and 896) to identify a season length of 128.

Alternatively, for *bursty* time series such as ESCAT's read interarrival times, the Haar wavelet analysis presented in Section §5.1 provides a simpler and faster approach to detect seasonality. The Haar transform does not compute autocorrelations and does not introduce aliasing. Instead, it computes wavelet detail coefficients and a confidence limit to determine the significant signals. Figures 5.6 and 5.8 show ESCAT's Haar coefficients and their 98% upper confidence limit (≈ 4600). The L-1 distance test, performed on the locations of the significant signals (lags $63, 127, 191, 255$, etc.), exposed a season length of 64. Because the Haar transform halves the number of observations, the original season length, 128, is restored by doubling 64.

Seasonally differencing the ESCAT series with a season length of 128 removes all significant dependencies among data *within a season*. This cancellation effect is shown in Figure 7.13 where all correlations are well below the confidence limit, suggesting a model with a *null* non-seasonal

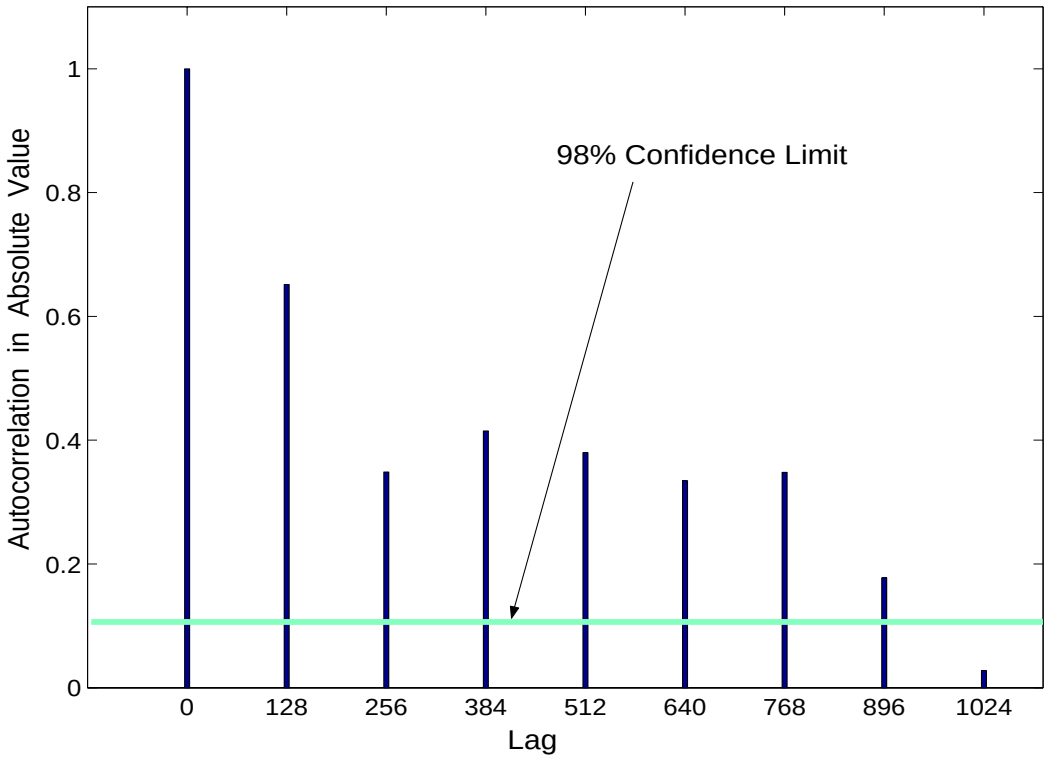

Figure 7.11: ESCAT ACF (after Seasonal Differencing) at Season Boundaries

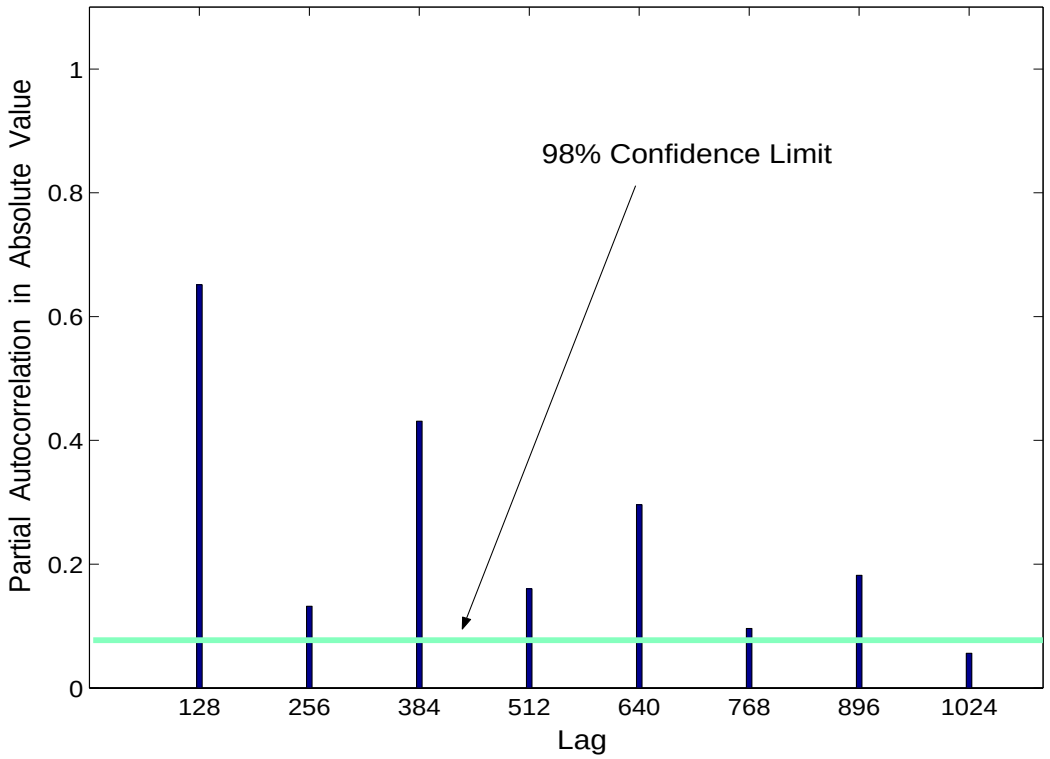

Figure 7.12: ESCAT PACF (after Seasonal Differencing) at Season Boundaries

component.

However, for the seasonal component, significant correlations were detected at the season boundaries 128, 256, 384, $\cdots$, 1024. These boundaries' ACF (Figure 7.11) displayed an overall exponential decay pattern with an average decay rate of 49%. In contrast, their PACF (Figure 7.12) had a mixed pattern – almost abrupt cutoff from odd to even lags, but exponential decay among odd lags. This PACF, with an average decay rate of 69%, was classified as abrupt cutoff. As a result, Automodeler identified a seasonal component $(P, D, Q) = (1, 1, 0)$.

The final composite model for ESCAT's read series is $(0, 0, 0) \times (1, 1, 0)_{128}$. The model indicates that each data point in the seasonally differenced series is correlated with only one data point in the preceding season, separated by 128 time steps. All remaining past data are statistically insignificant.

Our experiments revealed that identifying ARIMA models with large sample sizes often produces better results. For example, had we used a sample of 700 observations, Automodeler would have identified a moving average seasonal component, i.e., $(P, D, Q) = (0, 1, 1)$. The average root mean square prediction error from this model is about 11%, larger than the 5% error produced by the $(P, D, Q) = (1, 1, 0)$ model, identified from a sample size of 1280.

7.2.2.2 Results and Analysis

An overall comparison between one-season ahead predictions (dotted lines with square markers) and observed read interarrival times (solid lines) is illustrated in Figure 7.14. With more than 6000 requests, the ESCAT series has 46 seasons, marked by vertical lines, showing long computation times of more than 250 milliseconds between bursts of reads. These bursts are plotted very close to the x axis due to their short interarrival times (between 1600 to 3500 microseconds), as shown in Figure 7.15.

The first season, delimited by the first vertical line, is an initialization period during which no prediction is created. During initialization, the input queue of the RDI accumulated one season's worth of interarrival times, so seasonal differencing could start at the beginning of the second season. ARIMA Automodeler successfully predicts ESCAT's read interarrival times with average prediction accuracy around 95%.

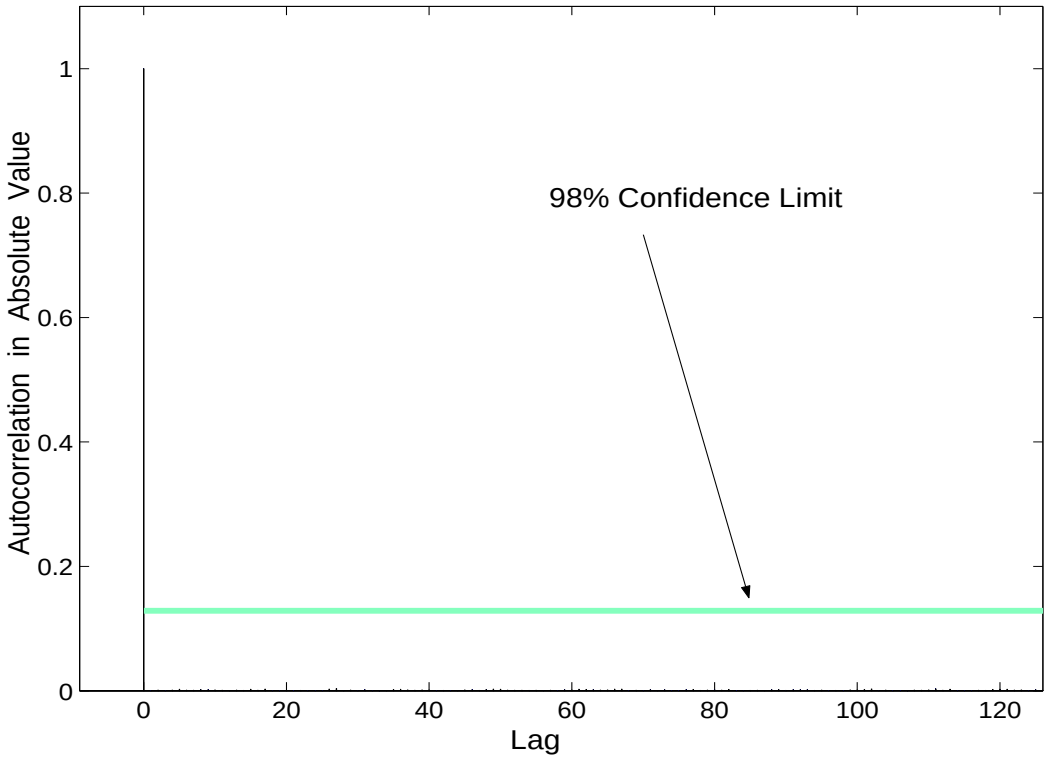

Figure 7.13: ESCAT ACF (after Seasonal Differencing) within the First Season

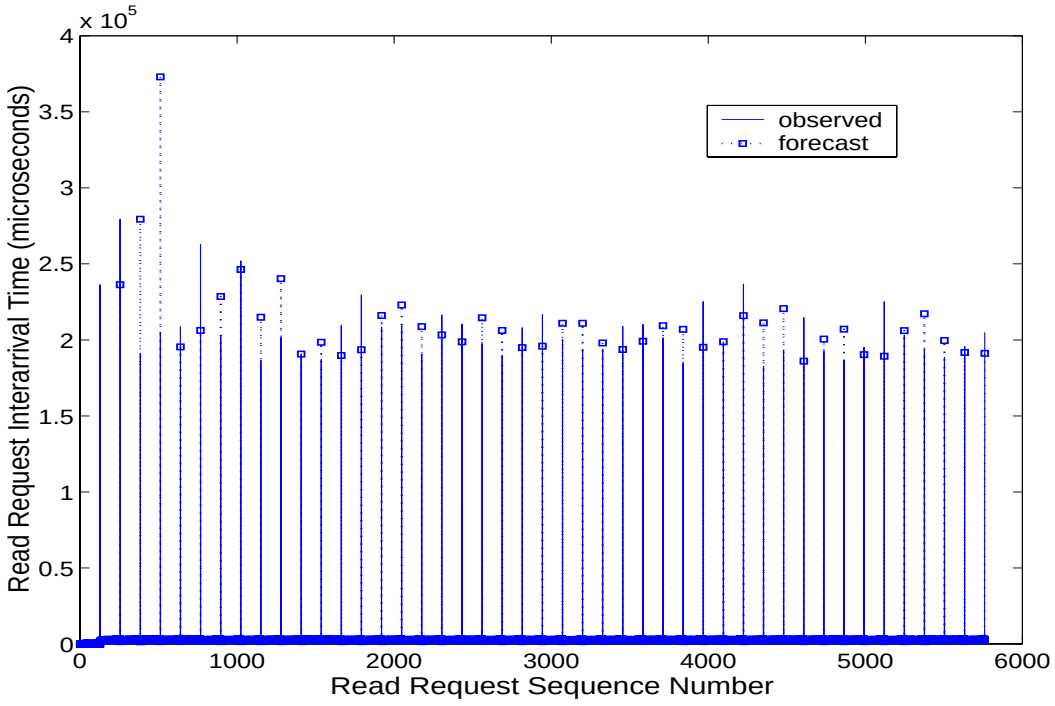

Figure 7.14: ESCAT General Distribution of Interarrival Time Predictions

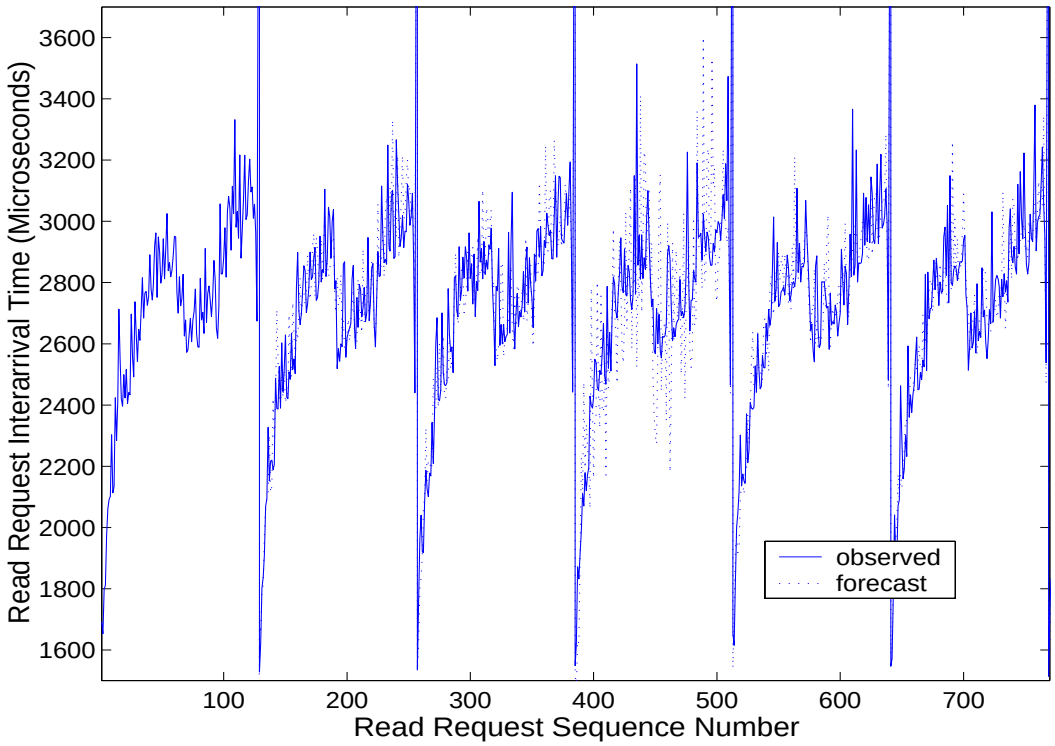

Figure 7.15: ESCAT Interarrival Time Predictions for the First 6 Seasons

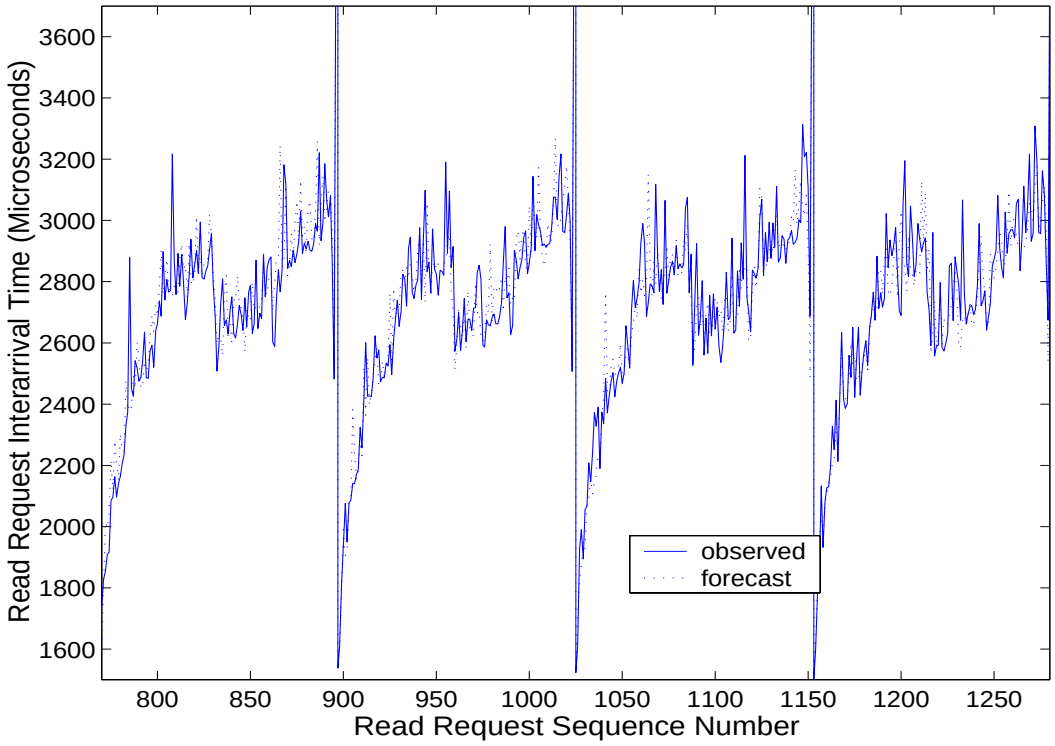

Figure 7.16: ESCAT Interarrival Time Predictions for Seasons 7 to 10

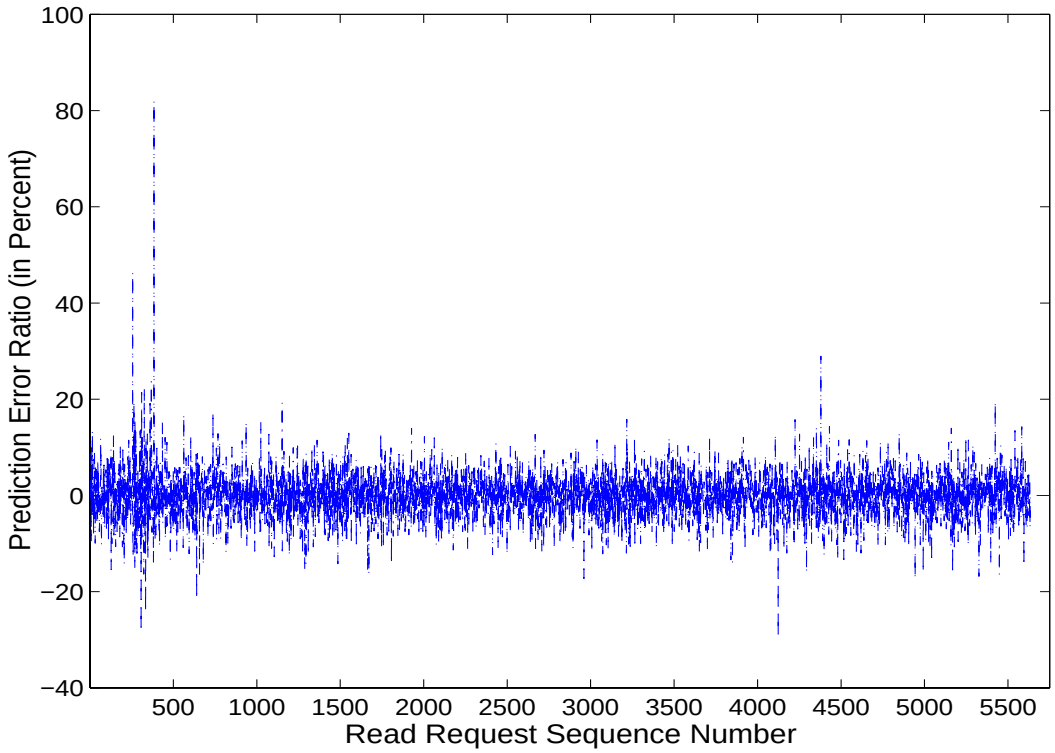

Figure 7.17: ESCAT Prediction Error Ratios

Figure 7.15 presents a detailed view of ESCAT's prediction pattern in the first six seasons, for 768 observations. Parameter estimation with seasonal differencing began at the second season. The 129th interarrival time was subtracted from the first interarrival time (saved in the RDI queue) to give the first seasonal difference. Seasons 2 to 6 correspond to a training period during which estimated parameters vary widely, resulting in noticeable prediction errors especially in season 4. However, starting from the seventh season, the average prediction accuracy reaches more than 90% and is maintained for the remaining seasons, a snapshot of which is shown in Figure 7.16.

The ESCAT prediction error ratios, plotted in Figure 7.17, indicate that the majority of predictions stay within a $\pm 5\%$ error band. The root mean square prediction error is 4.6%. These results demonstrate that Automodeler can successfully model and predict periodically bursty, non-stationary and seasonal I/O behaviors.

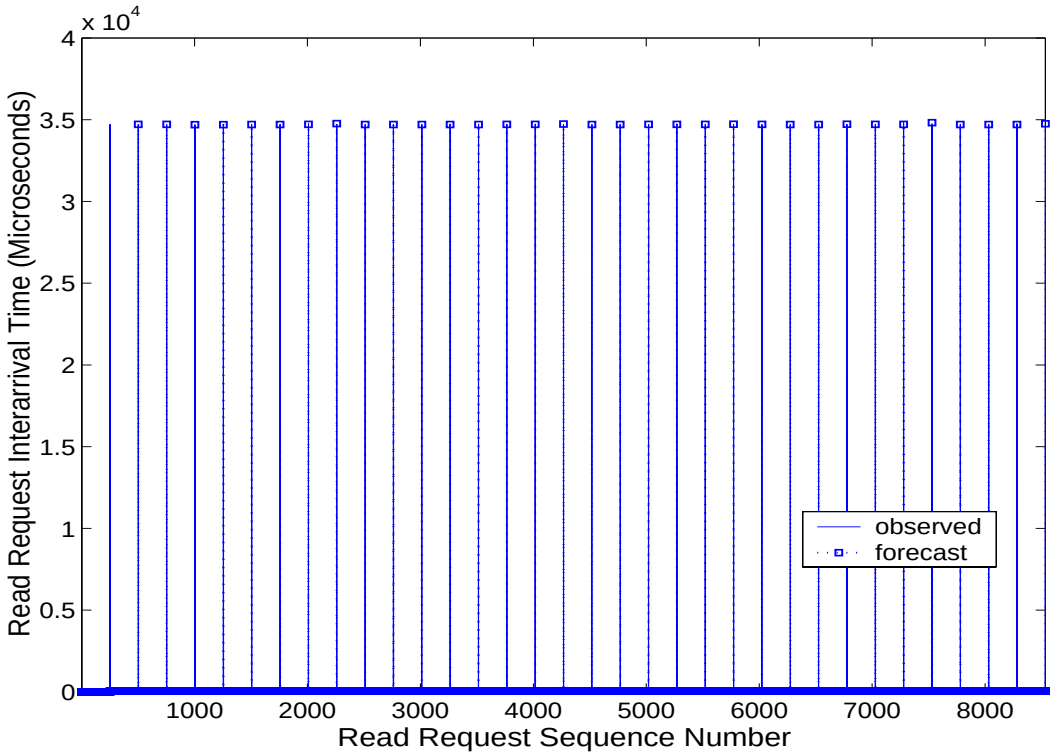

Figure 7.18: Synthetic Workload – General Distribution of Interarrival Time Predictions

7.3 Synthetic Workload for Read Request Interarrival Times

To validate modeling of I/O behavior *during application execution*, Automodeler was executed concurrently with a synthetic workload. Based on I/O traces and our experiments, we created a simple workload to simulate some commonly observed access patterns for read requests in scientific applications: seasonally bursty arrivals, high arrival rates within the seasons (i.e., short interarrival times), and small request sizes. We used nested loops to generate the workload seasonal behavior.

7.3.1 Read Interarrival Time Pattern

Our synthetic workload is characterized by periodic bursts of read requests, followed by long computation intervals. Read accesses have mixed sequential and sequentially strided patterns. Request sizes are variable and small (less than 100 bytes). Throughout the workload, groups of 250 requests, representing short bursts of reads with interarrival times between 54 to 60 microseconds, are interspersed with long computation intervals of approximately 35 milliseconds.

Using Automodeler to identify the workload's model structure, the read interarrival times ex-

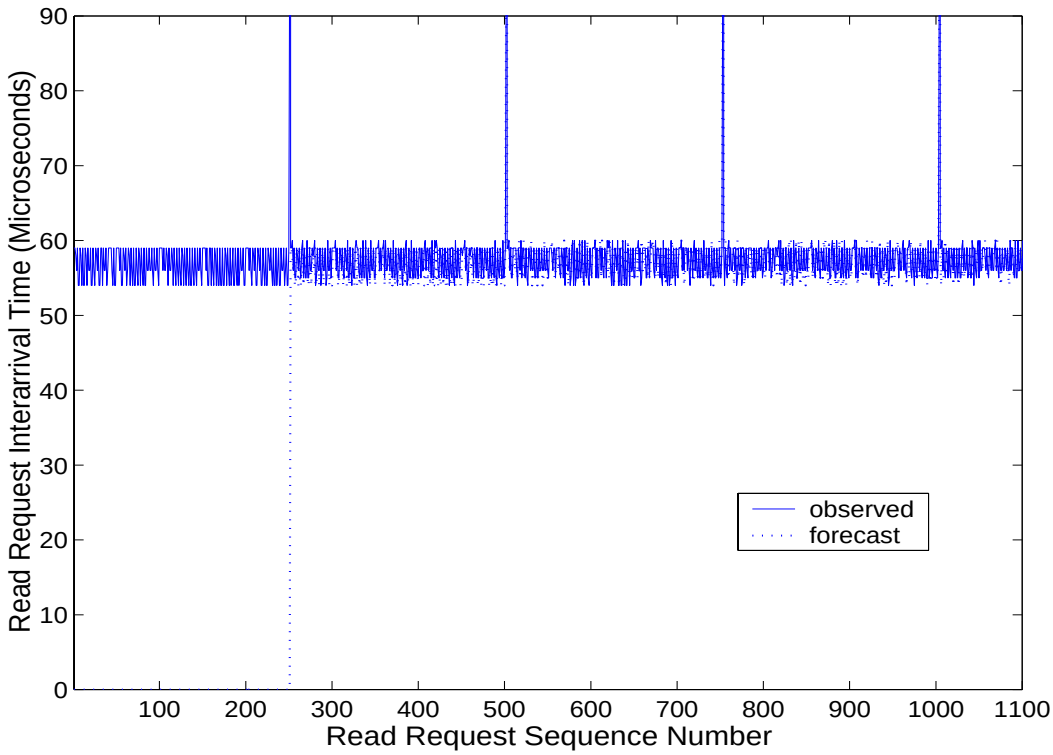

Figure 7.19: Synthetic Workload – Predictions for the First Four Seasons

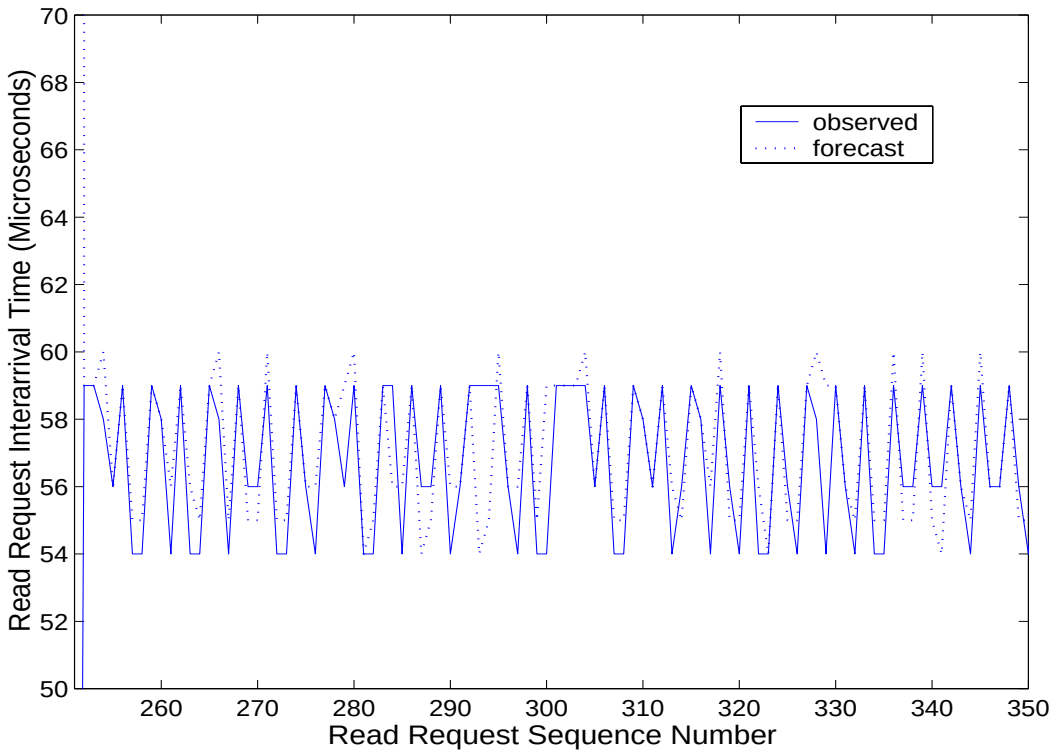

Figure 7.20: Synthetic Workload – Predictions Viewed at Close Range

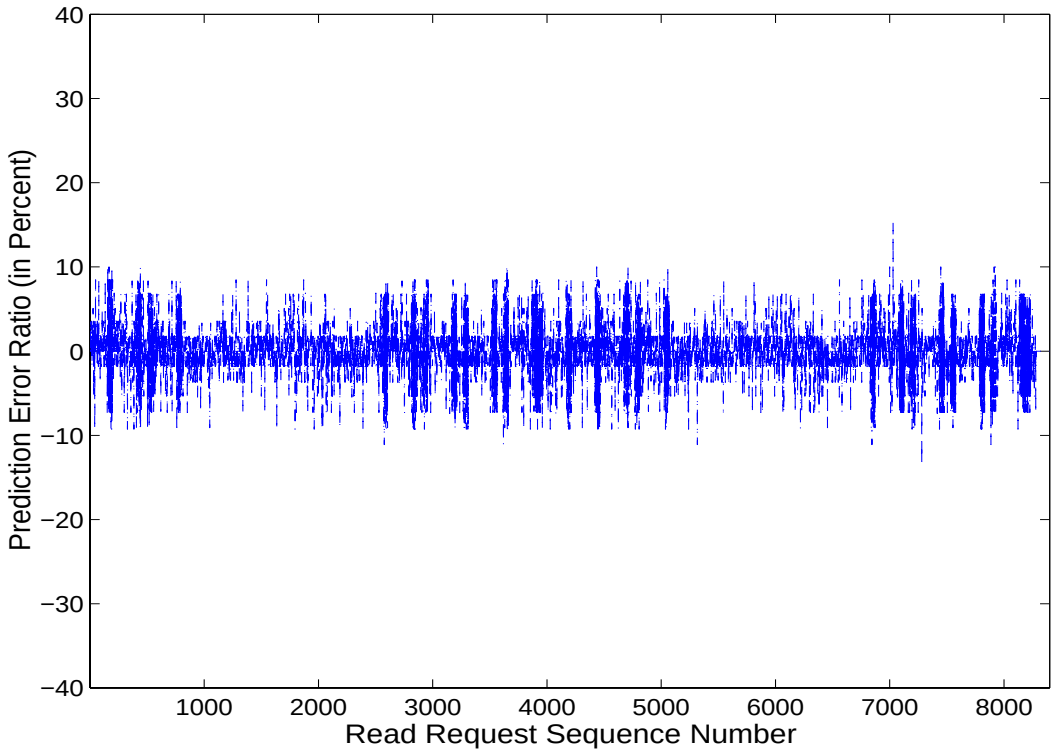

Figure 7.21: Synthetic Workload – Prediction Error Ratios

hibit a seasonal pattern with a season length of 251. One seasonal differencing level is required to transform the observations. The resulting stationary series has a single autoregressive element to be estimated. The identified model $(1,0,0) \times (0,1,0)_{251}$ suggests that interarrival times in the seasonally differenced series are correlated with only the previous differences.

7.3.2 Results and Analysis

Figure 7.18 compares the one-season ahead predictions (dotted lines and square markers) with actual interarrival times (solid lines) for the entire workload of more than 8500 read requests. This workload contains approximately 33 seasons, delineated by long vertical lines. The average prediction accuracy is 95%.

Figure 7.19 presents the overall temporal pattern of the first four seasons, spanning more than 1000 requests. Unlike ESCAT, the within-season read interarrival times in this workload form a stationary series, fluctuating around a mean of 57 microseconds. The first dotted line at observation 252 corresponds to the first prediction created after initialization. A closer view of the predictions,

presented in Figure 7.20, shows that they closely follow pattern variations in the series.

The prediction error ratios plotted in Figure 7.21 indicate that, with one exception, close to observation 7000, all errors are bracketed within a ±10% error band. The root mean square prediction error is 4.9%.

7.3.3 Request Interarrival Time Modeling Overhead

Because large overhead can diminish the effectiveness of a prefetching system, we need to assess the modeling cost. We distinguish two major cost components: model identification and parameter estimation. The latter includes the one-step ahead prediction cost.

7.3.3.1 Model Identification Cost

The dominant factor in determining model identification overhead is the amount of processing consumed by the *computations for the autocorrelations of a series and the confidence limit for these correlations*. Large samples used to identify seasonal series usually incur more overhead as the ACF and PACF involve every data point in the sample. Other costs associated with pattern recognition – the L1 and L2 distance tests, the lag count test, and the average rate of change test are relatively small when compared with the correlation computation cost.

Using ESCAT as an example because of its relatively large season length (128 read requests), a minimum of 5 seasons are needed to distinguish and recognize the frequently occurring patterns listed in Table 4.1. The total time taken by Automodeler to identify a seasonal model from a sample of 700 observations is around 113.99 milliseconds. Doubling the sample size almost quadruples the modeling time to 420.85 milliseconds. However, we will see shortly that the parameter estimation cost is more significant than the identification cost.

7.3.3.2 Parameter Estimation Cost

Figure 7.22 plots Automodeler's processing overhead against the number of estimated parameters. The top and bottom curves depict modeling time when executed with unoptimized and optimized versions of the newmat09 matrix library [16] respectively. Because more than 85% of the parameter estimation cost is attributable to matrix manipulations, using the optimized version reduces

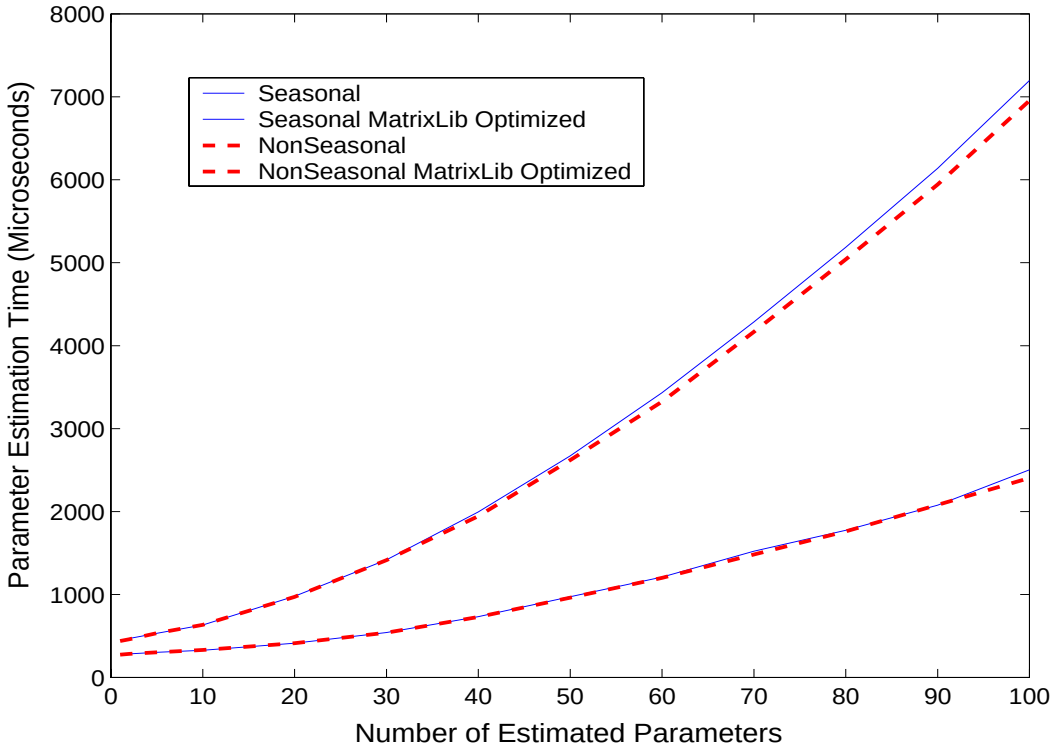

Figure 7.22: ARIMA Parameter Estimation Overhead

overhead by two-thirds, as displayed in the bottom curve.

The optimized modeling times increase nonlinearly with the number of estimated parameters, from 280 microseconds/estimation for two parameters to 2500 microseconds/estimation for 100 parameters. The small gap between the solid and dashed lines indicates that differences between modeling seasonal and non-seasonal behavior are negligible, given an equal number of parameters. These small differences arise from transformations of seasonal patterns, giving evidence of low overhead incurred by the RDI.

However, when compared with observed interarrival times, which can be as small as 10 to 20 microseconds per read request (e.g., the Cactus code), the optimized modeling overhead is still high — higher by one order of magnitude when two parameters are estimated. As a result, it cannot scale when a large number of requests arrive at high rates.

We propose *block downsampling* to amortize the overhead over several request arrivals. Instead of sampling every request, we sample every *block/group of requests* given a fixed block size. This technique matches the PPFS2 cache behavior. To serve a cache miss, PPFS2 retrieves data from

disks, in blocks of 1 KB or 4 KB for example. These are the same blocks that we use to group the requests. Downsampling via the block criterion avoids omitting critical information embedded in arrival patterns, necessary to obtain good predictions.

7.4 Synthetic Workload for Block Read Interarrival Times

Now, instead of modeling individual requests, we model read accesses grouped in fixed size of 1 KB blocks, using the same workload as the previous experiment (§7.3). Within a season length of 251, the first 250 requests, representing arrival bursts, can be mapped into five distinct blocks of 1 KB. Read accesses are sequential within each block, but sequentially strided across blocks. The $251st$ request, associated with a long computation interval, is mapped to another 1 KB block. In total, each season has six blocks.

The model structure $(1, 0, 0) \times (0, 1, 0)$ for the read interarrival times, sampled based on blocks, was identified using the autocorrelation functions. It is seasonal with a shorter season length of 6 blocks (instead of 251). The resulting seasonally differenced series is stationary with a single autoregressive element.

7.4.1 Results and Analysis

Figure 7.23 shows the general distribution of block interarrival time predictions for the entire workload of 198 blocks, spanning 33 seasons. Except the first season used for initialization, predictions are close to observed values for both the long computation times (top part of the plot) and the request arrival bursts (bottom part). The average prediction accuracy is about 93%.

Figure 7.24 presents a more detailed view of the arrival bursts and the associated predictions for the first 100 blocks. For legibility, the y axis is cut off at 6000 microseconds. Each season, delineated by an inverted bar, contained a burst of 5 blocks only, giving a very short stationary series. Each burst, in turn, was followed by a single access with a very long interarrival time (the y axis cutoff portion). Because 50 requests were grouped in a block, interarrival times for blocks, fluctuating between 3000 and 3300 microseconds, were longer than those for requests by about 50 times.

Using block downsampling, the number of interarrival time samples reduces 43 times, from 8534

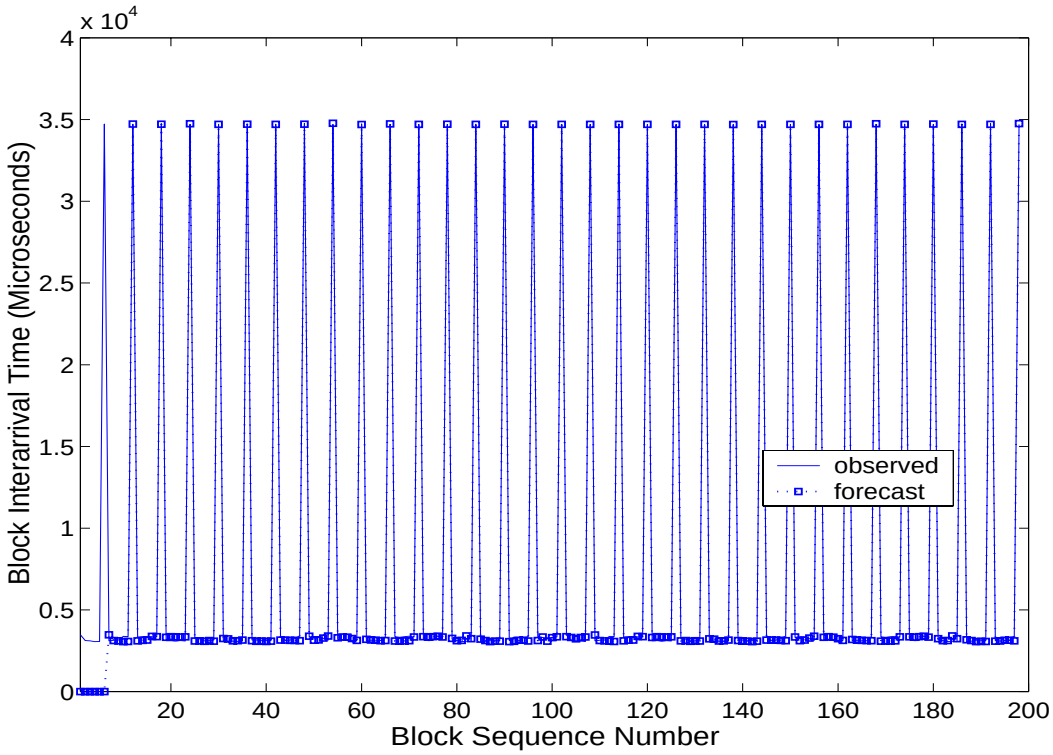

Figure 7.23: Synthetic Workload – General Distribution of Block Interarrival Time Predictions

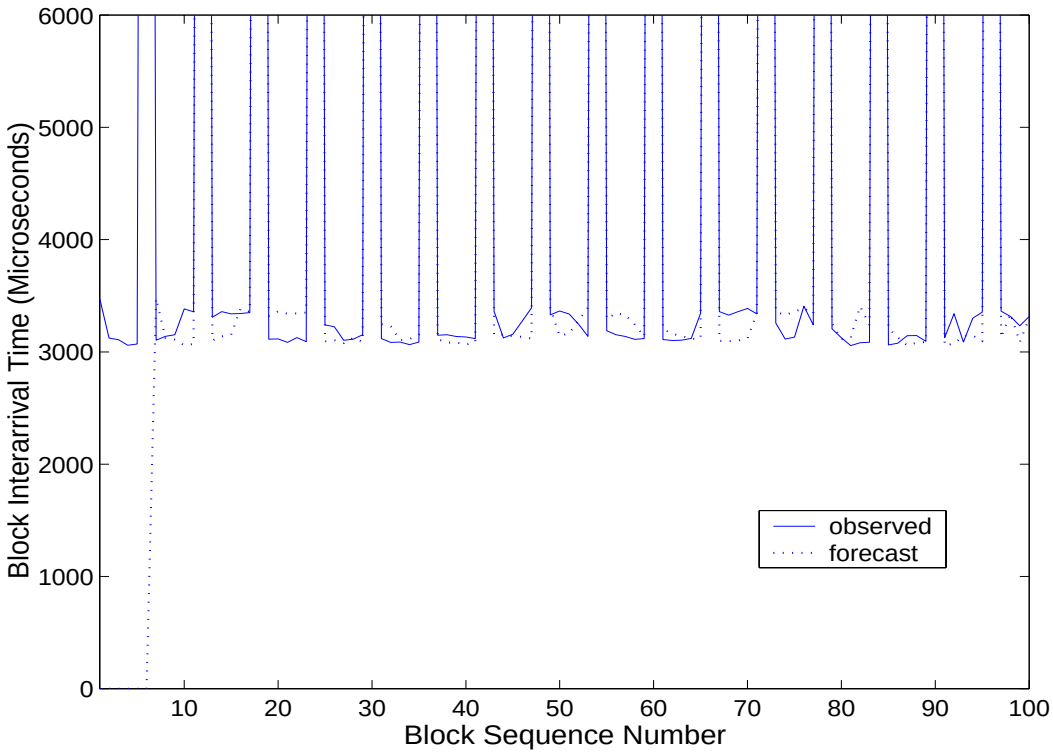

Figure 7.24: Synthetic Workload – Predictions for the First 100 blocks

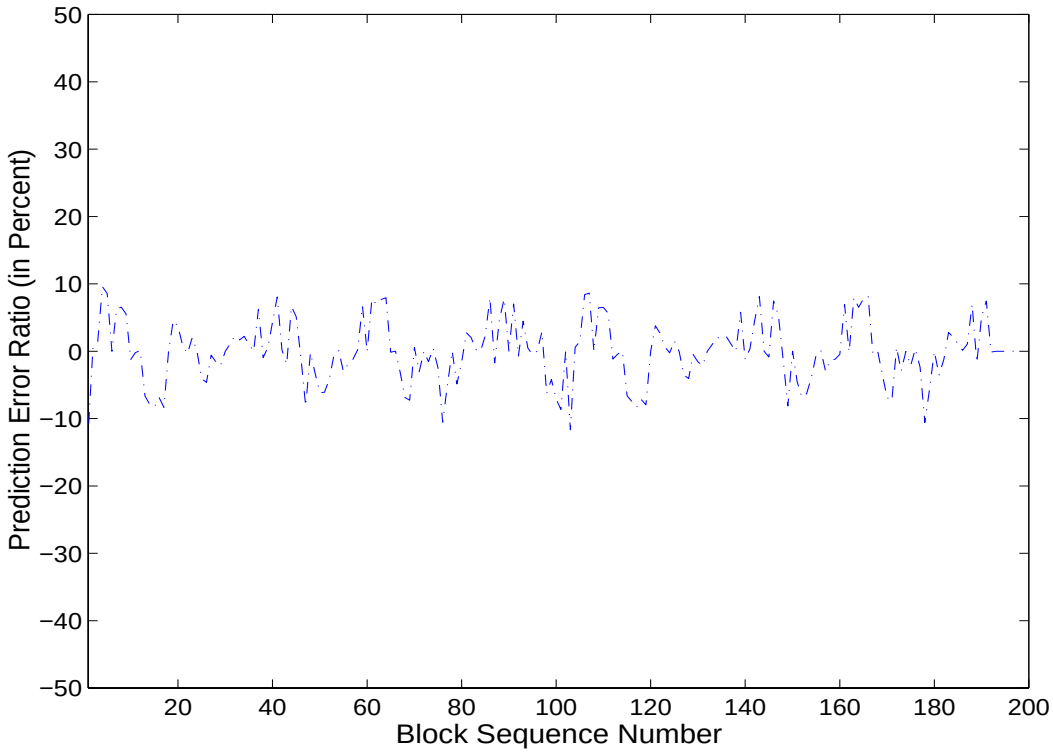

Figure 7.25: Synthetic Workload – Block Prediction Error Ratios

requests to 198 blocks. As a result, modeling overhead decreases proportionally from approximately 22900 to 55 milliseconds, about 40 times. Prediction error ratios, plotted in Figure 7.25, show that block downsampling does not compromise prediction accuracy, which is maintained at around 90% for the entire workload. All errors are confined within ±10% of observed values, although they include some remaining patterns. The root mean square prediction error is 6.7%.

7.5 A Computational Physics Application – Cactus Wavetoy

The remainder of this section describes experiments with a physics application, the Cactus code. Our objective in this section is to evaluate Automodeler's ability to *create online models and predictions* for read arrival patterns during scientific application executions.

The Cactus code [49] provides a platform for collaborative development of applications in numerical relativity and astrophysics. To share codes, users add their applications to the collaboration manager, the Cactus core. A shared application, known as a cactus thorn, can run on stand-alone machines, shared-memory machines, or PC clusters. User-supplied parameters, designed to allow

user control of application execution, include the number of iterations, the computation grid size, the granularity of simulation in the X-Y-Z dimensions, and the simulation variables. A large grid size increases the simulation data volume written to disks and the write request sizes. In our experiments, we used a grid size of $50 \times 50 \times 50$. We varied the number of iterations to produce different test scenarios.

The Cactus code supports several I/O libraries: IOASCII for 1-D data, IOFlexIO, and NCSA's HDF5 [59] for 2-D and 3-D data. IOFlexIO is based on NCSA's implementation of IEEEIO [50]. It is chosen for our experiments because of its robustness, portability, and support for 3-D data. We ported the IEEEIO library to the PPFS2 testbed, changing all Unix file system calls to PPFS2 Unix I/O API calls. Because the design of IOFlexIO follows a stateless approach, files are opened and closed in every iteration, creating large overhead in PPFS2. We solved this problem by modifying IOFlexIO to open and close files only once for all iterations.

7.5.1 Wavetoy I/O Behavior

Cactus exploits the IEEEIO library to support two major I/O modes: write only and read-write. In the latter mode, each iteration goes through a cycle: read, simulate (compute), and write. Our experiments used this mode to generate file read requests.

For each iteration, Wavetoy appends to the end of a file an *individual metadata header*, followed by simulation results produced at the end of the iteration. A *general header*, located at the first record in block zero, is also updated with new medadata for the *whole file*. Because the IEEEIO library does not keep state information, headers created during all previous iterations are reread when an iteration starts, and released when the iteration ends. The goal is to allow fast data retrieval from locations recorded in the headers, bypassing sequential data scanning.

In each iteration, read accesses are sequential within a header, consecutively interleaved between headers, followed by a single access to block zero for the general header update.

As the number of iterations increases, the application I/O intensity, dominated by header read requests, also increases. These requests arrive in bursts at high rates, followed by long computation cycles, creating bursty and seasonal arrival patterns.

7.5.2 Experiments on Wavetoy Read Interarrival times

To assess prediction accuracy and modeling overhead for Cactus read behavior, we modeled the interarrival times of Cactus Wavetoy's read requests using block downsampling. Wavetoy is a simple application to simulate 3-D wave computations. Application parameters included a $50 \times 50 \times 50$ grid size and 2000 iterations. Comprised mostly of record headers, Wavetoy's read requests are small and varied in sizes, ranging from 5 to 12 bytes.

As a result, we selected a small block size of 1 KB for block downsampling. PPFS2 cache modules measured the interarrival times of these 1 KB blocks of requests, which were subsequently used by Automodeler to predict future arrivals.

7.5.3 Wavetoy Interarrival Time Pattern

Our experiments reveal that Wavetoy's read interarrival time pattern has two types of seasonal variations: between iterations and within iterations. The between-iteration variations are associated with requests arriving at the end of each iteration. They correspond to the major seasons. Within each major season, there are minor seasons associated with all the requests in a given iteration. As the number of requests increases with the number of iterations, the lengths of the major seasons increase commensurately in an incremental fashion.

We have extended Automodeler to include support for this behavior by modeling two series simultaneously: one series for the major season (major series), the other for the minor season (minor series). In general, incremental season lengths do not occur frequently in I/O access patterns.

High Frequency Location (lag)	0	46	94	144	195	248	303	359	417	477
	538	601	666	733	802	873	946	1021	1097	1175
L-1 Distance		46	48	50	51	53	55	56	58	60
	61	63	65	67	69	71	73	75	76	78
L-2 Distance			2	2	1	2	2	1	2	2
	1	2	2	2	2	2	2	2	1	2

Table 7.1: Detection of Incrementally Seasonal Behavior in Wavetoy

7.5.4 Automatic Model Structure Identification

The Haar wavelet transform provides one effective method to detect incrementally seasonal behavior in Wavetoy's read interarrival times. Table 7.1 shows the first 20 locations of the high frequency components identified by Automodeler through the Haar wavelet detail coefficients. Figures 7.26 and 7.27 plot the first 250 coefficients, corresponding to the first 500 observations. All the high frequency signals are above the 95% confidence limit of 24643.76. Because of wavelet differencing, the locations of these signals occupy half of the actual locations.

The L-2 distance test, performed on the L-1 distances of these locations, exposed a majority (14 out of 18) season increment of 2. The first season, starting from lag 0, has a season length 46 (the first L-1 distance associated with the first detected season increment). The increment 2 indicates that, most of the time, the added read requests span 2 data blocks. Occasionally, some requests can fit into a single block, resulting in an increment of 1.

The major series, made up of all the high frequency components at season boundaries, has ACF and PACF displayed in absolute values on Figures 7.28 and 7.29. The 95% confidence limit on both plots is around 0.10. An exponential decay pattern in the ACF was detected by the lag count test. The average rate of decay test, applied to the PACF, suggested an abrupt decay pattern. As a result, the major series is simply an AR(1) autoregressive process. Its season length corresponds to the length of the associated iteration, which increases incrementally.

On the other hand, the minor series is more complicated, displaying seasonal behavior. A season length of 40 blocks was identified via ACF (see Figure 7.30), which was computed on the minor series after it was regularly differenced once. From this ACF, Automodeler found high frequency signals at lags $0, 5, 10, 35, 40, 80, 120$. Applying the L-1 distance test $(5, 5, 25, 5, 40, 40)$ on these correlations gave a season length 40 for the minor series (the majority consists of three 5's and two 40's).

Automodeler seasonally differenced the minor series with a length 40. This operation removed all significant correlations at the *season boundaries* (lags 40, 80, 120), as illustrated by the ACF in Figure 7.31. For this reason, Automodeler assigned a structure $(P, D, Q) = (0, 1, 0)_{40}$ to the *seasonal component* of the minor series. For the *non-seasonal component*, the correlations *within the seasons* must be examined. Figures 7.32 and 7.33 show the ACF and PACF in the first 40

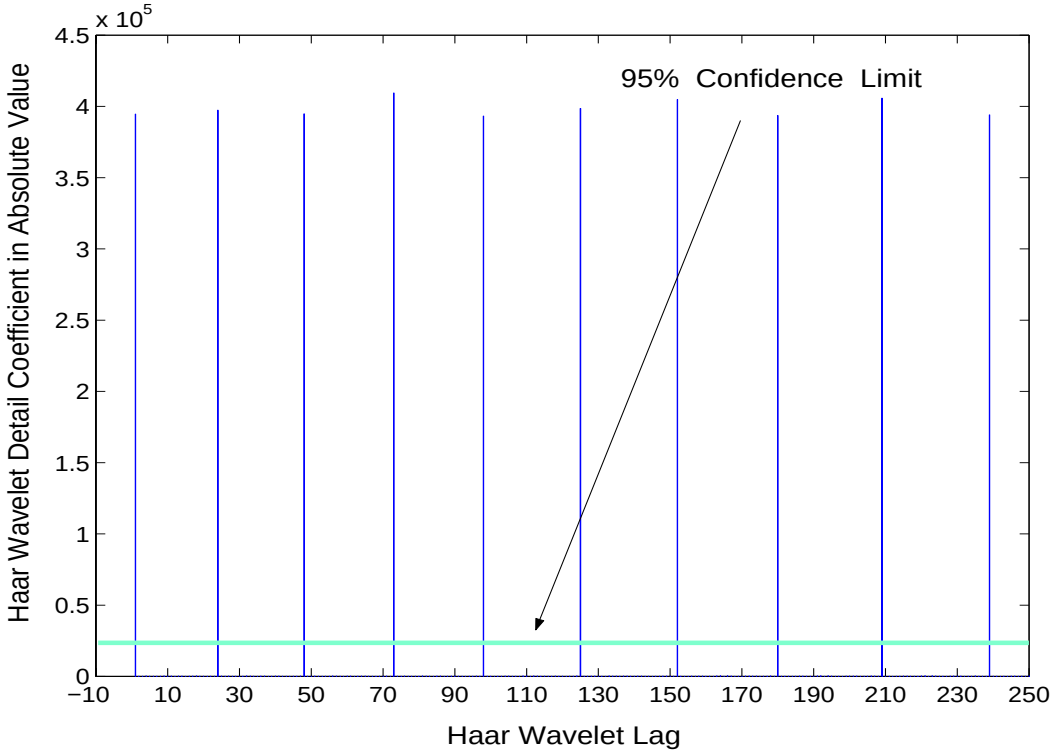

Figure 7.26: Wavetoy Detail Coefficients

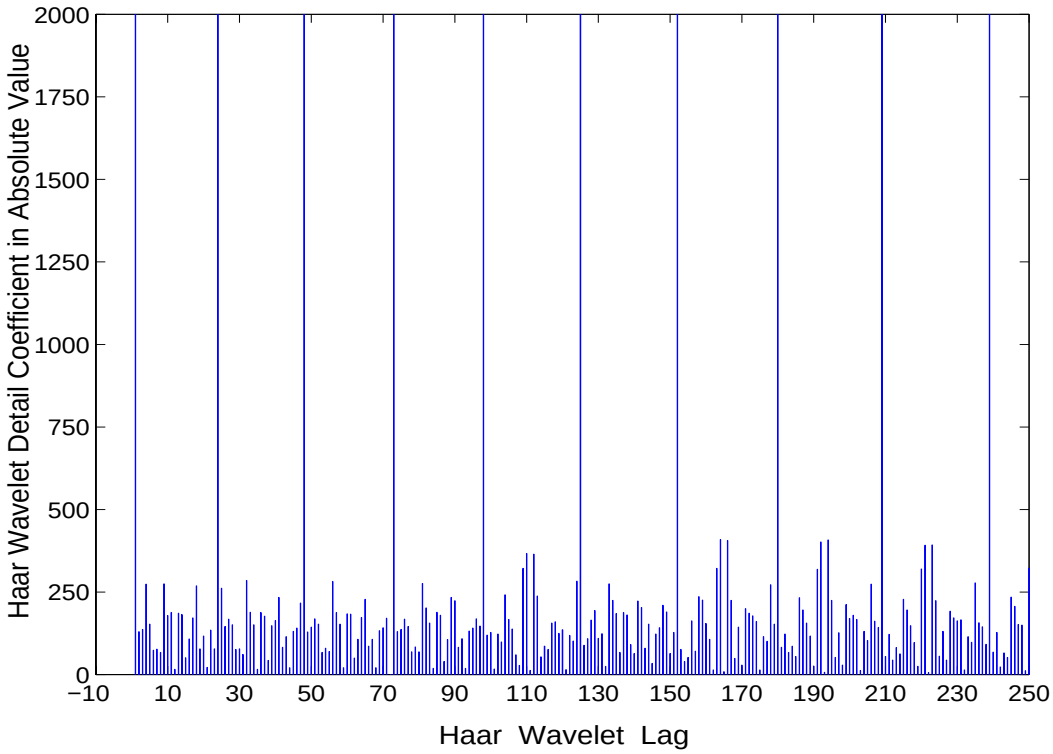

Figure 7.27: Wavetoy Detail Coefficients Viewed at Close Range

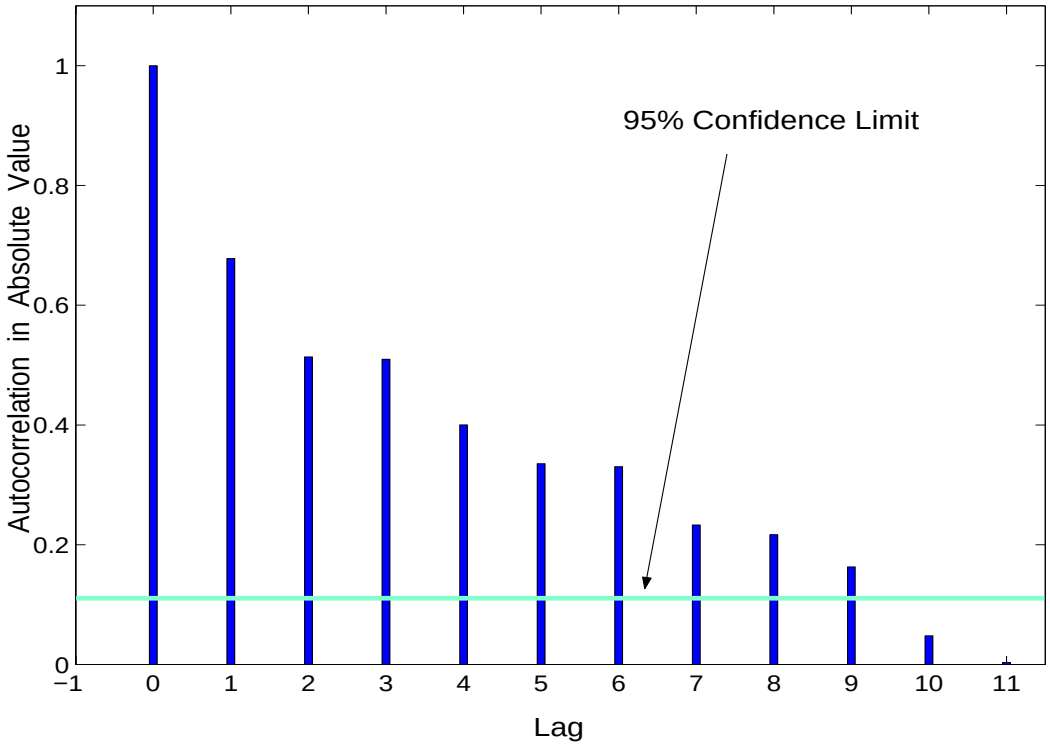

Figure 7.28: Wavetoy ACF of the Major Series

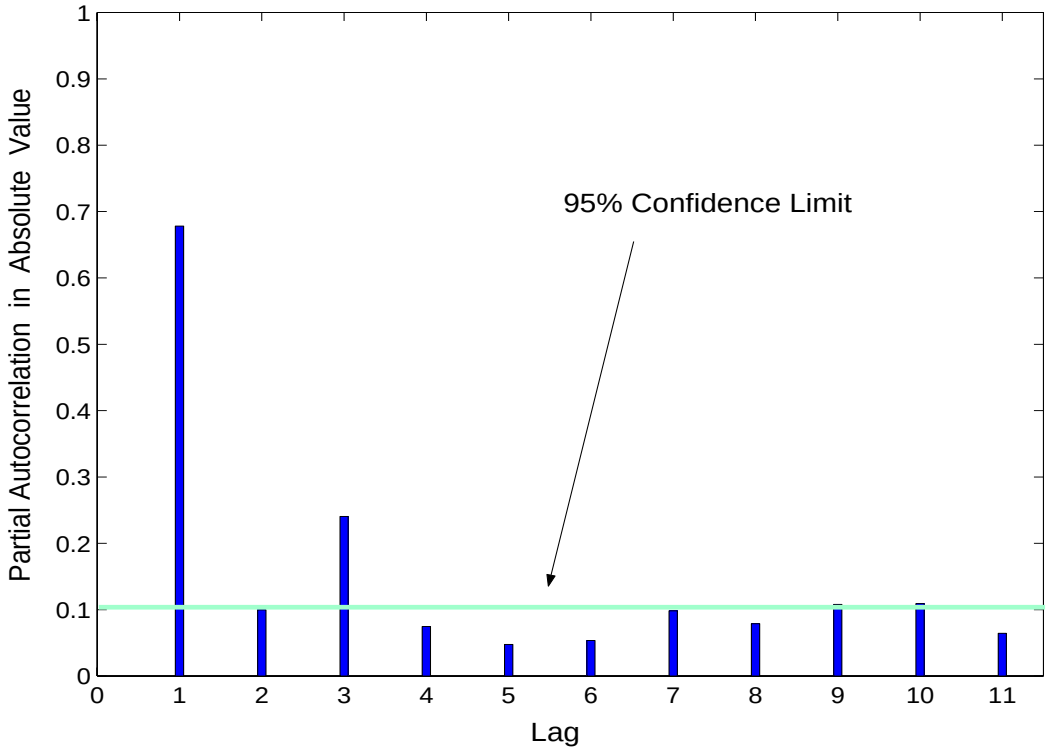

Figure 7.29: Wavetoy PACF of the Major Series

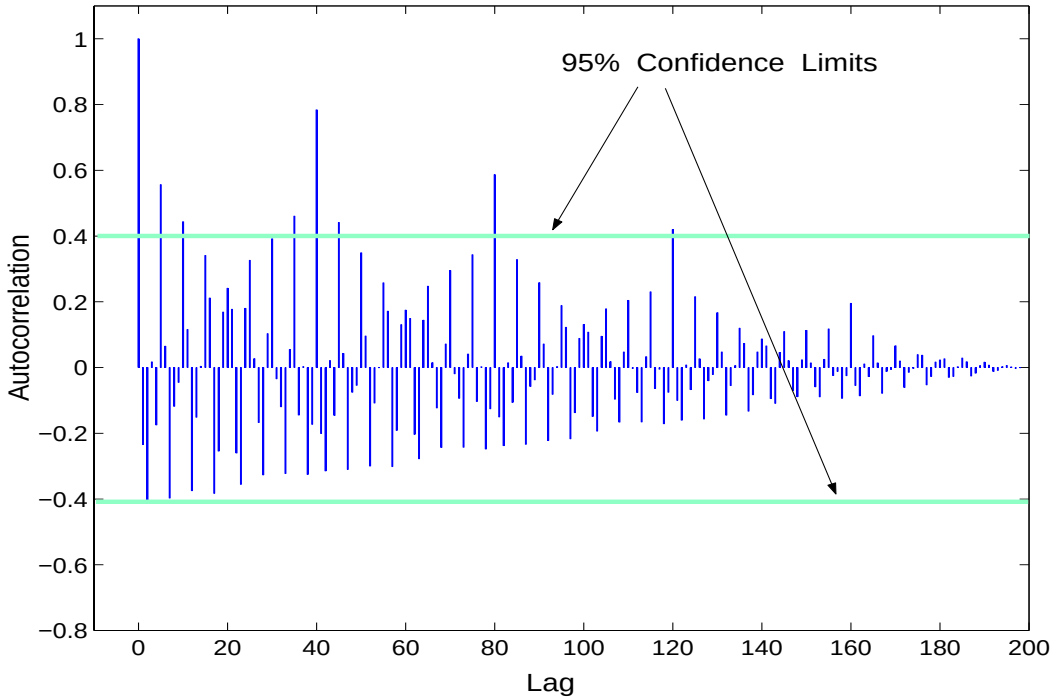

Figure 7.30: Wavetoy ACF of the Minor Series - after Regular Differencing

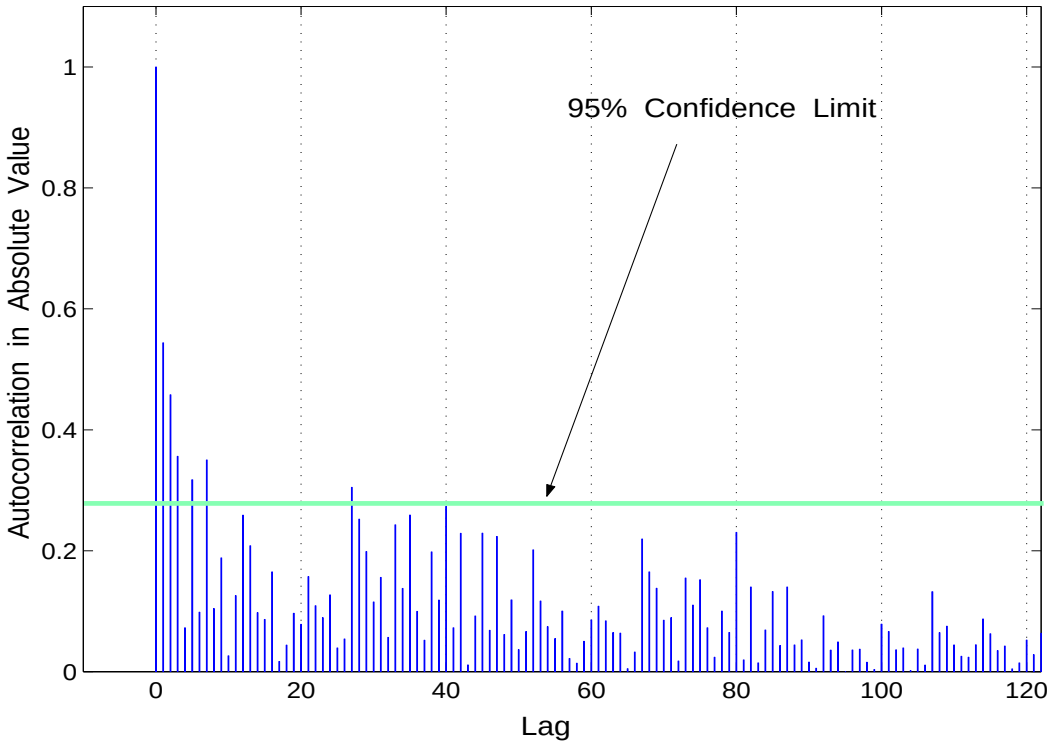

Figure 7.31: Wavetoy ACF of the Minor Series - after Seasonal Differencing

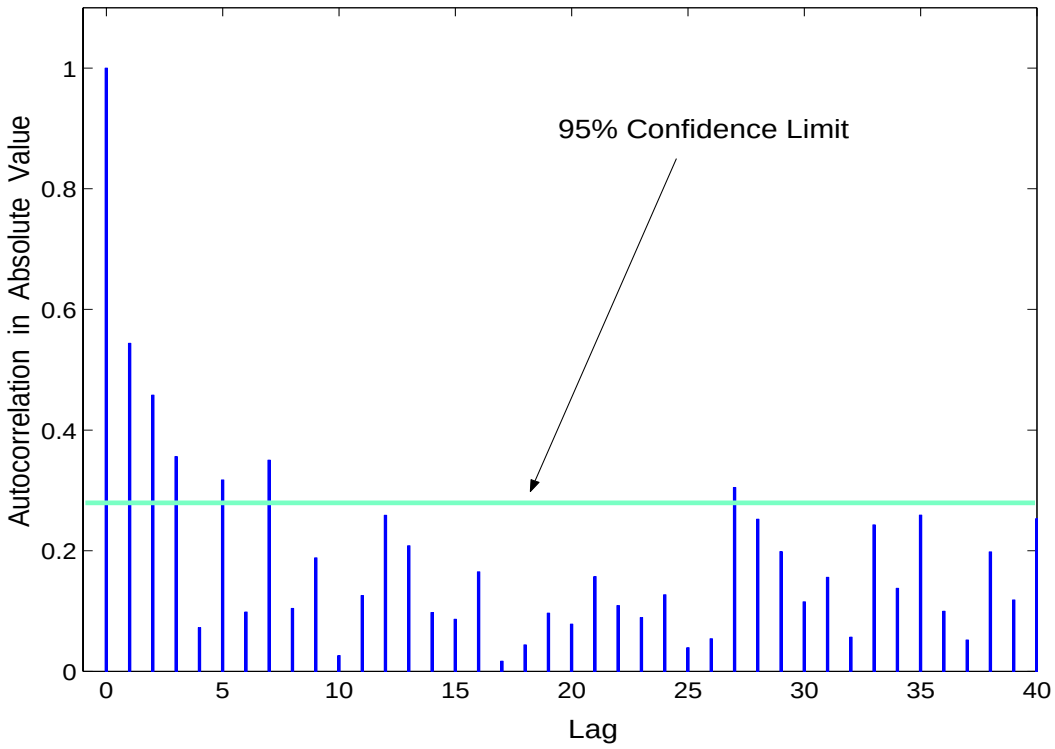

Figure 7.32: Wavetoy ACF of the First Season in the Minor Series (after Seasonal Differencing)

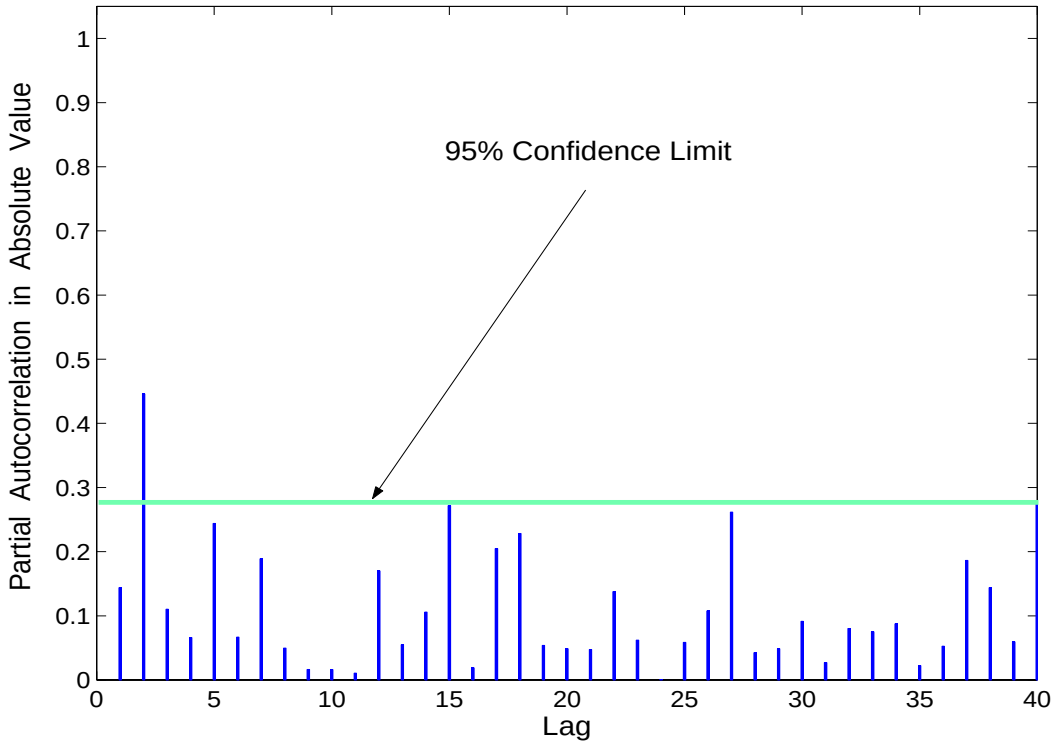

Figure 7.33: Wavetoy PACF of the First Season in the Minor Series (after Seasonal Differencing)

lags of the seasonally differenced series. The abrupt cutoff pattern in PACF at lag 2 suggested an AR(2) process.

To avoid over-parameterization, a common practice in time series analysis is to first model with the smallest number of parameters (i.e., AR(1) process). Only when prediction errors from this model are unacceptable that one selects a larger number of parameters. As a result, Automodeler chose the combined model $(1,0,0) \times (0,1,0)_{40}$ for the minor series.

7.5.5 Results and Analysis

Figure 7.34 compares one-season ahead predictions with observed interarrival times. For legibility, we plot only the first 1600 blocks of read requests. The plot shows seasonal behavior, marked by spikes of long interarrival times (more than 400 milliseconds) at the end of each iteration. These spikes, delimiting the major seasons, are interspersed with bursts of reads having very short interarrival times. The gradual widening in the horizontal spaces between the spikes demonstrates the incremental increases in the major season lengths. This incremental pattern is correctly predicted by Automodeler, with an average prediction accuracy of 87%.

Figure 7.35 presents the detailed pattern in the minor seasons for the first 300 blocks. Interarrival times for the short bursts of requests fluctuate between 50 and 450 microseconds, exhibiting wave-like patterns that persist throughout the entire execution. As the number of blocks in a major season becomes sufficiently large, the existence of several minor seasons within a major season becomes evident, as demonstrated in Figure 7.36. The ARIMA predictions successfully tracked the bursty pattern in the minor seasons. The root mean square prediction error is 12.9%. The prediction error ratios, plotted in Figure 7.37, indicate that most errors are confined within an error band of $\pm 15\%$.

7.5.6 Model Structure Simplification

A closer examination of the behavior of Wavetoy read requests that took place *within the iterations* indicates that the block interarrival times varied in a rather small range – from 50 to 450 microseconds. From the perspective of prefetching for bursty I/O, forecasts for interarrival times that exhibit small variations do not need to be very accurate as long as they follow the overall

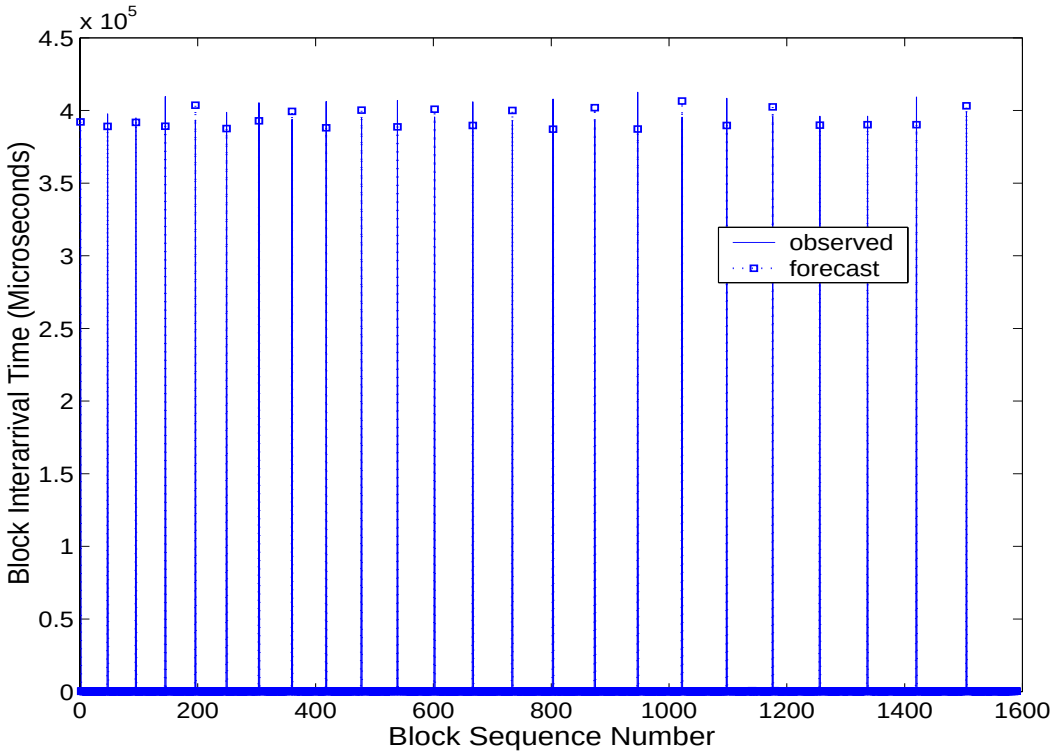

Figure 7.34: Wavetoy General Distribution of Interarrival Time Predictions

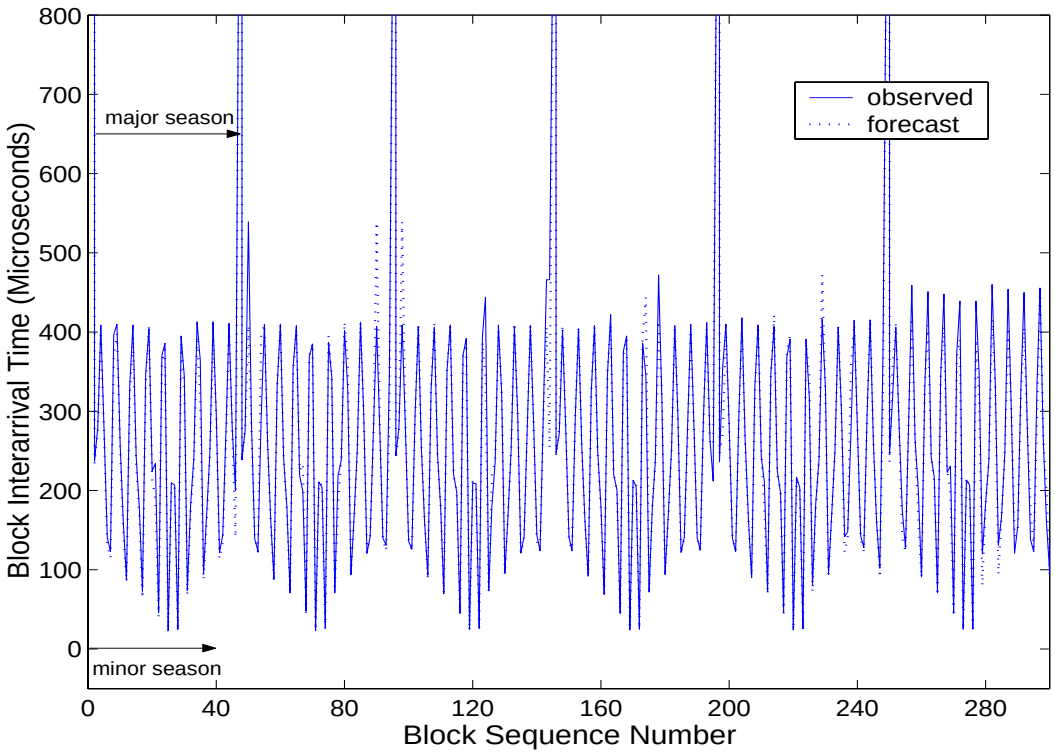

Figure 7.35: Wavetoy Interarrival Time Predictions for the First 300 blocks

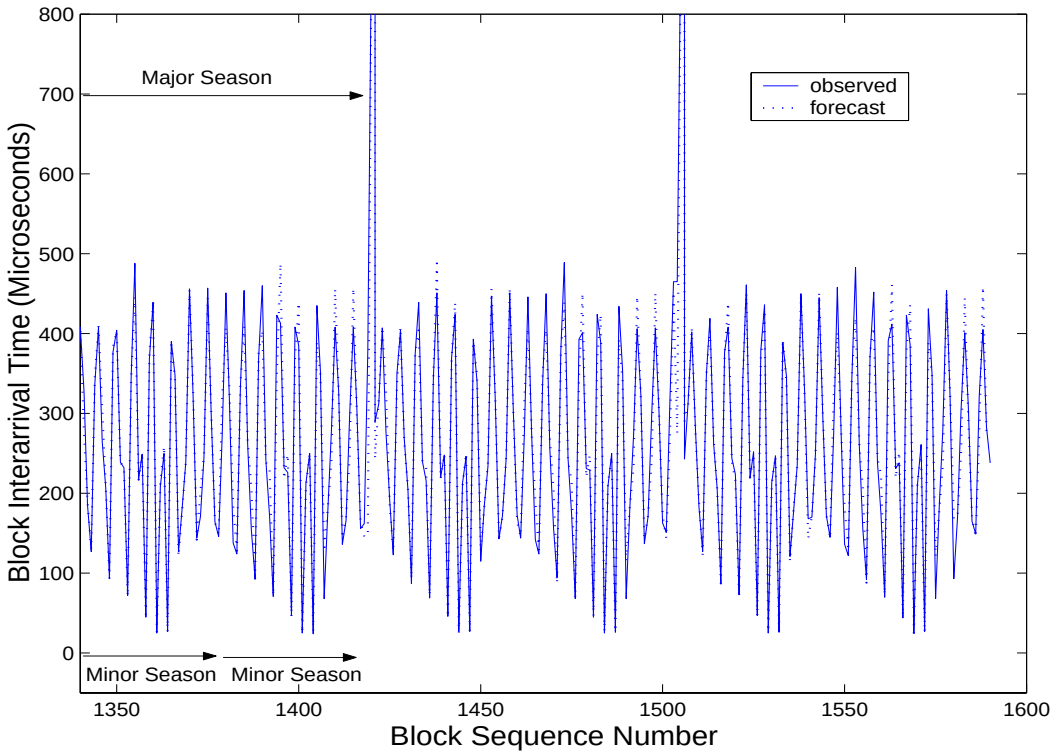

Figure 7.36: Existence of Several Minor Seasons within a Major Season

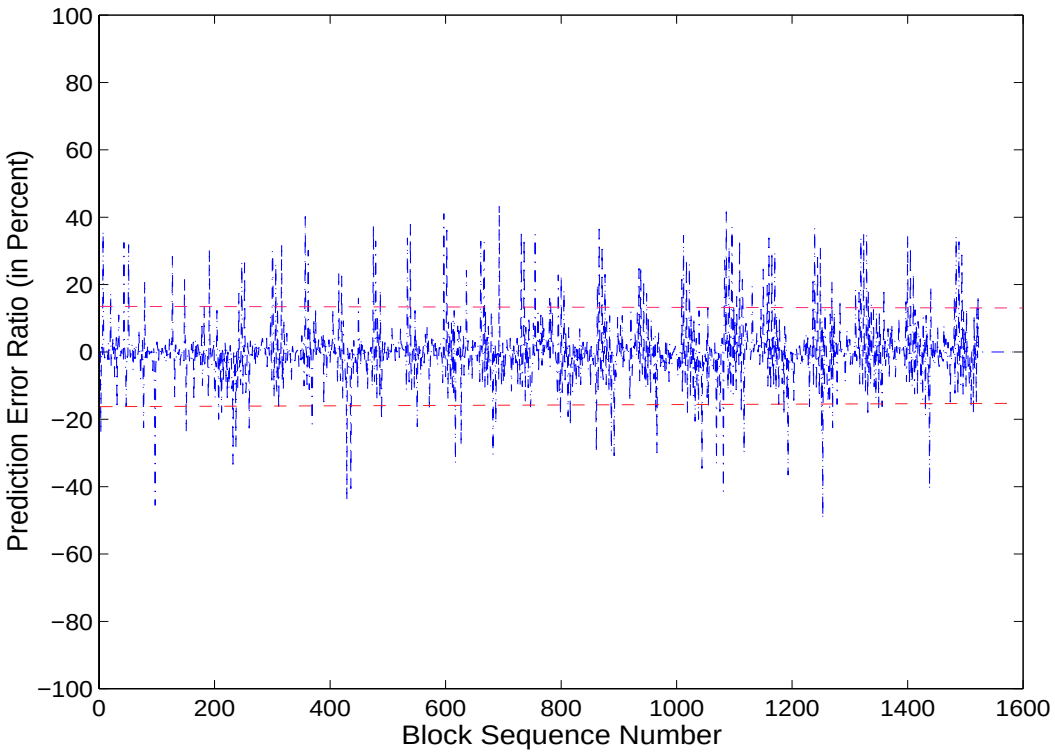

Figure 7.37: Wavetoy's Prediction Error Ratios

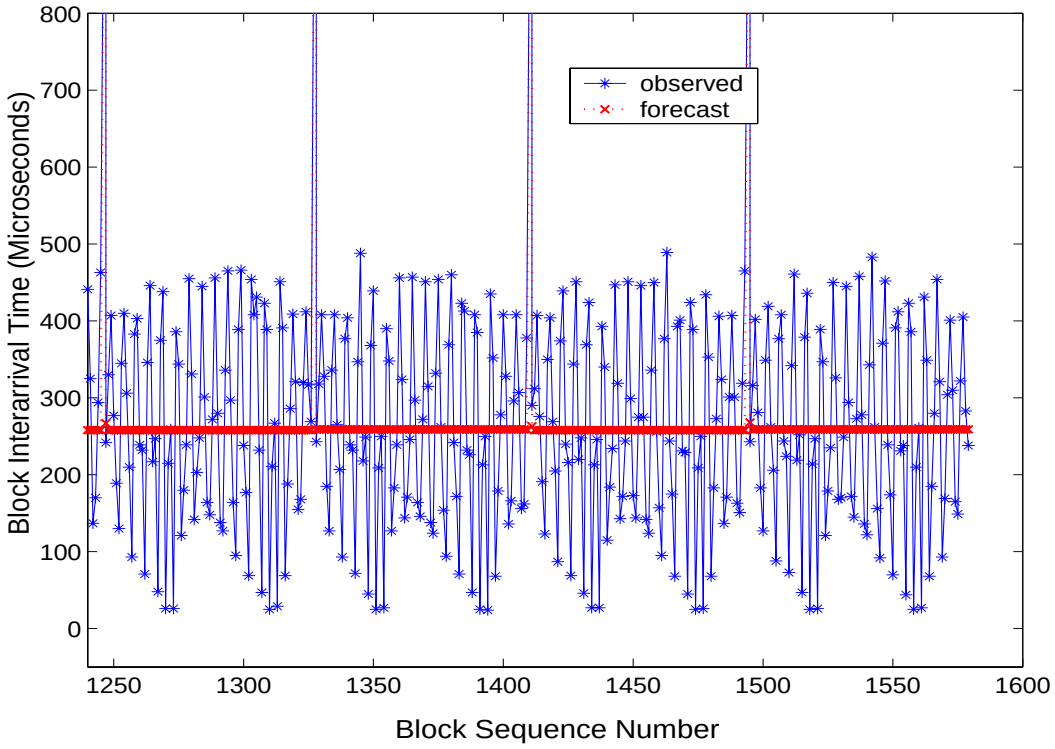

Figure 7.38: Wavetoy Predictions Using a Simple Model

I/O pattern. In these circumstances, using simple linear models to approximate the general I/O behavior can further reduce modeling overhead. In searching for simple models, we observe that the AR(1) process

$$y(t) = a_0 + a_1 y(t-1) + e(t)$$

is in fact a line equation of the form $y(t) = a_0 + a_1 x(t)$, where the noise term $e(t)$ is assumed to be drawn from a Gaussian distribution $N(0, \sigma^2)$ and thus to have a zero mean.

For this reason, in the Wavetoy application, instead of the seasonal model $(1,0,0) \times (0,1,0)_{40}$, we can use AR(1) to model data within the iterations (i.e., the minor series). The constant a_0 and the slope a_1 are automatically estimated by the extended least square algorithm during program execution, based on the most recently collected read interarrival times. At the end of each iteration, Automodeler generates two types of forecasts for one iteration ahead: those for data located within the *next iteration* using the simple model AR(1), and those for data at the *season boundaries* (i.e. the major series) using the detected model – which happens to be also AR(1).

To demonstrate how to compute the one-iteration ahead forecasts for the minor series, let us

focus on iteration 20. At the end of this iteration (block sequence number = 1254), Automodeler predicted the next iteration's block interarrival times using AR(1). The model parameters, estimated by ELS, were $a_0 = 246.21$, $a_1 = 0.046$. Based on the latest interarrival time $y(t) = 463$ and the white noise assumption, the n-step ahead forecasts $\hat{y}(t+1) \ldots \hat{y}(t+n)$ were bootstrapped from the 1-step ahead:

$$
\begin{aligned}
\hat{y}(t+1) &= a_0 + a_1 y(t) + e(t+1) &&= 246.21 + (0.046 \times 463) + 0 &&= 267.508 \\
\hat{y}(t+2) &= a_0 + a_1 \hat{y}(t+1) + e(t+2) &&= 246.21 + (0.046 \times 267.508) + 0 &&= 258.515 \\
\hat{y}(t+3) &= a_0 + a_1 \hat{y}(t+2) + e(t+3) &&= 246.21 + (0.046 \times 258.515) + 0 &&= 258.102 \\
\hat{y}(t+4) &= a_0 + a_1 \hat{y}(t+3) + e(t+4) &&= 246.21 + (0.046 \times 258.102) + 0 &&= 258.083 \\
\hat{y}(t+5) &= a_0 + a_1 \hat{y}(t+4) + e(t+5) &&= 246.21 + (0.046 \times 258.083) + 0 &&= 258.082 \\
\hat{y}(t+6) &= a_0 + a_1 \hat{y}(t+5) + e(t+6) &&= 246.21 + (0.046 \times 258.083) + 0 &&= 258.082 \\
&\ldots
\end{aligned}
$$

The n-step ahead forecasts converge towards an average value around 258. In fact, this convergence property holds for all stationary series. The dark line segments in Figure 7.38 illustrate Wavetoy's forecasts between blocks 1250 and 1570. They cut across the observed interarrival times at about mid-point, showing that the AR(1) model can indeed approximate Wavetoy's general read behavior.

7.6 Summary

Using two I/O traces, one synthetic workload and one scientific workload, we have successfully validated the main functionalities of Automodeler: model identification, parameter estimation, and prediction. Our time series modeler successfully identified model structures from application data captured at run time. These structures determine the number and types of parameters to be estimated. In all of our experiments, predictions were derived from parameters estimated in real-time with overall accuracy above 80%.

Block downsampling, grouping several requests into blocks, is an effective approach to reduce overhead. The blocking factor can be manipulated to control the degree of reduction. On the other hand, model structure simplification scales down the number of computations required for parameter estimation, trading accuracy for smaller overhead. Predictions generated from simple models can only approximate the general patterns in interarrival times series. However, in the next chapter, we will experimentally demonstrate that these approximations are sufficient and proven to be useful for prefetching purposes.

Chapter 8

Prefetching Experiments

In this chapter, we continue experimenting with the Cactus Wavetoy application to evaluate the effectiveness of our prefetching system in improving application-level I/O performance. Table 8.1, summarizing Wavetoy's read/write activities in 1000 iterations, reveals that the I/O workload is dominated by read requests. Over 96% of all block accesses are reads, accounting for 86% of total I/O time, but only 22.4% of the total data volume accessed. This intense read activity, disrupted by periodic, long computations, creates bursty I/O. Bursty patterns, together with small request sizes (5 to 12 bytes), provide opportunities for data aggregation, caching, and prefetching.

Our evaluation of the prefetcher is based on two criteria: improvement in application execution times and reduction in demand fetch cache misses. To explore the impact of various hardware configurations on prefetching performance, we conducted experiments on three PC clusters, equipped with different processor types, memory sizes, and disk characteristics.

Section §8.1 briefly introduces the various hardware platforms. It is followed by descriptions in §8.2 of specifications for the different parameters used in the experiments. In §8.3, we present the evaluation of Wavetoy's performance on a high-speed, 32-node cluster. Our evaluation considers the cumulative effect of prefetching, caching, and striping. Finally, in §8.4, we compare Wavetoy's performance among the three clusters.

Operation	Block Access Count	Percentage Count	I/O Time (Seconds)	Percentage I/O Time	Volume (Megabytes)	Request Sizes (Bytes) Minimum	Maximum
read	26040000	96.37	80.65	85.97	275.57	5	12
write	979979	3.63	13.17	14.03	955.43	16	991844
all I/O	27019979	100.00	93.82	100.00	1231.00	5	991844

Table 8.1: Wavetoy Read Write Summary (1000 Iterations)

8.1 Experimental Hardware Platform

All PC clusters used in our experiments were equipped with Redhat Linux 6.2 operating system, providing a consistent environment for test result comparisons. We distinguish an 8-node, 16-node and 32-node cluster. The 8-node cluster was described in Chapter 7 with Automodeler's validation experiments. It has slower processors, smaller memory capacities, and narrower disk bandwidths than the other two clusters.

8.1.1 16-Node Cluster

Using a setup similar to the 8-node cluster, this cluster has one front-end server and 16 PCs, all connected to a 100 Mb/second, switched, fast ethernet network, in addition to a Myrinet [5] full crossbar switch with 20.48Gb/second aggregate peak bandwidth. Each PC includes a Pentium II 450 MHz processor, 256 MB main memories, one system disk, and four storage disks with 9 GB each. The storage disks have 12 milliseconds full stroke seek time and 6 milliseconds average access latency. They are linked to an Ultra2 SCSI adaptor with 80MB/second peak bandwidth.

8.1.2 32-Node Cluster

Comprised of one front-end server and 31 PCs, this high-speed dual-processor cluster is connected to two types of networks: a 100 Mb/second, switched, fast ethernet network and a gigabit network. The gigabit interconnect includes three switches with 16 ports each – 12 for copper links and 4 for fiber links.

Fastest among the three clusters, each PC has a Pentium III 930 MHz processor and 896 MB main memories. The two storage disks in each machine are connected to an 80 MB/second peak bandwidth SCSI adaptor. Each disk supplies 18 GB of storage, at 12 milliseconds full stroke seek time, and 3 milliseconds average access latency.

8.2 Experiment Parameter Specifications

The Wavetoy application, the Automodeler, the adaptive prefetcher, and the cache modules in PPFS2 were launched concurrently in our prefetching experiments. To stress test the prefetcher

under heavy I/O workload, we gradually increased Wavetoy I/O intensity by varying the number of iterations from 200 to 2000 in increments of 200. PPFS2's cache was configured with fixed size blocks of 4 KB.

As described in §7.5.4, Automodeler identified incrementally seasonal behavior in Wavetoy's read interarrival times. It built a one-parameter pure autoregressive model for data between the iterations (major series). It also built a *simple AR(1) model* to linearly approximate data (see Figure 7.38) located within the iterations (minor series). This simplification bypassed computations required for seasonal differencing and estimation of the parameters in the seasonal components.

At the *end of each iteration*, both spatial and temporal pattern predictions were created from the ARIMA and Markov models. These predictions were used to compose prefetch schedules based on a variant of the Forestall algorithm [60], as explained in Chapter 6.

In Wavetoy, the number of data blocks that arrived within an iteration increased incrementally with the number of iterations. Hence, for interarrival times in long iterations, Automodeler periodically makes predictions at *special intervals*, in addition to those made at the end of an iteration. To compute these intervals, Automodeler heuristically divides the sum of the monitored average disk service time, the estimated context switching time, and a fraction (e.g. 10%) of the Linux OS time slice with the predicted interarrival times derived from the simple model. The idea is to estimate the number of blocks that could arrive while a disk is servicing requests, augmented by the overhead due to context switching between the application and the prefetch issuer thread. For the 32-node cluster, the interval is about 42 blocks. The 16-node and 8-node clusters have approximately 28 and 21 blocks respectively. Because the read interarrival times are larger in slower processors, the size of a special interval decreases as processor speed increases.

8.3 Performance on the 32-Node Cluster

8.3.1 Prefetching on Single Disk with Read-Write Caching

8.3.1.1 Execution Time Performance

Figure 8.1 compares the total execution times of Wavetoy executing with and without prefetching atop PPFS2 on the 32-node cluster, under various caching and striping strategies. The top curve

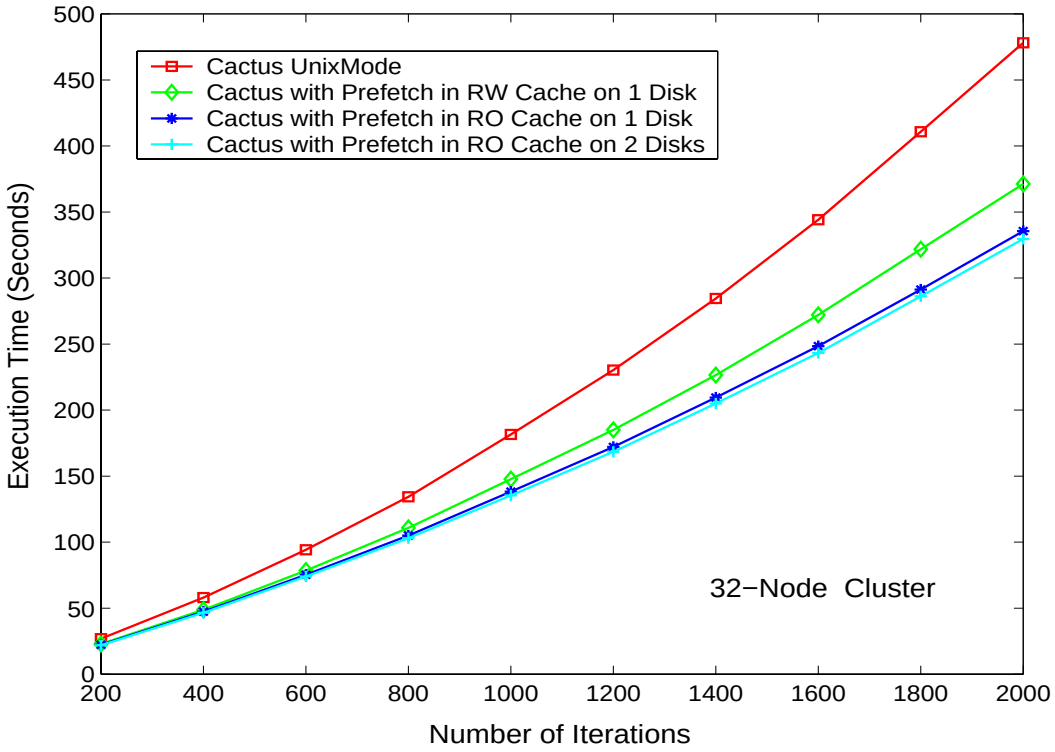

Figure 8.1: Wavetoy Execution Time under Different Configurations – 32-Node Cluster

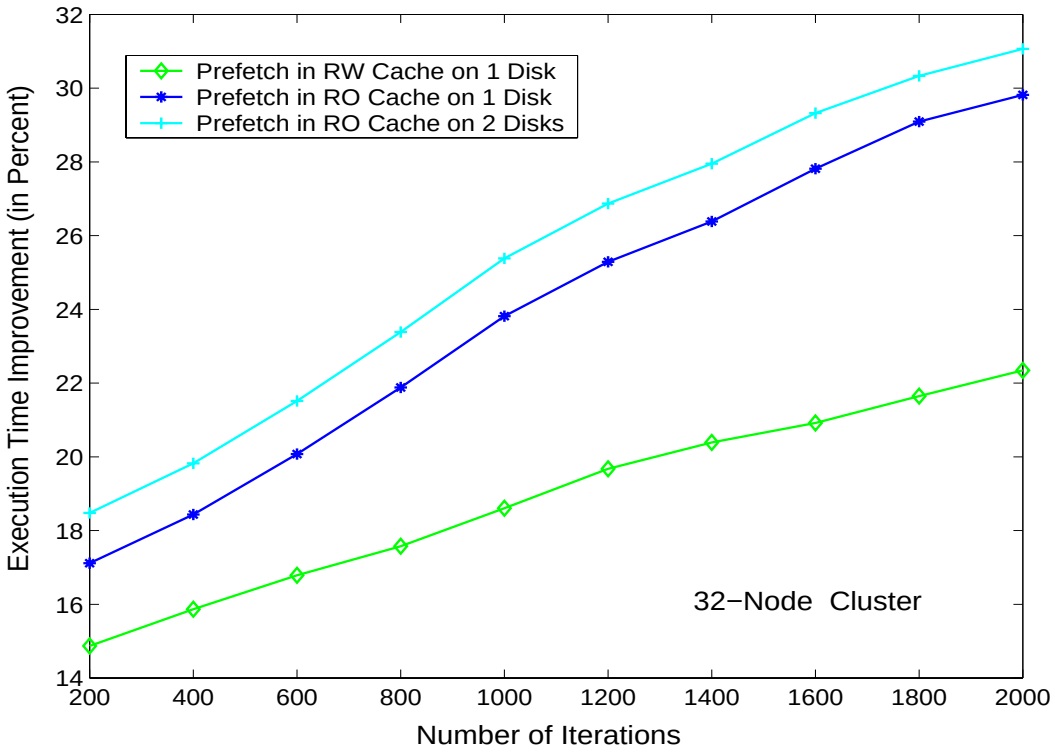

Figure 8.2: Wavetoy Execution Time Improvement – 32-Node Cluster

Iterations	200	400	600	800	1000	1200	1400	1600	1800	2000
Read Block Arrivals/second	902	1661	2297	2859	3285	3740	4106	4442	4703	4992
Execution Time (Rounded to the Nearest Second)										
Stand Alone	27	58	94	134	182	230	285	344	411	478
With Prefetch &										
RW Caching	23	49	78	111	148	185	226	272	322	371
RO Caching	22	47	75	105	138	172	209	248	291	335
RO Caching & 2-Disk Striping	22	46	74	103	135	168	205	243	286	330
Improvement (%)										
RW Caching	14.8	15.5	17.0	17.2	18.7	19.6	20.7	20.9	21.6	22.4
RO Caching	18.5	18.9	20.2	21.6	24.2	25.2	26.7	27.9	29.2	29.9
RO Caching & 2-Disk Striping	18.5	20.7	21.3	23.1	25.8	26.9	28.1	29.4	30.4	31.0

Table 8.2: Percentage Improvement in Wavetoy Execution Time – 32-Node Cluster

represents Wavetoy running alone, making Unix read calls to a local disk. The next curve depicts Wavetoy running with prefetching. All read and write blocks were cached in a 32 MB PPFS2 cache.

For a small workload of 200 iterations, performance improvement via prefetching was small. In this case, the initialization costs associated with modeling and other components in PPFS2 dominated. However, as the number of iterations grew, the costs were amortized and improvement in execution time became evident. For 2000 iterations, Wavetoy executed for 478 seconds in Linux stand-alone mode and 371 seconds in prefetching and read-write caching mode, resulting in an improvement of 22.4%.

Table 8.2 summarizes the percentage improvements for the 3 prefetching modes: read-write caching, read-only caching, and 2-disk striping with read-only caching. Read arrival rates increased from about 900 for 200 iterations to more than 4900 blocks/second for 2000 iterations – 5 times larger. Throughout this range, Wavetoy execution time improvement in read-write caching mode, varying from 14.8% to 22.4%, was shown to scale with increasing I/O intensity. These encouraging results can be attributed to reduction in I/O stalls, made possible by successful prefetches and the reuse of cache blocks fetched during previous iterations.

Iterations	200	400	600	800	1000
	1200	1400	1600	1800	2000
Demand Fetch Cache Misses For Reads (32 MB Cache Size)					
Without Prefetch	15912	78720	189119	347096	552703
	805938	1106778	1455210	1851282	2295013
With Prefetch & RW Caching	198	579	1071	1710	2527
	3367	4319	5394	6630	8027

Table 8.3: Cache Miss Comparison between Prefetch and without Prefetch – 32-Node Cluster

8.3.1.2 Cache Miss Performance

If prefetches are not late, block predictions are accurate, and the disk is not over-utilized, demand fetch requests will be served without I/O stalls. Otherwise, they will be delayed by cache misses. To evaluate caching performance improvement due to prefetching, Table 8.3 compares the number of demand fetch cache misses for read requests when Wavetoy was executing with and without the prefetcher.

Without prefetching, cache misses grew rapidly from approximately 16000 for 200 iterations to more than 2.2 million for 2000 iterations. Wavetoy issued approximately 104 million read requests, or about 2.3 million blocks of 4 KB for the 2000-iteration experiment.

However, with prefetching, these numbers reduced significantly as the number of iterations increased. They ranged from 198 for 200 iterations to slightly over 8000 for 2000 iterations. Overall, cache misses for read blocks were reduced by at least 2 orders of magnitude.

8.3.2 Prefetching on Single Disk with Read-Only Caching

At the end of each iteration, Wavetoy issued a long write request of 991844 bytes ($\approx$ 1 MB) to store new simulation results. Out of this megabyte, only 0.2% (around 2.3 kilobytes) were read in subsequent iterations. This imbalance between the amount of data written and reused wasted cache resources when both read and write requests were cached. One solution for better resource utilization would be to *cache only the read requests* for both prefetches and demand fetches.

In Figure 8.1, the third curve from the top depicts Wavetoy execution times when read only caching, with a 6400 KB cache size, was activated in PPFS2. For 2000 iterations, the execution time dropped from 371 (with read-write caching) to 335 seconds. This additional 7.5% improvement

Iterations	200	400	600	800	1000	1200	1400	1600	1800	2000
Number of Demand Fetch Cache Misses For Reads										
With Prefetch										
RW Caching	198	575	1071	1710	2527	3367	4319	5394	6630	8027
RO Caching	240	480	720	955	1191	1427	1667	1908	2147	2386
RO Caching & 2-Disk Striping	240	480	720	955	1191	1427	1667	1908	2147	2386

Table 8.4: Cache Miss Comparison between Read-Write and Read-Only Caching

was achieved with a much smaller memory footprint – 9600 KB versus 32 MB, about 3.5 times smaller than that used with full read-write caching. Here, the main cost savings was due to the elimination of expensive memory copy operations to store application output data in the cache. Despite current low memory prices, small memory requirements are still attractive because they make a prefetching system more scalable, allowing better resource sharing among multiple files within the same application or across different applications.

Compared with Linux stand-alone mode, prefetching with read-only caching yielded a gradual execution time improvement, from 18.5% for 200 iterations to 29.9% for 2000 iterations, as shown in Table 8.2.

Even with a much smaller cache size, the demand fetch cache misses for reads were also smaller than those obtained with read-write caching. Table 8.4 shows that, for 2000 iterations, the cache misses dropped from 8027 to 2386, around 68%.

8.3.3 Prefetching with Read-Only Caching and Disk Striping

Disk striping, a standard technique to distribute data across multiple disks, allows the overlapping of disk data transfers with concurrent disk accesses. Providing larger I/O bandwidths and lower I/O latencies can significantly improve the performance of both read and write accesses.

We investigated the impact of local disk striping on prefetching and caching. We chose local disks over network disks because bandwidths in the former can provide a much higher data rate, around $40-60$ MB/second on most disks in today's PCs. Taking into account the TCP/IP protocol stack overhead, even gigabit networks on LAN lag behind local disks in delivering the necessary bandwidths. For example, on the 16-node cluster, each port in the Myrinet interconnect has a 1.28Gb/second peak bandwidth. However, the link speed is only 230 Mb/second when accessing

the network through TCP/IP using the Myrinet's device driver to emulate an Ethernet network.

We used two local disks to stripe Wavetoy's data in a round-robin manner with a stripe depth of 1 MB to match the application's maximum write request size. Again, only the read requests were cached to achieve small memory requirements. With striping, we obtained an additional 1% improvement as illustrated by the top curve in Figure 8.2. For 2000 iterations, Wavetoy ran faster than Linux Vanilla by about 31% when prefetching, read-only caching, and striping were deployed concurrently.

The modest performance gain achieved through striping could be explained by the small read request sizes in Wavetoy. The maximum read data rate for the largest iteration (2000) was about 14 MB/second, well within the sustainable effective disk bandwidth of 40 to 60 MB/second – the disk bandwidth supply was larger than the demand. The perceived performance gain was due mostly to disk I/O overlapping rather than abundant disk bandwidths.

Adding more disks to stripe data did not improve performance further. With a small read data volume, the overhead associated with software striping across multiple disks outweighed the benefits and caused the application execution time to increase.

8.3.4 Cache Residency Measurements

To assess the performance of our prefetching system in utilizing the cache resource, we considered three metrics aimed at estimating the time interval a block stayed in the cache. These metrics included the mean time from prefetch to access, the mean time from prefetch to eviction, and the cache block hit ratio.

The *mean time from prefetch to access* is the average time interval between the instant a block is prefetched into the cache until it is first accessed by the application. It exposes the extent of prefetches' earliness. A large mean time strongly suggests premature prefetches, whereas a very small mean time indicates that prefetches are approaching being late. Table 8.5 shows results obtained for Wavetoy on the 32-Node cluster for 2000 iterations. We used the read-only prefetching strategy on two cache sizes, 3200 and 6400 KB. The smaller cache had a mean time from prefetch to access of about 1.36 milliseconds, or 40 times the mean block interarrival time (≈ 34 microseconds) during a read burst. Doubling the cache size from 3200 to 6400 KB increased the mean time from

prefetch to access to 1.71 milliseconds.

Once a given block is accessed by the application, it becomes a candidate for replacement. If later predictions include the block, it is moved to the non-evicted portion of the cache to anticipate upcoming requests. *The mean time from prefetch to eviction* measures the average time between the instant a block is prefetched until it is evicted for replacement. It gauges the impact of data reuse patterns and cache sizes on block replacements. Frequently accessed blocks tend to stay longer in the cache. Similarly, a large cache, allowing more blocks to be retained, incurs fewer cache misses, hence fewer block replacements. This behavior is evident in the results – the mean times from prefetch to eviction were 51.82 milliseconds and 73.02 milliseconds for the two cache sizes respectively. Our measurements would be larger if we also counted the blocks that got evicted at file closing time.

Finally, the third metric evaluates lateness in prefetching. In PPFS2 mispredictions and late arrivals are manifested as cache misses. Table 8.6 summarizes the cache block hits and misses that occurred in Cactus Wavetoy. For 2000 iterations, less than 0.2% of the total blocks requested were mispredicted or late.

8.3.5 Prefetching Using Only Markov Model Predictions

To investigate the performance of prefetching without temporal information assistance, we conducted experiments on Cactus Wavetoy using only predictions from the Markov models. To prevent late prefetches and subsequent I/O stalls, one simple strategy is to *prefetch a fixed number of blocks for every new requested block*. Our experiments were run for 2000 iterations, with the same experimental setup designed for the striping and read-only caching experiment described in the previous section.

Figure 8.3 plots the application execution time against the number of prefetched blocks, from

Cache Size (KB)	Mean Time from Prefetch to Access (milliseconds)	Mean Time from Prefetch to Eviction (milliseconds)
3200	1.36	51.82
6400	1.72	73.02

Table 8.5: Cache Residency Measurements for 2 Cache Sizes

Cache Size (KB)	Cache Block Hits	Cache Block Misses	Hit Ratio
3200	2383834	4427	99.81%
6400	2385447	2814	99.88%

Table 8.6: Cache Hits/Misses for 2 Cache Sizes

1 to 64, each time doubling the previous number $(1, 2, 4, 8, \ldots)$. The plot was generated using a logarithmic scale of base 2 on the x axis and a linear scale on the y axis. The smallest execution time was associated with a prefetch depth of 2 blocks. Increasing the depth beyond 8 caused the execution time to grow exponentially.

The execution time improvement (346 seconds) obtained by prefetching two Markov-predicted blocks for every new requested block was slightly lower ($\approx 4\%$) than that based on both ARIMA and Markov predictions (330 seconds). Prefetching via Markov models alone offers the advantage of eliminating the overhead associated with recording interarrival times and estimating ARIMA model parameters.

Unfortunately, there is no single, universal, fixed prefetch depth that can accommodate most applications. Changes in applications' I/O access patterns, request sizes, numbers of iterations, and system loads can have a significant impact on the choice of a prefetch depth. For verification, we experimented with a simple synthetic workload that read large requests of 1 MB for 1000 iterations. The workload was created from Wavetoy's data. It had similar incrementally periodic read behavior, but much longer requests. All other caching parameters remained the same. Figure 8.4 reveals that, unlike Wavetoy, the most suitable depth was around 8 blocks.

Our experiments with the Cactus code confirm that, without knowledge of temporal I/O behavior to guide prefetching, arbitrarily deciding when and how many data blocks to prefetch could severely degrade application I/O performance. As long as time series modeling overhead can be contained, the benefits realized using time series predictions can outweigh the costs.

8.4 Prefetching Performance on Other Clusters

In the remaining experiments, we used results previously obtained for the high-speed cluster as a baseline to compare prefetching performance on other slower clusters. The 930 MHz processors in

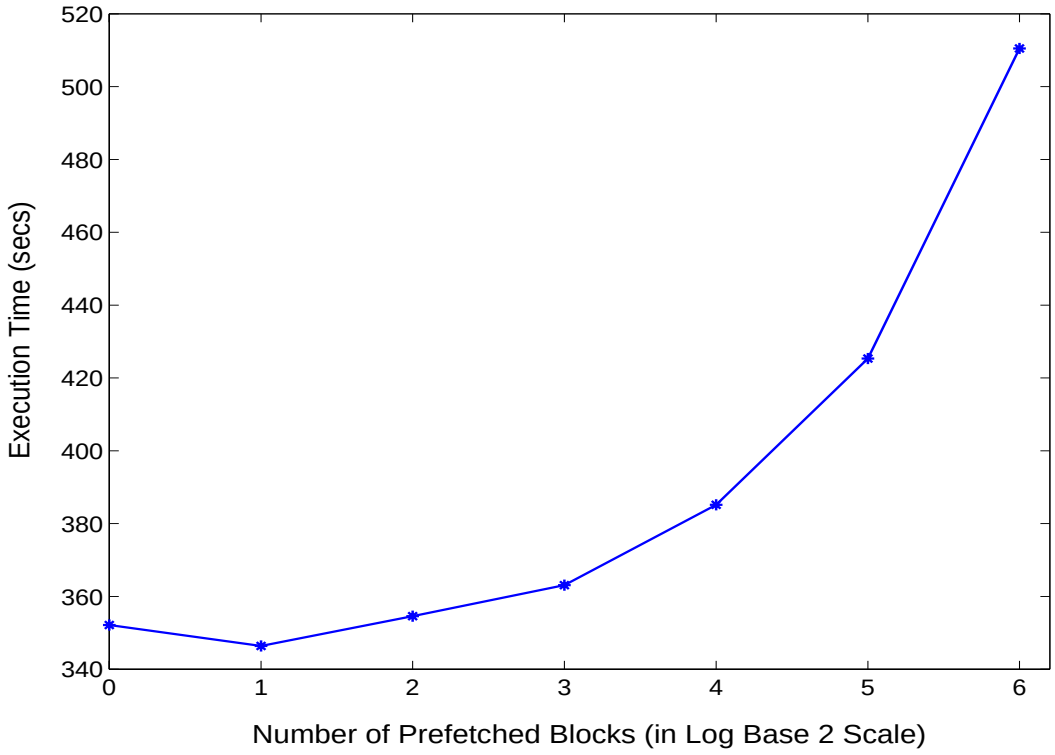

Figure 8.3: Wavetoy Execution Time Using Only Markov Predictions

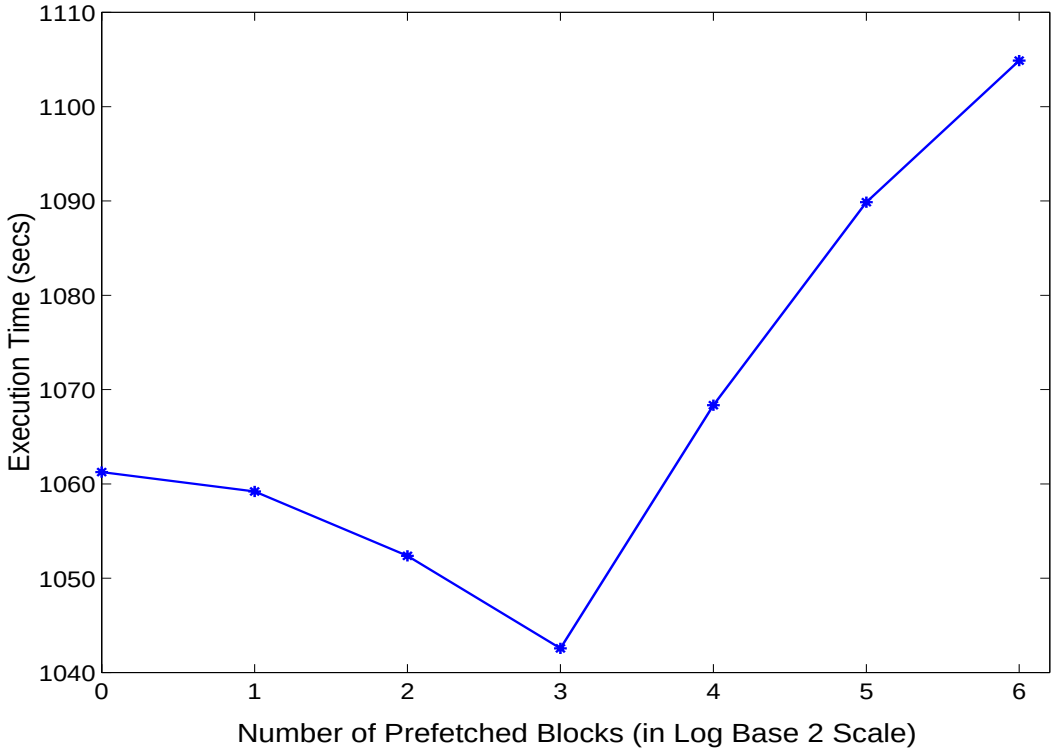

Figure 8.4: Synthetic Workload Execution Time Using Only Markov Predictions

the 32-node cluster are about twice as fast as the 16-node cluster, and 3.5 times faster than the 8-node cluster.

Overall, performance improvement was maintained for various processor speeds. Figures 8.2, 8.6, 8.8 and Tables 8.2, 8.7 and 8.8 show that, in all three clusters, prefetching with 2-disk striping and read-only caching achieved an aggregate improvement of around 30% over Linux stand-alone for 2000 iterations.

Improvements with read-write caching on the 8-node and 16-node clusters were about 20.8%, smaller than the 22.4% gain achieved on the 32-node cluster. With slower processors, the overhead of copying data from large write requests into the cache consumed a larger fraction of the execution time, yielding smaller percentage improvement.

Striping across 2 disks gave only 0.7% performance gain on the 16-node cluster. On the 8-node cluster with even slower processors and disks, the performance gain from striping was almost insignificant. Our experiments show that, on slow machines, software striping costs could offset the striping benefits, especially when read request sizes are small, but the number of reads is large.

Wavetoy's demand fetch cache misses were comparable across all three clusters when the same cache size was used. For 2000 iterations, cache misses incurred with read-only caching were approximately 3 times smaller than those with read-write caching.

8.5 Summary

In this chapter, we have experimentally validated our adaptive prefetching system using a scientific application. The ARIMA Automodeler, the Markov modeler, the prefetcher, along with caching and striping were concurrently deployed during application execution. We evaluated the application execution time improvement and reduction in demand fetch cache misses on 3 Linux clusters with different hardware characteristics. Overall, we achieved around 30% improvement in execution time. The number of cache misses also dropped significantly, by at least two orders of magnitude.

When only a small fraction of an application's output data is reused, caching the read requests only can be more beneficial – it reduces expensive memory copy operations and uses the cache resources more effectively. Software striping exercised on high-speed processors yields higher performance gain than when used with slow processors.

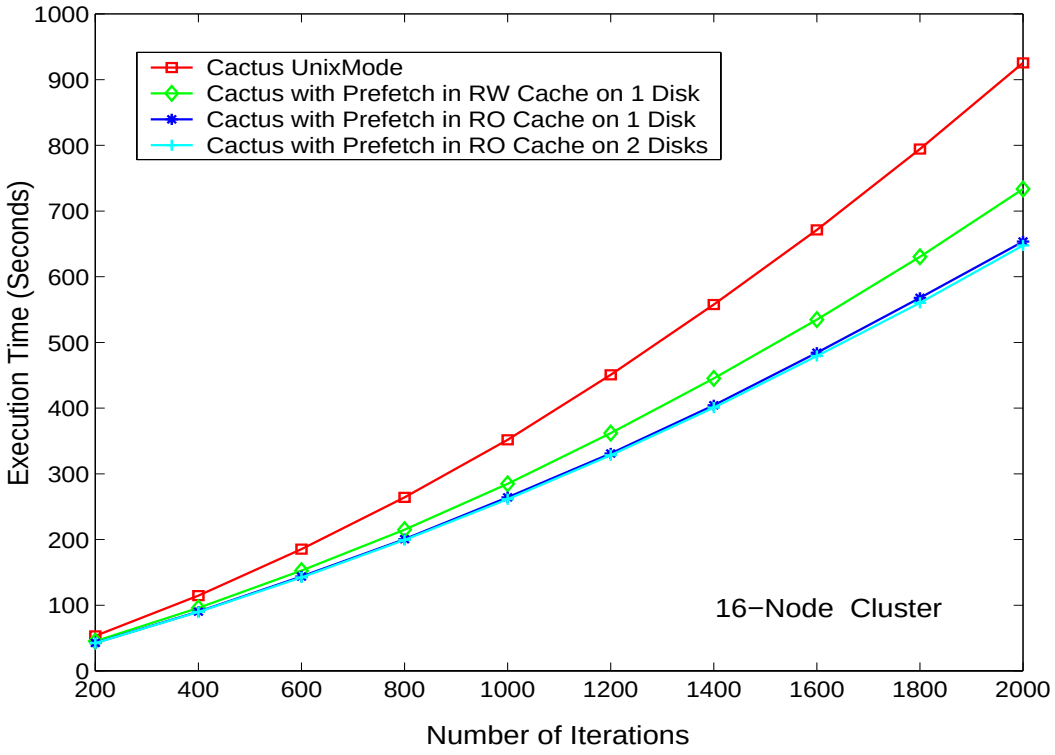

Figure 8.5: Wavetoy Execution Time under Different Configurations – 16-Node Cluster

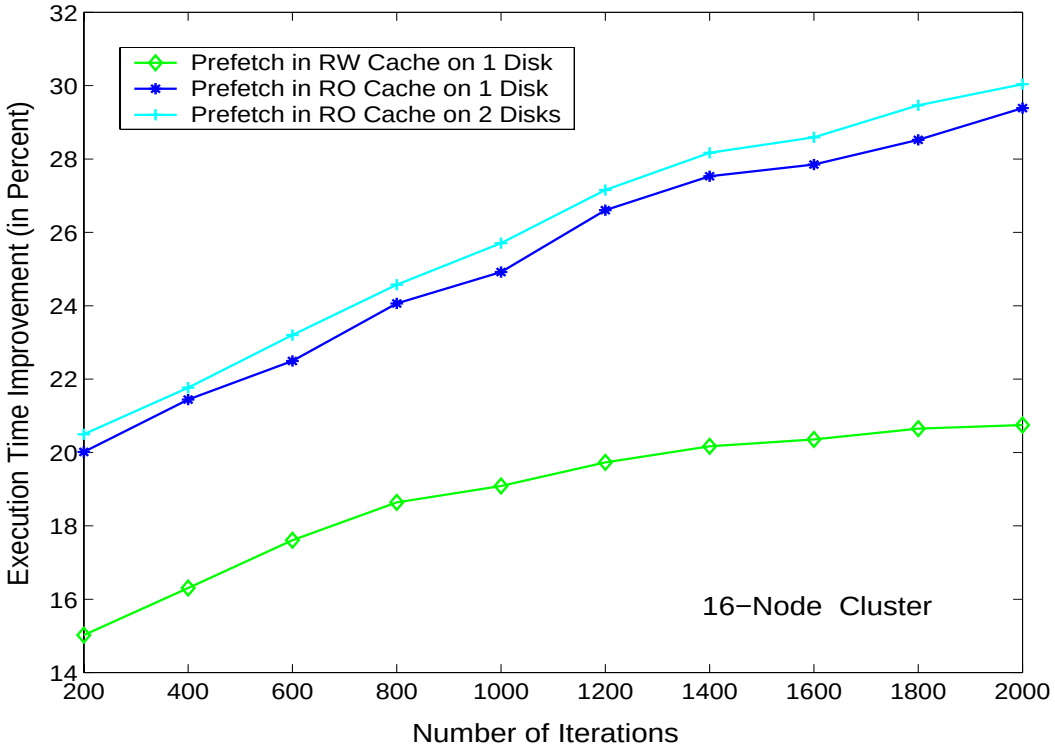

Figure 8.6: Wavetoy Execution Time Improvement – 16-Node Cluster

Iterations	200	400	600	800	1000	1200	1400	1600	1800	2000
Execution Time (Rounded to the Nearest Second)										
Stand Alone	53	115	185	264	352	451	558	671	794	925
With Prefetch &										
RW Caching	45	96	153	215	285	362	445	535	630	733
RO Caching	43	90	144	201	264	331	404	484	568	654
RO Caching & 2-Disk Striping	42	90	142	199	261	328	401	479	560	647
Improvement (%)										
RW Caching	15.1	16.5	17.3	18.6	19.0	19.7	20.2	20.2	20.7	20.8
RO Caching	18.9	21.7	22.1	23.9	25.0	26.6	25.6	27.9	28.5	29.3
RO Caching & 2-Disk Striping	20.8	21.8	23.2	24.6	25.9	27.3	28.1	28.6	29.5	30.0

Table 8.7: Percentage Improvement in Wavetoy Execution Time – 16-Node Cluster

Iterations	200	400	600	800	1000	1200	1400	1600	1800	2000
Execution Time (Rounded to the Nearest Second)										
Stand Alone	88	190	304	436	582	740	915	1102	1305	1524
With Prefetch &										
RW Caching	74	157	247	351	466	590	725	872	1031	1204
RO Caching	69	147	233	327	431	541	661	789	925	1072
RO Caching & 2-Disk Striping	69	147	231	326	430	540	660	787	923	1070
Improvement (%)										
RW Caching	15.9	17.4	18.8	19.5	19.9	20.3	20.8	20.9	21.0	21.0
RO Caching	21.6	22.6	23.3	25.0	25.9	26.9	27.8	28.4	29.1	29.7
RO Caching & 2-Disk Striping	21.6	22.6	24.0	25.2	26.1	27.0	27.9	28.6	29.3	29.8

Table 8.8: Percentage Improvement in Wavetoy Execution Time – 8-Node Cluster

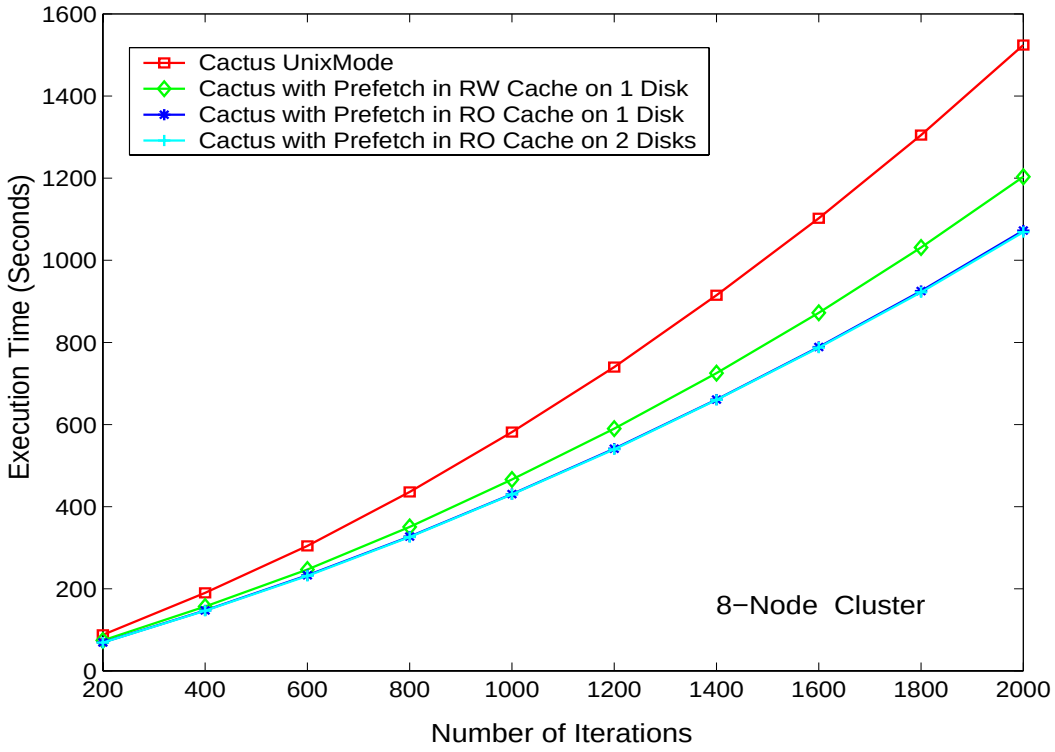

Figure 8.7: Wavetoy Execution Time under Different Configurations – 8-Node Cluster

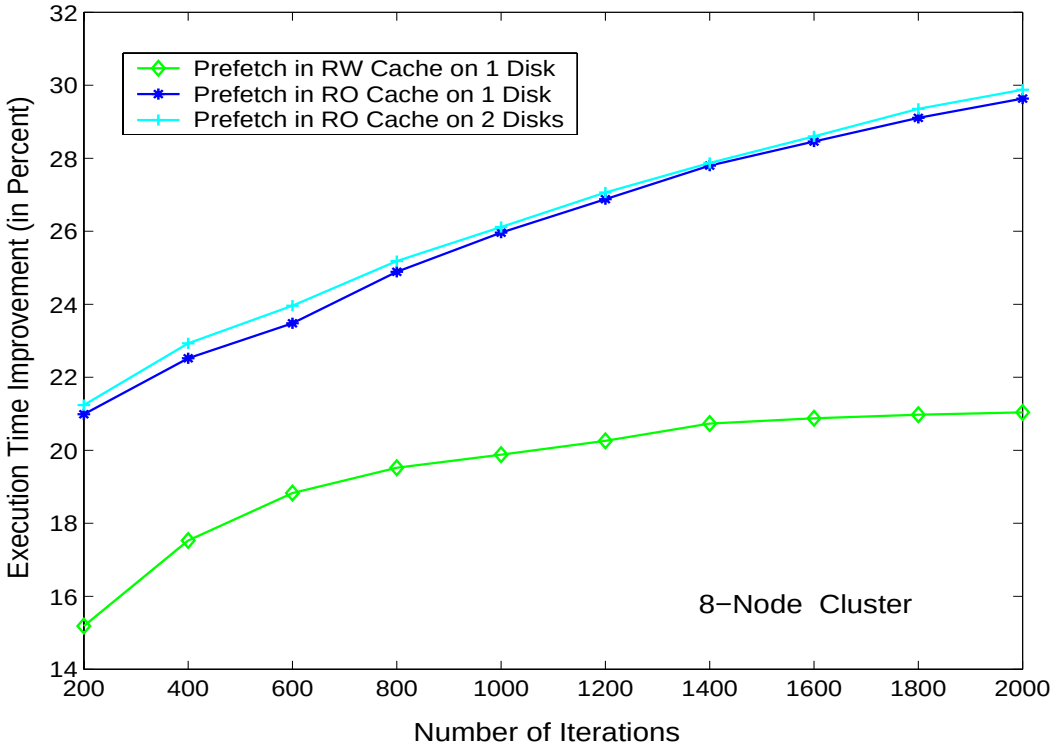

Figure 8.8: Wavetoy Execution Time Improvement – 8-Node Cluster

Iterations	200	400	600	800	1000	1200	1400	1600	1800	2000
Demand Fetch Cache Misses For Reads										
With Prefetch										
RW Caching	197	571	1068	1705	2518	3358	4307	5384	6624	8016
RO Caching	240	480	720	955	1191	1427	1667	1908	2147	2386
RO Caching & 2-Disk Striping	240	480	720	955	1191	1427	1667	1908	2147	2386

Table 8.9: Cache Miss Comparison – 16-Node Cluster

Iterations	200	400	600	800	1000	1200	1400	1600	1800	2000
Demand Fetch Cache Misses For Reads										
With Prefetch										
RW Caching	198	568	1071	1710	2525	3367	4312	5394	6630	8024
RO Caching	240	480	720	955	1191	1427	1667	1908	2147	2386
RO Caching & 2-Disk Striping	240	480	720	955	1191	1427	1667	1908	2147	2386

Table 8.10: Cache Miss Comparison – 8-Node Cluster

Chapter 9

Conclusions

In this thesis, our primary objective is improving the I/O performance of applications by adaptively prefetching future read requests to hide disk latency. We addressed the problem of predicting future requests from two perspectives – space and time. We built a prototype prefetching system that combined spatial and temporal predictions to adaptively determine when, what and how many data blocks to prefetch. We experimentally validated the time series modeling engine and the adaptive prefetcher on executions of scientific applications.

In the space domain, we reused Oly's Markov modeling system [41] to provide quantitative predictions for future file blocks. These state-based probabilistic models were created online from applications' read accesses monitored at run time. Predictions were generated on a greedy basis, selecting the most likely transition from each state to determine the next state.

In the time domain, we built a software framework for time series modeling and forecasting I/O arrival patterns. Our framework prototype Automodeler, has the ability to automatically identify, track, estimate, and predict stationary, non-stationary, and seasonal behavior in read interarrival times during application execution. Below, we summarize the design of Automodeler and its integration into our adaptive prefetching system.

9.1 Automodeler Design Summary

Automodeler consists of four major components:

- *An automatic model structure identifier* to automatically construct ARIMA models at run time. The identification process exploits the autocorrelation function (ACF) and partial

autocorrelation function (PACF), two fundamental tools to expose data patterns in time series. We used the Barlett equations to approximate a confidence interval for the ACF and PACF of a series. Automatic isolation of significant correlations is achieved by selecting those that exceed the confidence interval.

We devised various tests to automatically recognize patterns exposed by the significant correlations and identify stationary, non-stationary, and seasonal processes. The L-1 and L-2 distance tests help determine the non-stationarity and seasonality of a series in addition to its season length. The lag count and the average rate of change tests help differentiate between exponential decay and abrupt cutoff patterns. For simplification, these patterns are matched with those of the five most common ARIMA processes – AR(1), MA(1), AR(2), MA(2) and ARMA(1,1) – to approximate the number of elements in the autoregressive and moving average components.

To detect abrupt phase transitions that can occur during program execution, we applied the Haar wavelet transform directly to the read interarrival times. The detail coefficients from the transformation remove average information and expose high frequency signals. We used the statistical Jackknife procedure to compute a confidence interval for the mean of these coefficients, without requirement knowledge of its distribution. High frequency signals that exceed the confidence interval pinpoint the locations of abrupt transitions.

- *A real-time recursive parameter estimator for stationary processes.* To reflect changes intrinsic or external to applications that result from changes in underlying system dynamics, parameters of a given identified model are re-estimated for every new read block arrival. Due to its simplicity and efficiency, we chose the extended least squares (ELS) algorithm [2, 11, 52] to estimate parameters for stationary series.

  ELS exploits least squares to minimize the sum of the squared estimation errors, recursion to reduce processing costs, and noise bootstrapping to compute a model's moving average components. Through recursion, new parameters are estimated by simply making adjustments to the previous parameters instead of restarting the entire computation chain.

- *A recursive differencer/integrator for non-stationary and seasonal processes.* Differencing is

a transformation technique that stabilizes the means of non-stationary and seasonal series such that one can reuse ELS for parameter estimation. Regular differencing applies to non-stationary but non-seasonal series, whereas seasonal differencing applies to seasonal series. We designed a chained structure of recursive differencers/integrators (RDI) to manage the sequence of differencing transformations – results from regular differencing can be forwarded for seasonal differencing and vice versa until the series is stationary.

Integration reverses the differencing transformations to recover forecasts for the original non-stationary and non-seasonal series. This reversal task can be efficiently managed by simply traversing the RDI chain upwards to gradually re-integrate the previously taken differences.

- *A n-step ahead forecaster.* We implemented this forecaster to produce long-range forecasts, allowing multiple block prefetching. For stationary arrival processes, we bootstrapped from the one-step ahead forecasts for n time steps, each time using the new forecasts to derive results for the next time step. For non-stationary and seasonal processes, we make use of the RDI's integration capabilities to reverse the differences.

Experiments validating Automodeler showed that a) read interarrival times can be automatically modeled and predicted during program execution, b) one-season ahead forecasts produced for two I/O traces and one scientific application achieve more than 80% in overall accuracy.

9.2 Adaptive Prefetcher Design Summary

We integrated the Markov modeler [41], the ARIMA Automodeler, and the adaptive prefetcher atop PPFS2. The prefetcher uses predictions for future block numbers from the Markov modeler and predictions for future interarrival times from the ARIMA modeler to schedule prefetch requests. It builds prefetch schedules based on a variant of the Forestall algorithm [60, 44], comparing the supply and demand of disk bandwidth to determine the number of data blocks to be prefetched.

The task of issuing prefetch requests to disks is assumed by a separate thread structure, which examines the prefetch schedules. Prefetch requests can be directed to either a single or multiple disks, using disk striping to achieve deeper overlapping of computation with disk I/O.

Experiments evaluating the prefetcher on different hardware platforms indicate that our prefetching system scales with processor speeds, achieving 30% performance improvement over the traditional Unix file system on all platforms. This improvement results from a combination of several optimizations:

- When only a small fraction of data written is re-read, caching only the read requests instead of both read and write requests is more cost effective because of lower memory requirements and fewer memory copy operations.

- Selecting simple models over complex ones offers the advantage of reducing modeling overhead, especially in the stages of parameter estimation and forecasting, at the expense of accuracy. However, from an I/O prefetching perspective, forecasts of future interarrival times do not have to be very accurate as long as the general, temporal I/O behavior can be approximated. From this view point, we addressed the issue of prefetching in three domains – space, time, and approximation.

- If an application's read request sizes are small, striping data across several disks improves performance only marginally in slow uniprocessors, where the prefetcher thread, the striping routine, and the application compete for processing resources and cause context switching overhead. On high-speed multiprocessor machines, the performance gain is more noticeable as more CPU resources become available. Data striping can be substantially more beneficial for workloads characterized by large request sizes with high data volumes accessed at high rates.

Chapter 10

Future Work

Having described how online predictions created from ARIMA models can guide I/O prefetching, in this chapter, we will discuss the potential of automatic time series modeling for improving the performance of other types of applications. Our discussions focus on two aspects a) enhancements to our modeling framework, and b) extensions of the framework to accommodate a larger class of applications.

10.1 Model Identification Enhancement

In Automodeler, autocorrelation analysis plays a prominent role in identifying model structures for time series. Although it successfully found causal models for read interarrival times in all of our experiments, it is highly susceptible to outliers. Perturbations caused by outliers induce high variances which, in turn, distort and hide the base correlation patterns, making them difficult to recognize. Hence, models identified via ACF and PACF are often considered as preliminary models.

A more robust approach is minimizing the Akaike information criterion with bias correction (AICC) statistics [1, 27, 8]. It involves a systematic search of available models with increasing parameters. The most appropriate model will have parameters that maximize the likelihood function, giving the smallest AICC value. However, this computationally intensive search for the minimum AICC must be optimized to be useful for online applications.

10.2 Write-Behind

Write-behind is a standard cache management technique used to flush dirty blocks from a file cache. Normally, when disk blocks are brought into the cache either by demand fetch or prefetch, block evictions require replaced data be written to disks. Alternatively, one can use write-behind to release cache resources earlier, anticipating future block arrivals and minimizing block replacement times. When running with a large cache size, write-behind can also reduce an application's execution time as fewer dirty blocks are left to be flushed at file closing.

One common write-behind strategy involves triggering some lazy-write worker threads to flush data periodically. However, when the number of dirty blocks is large, this periodic operation can intensify I/O burstiness and cause I/O bottlenecks. Instead, one can use forecasts for write interarrival times (during which processors are busy with computations) to schedule write-behinds, smoothing disk traffic and overlapping computations with disk I/O.

10.3 Transformations other than Differencing

This thesis is limited to investigating differencing transformations, techniques used to stabilize the means of those non-stationary series that do not vary about a *fixed mean*, but have a *constant variance*. This restriction precludes modeling a wide range of useful applications that have *non-constant variances* – the deviations of a series' data from its mean change through time. Variance-stabilizing transformations are needed to model these series [7].

The Box-Cox power transformations [6] define a large class of non-linear transformations aimed at stabilizing variances in time series. Data points $y(t)$ in a series are raised to some power λ which can be any non-zero real number. For example, $\lambda = -1$ corresponds to the reciprocal transformation (i.e., $1/y(t)$), whereas $\lambda = 1/2$ gives the square root transformation (i.e., $y(t)^{1/2}$).

Such fractional transformations are important because they can model the self-similar, fractal-like behavior found in many applications [35, 62, 21], where patterns look similar at all time scales, from coarse to fine resolutions. For example, patterns extracted from a one-second sample are similar to those extracted from a one-minute sample. Unlike the types of time series we studied, where autocorrelations decay exponentially (see Chapter 4), self-similar processes have

non-degenerate correlation structures where autocorrelations decay hyperbolically. They can be readily modeled by fractional ARIMA processes (p, d, q), with $d = \lambda$ to indicate a desired power level for the transformations [35]. Below, we briefly describe two types of applications that exhibit self-similar properties.

- Leland et al. reported in [35] that ethernet LAN traffic is self-similar and cannot be modeled by Poisson processes. When network traffic was analyzed at the network level (as opposed to individual user level), they found that the aggregate traffic from several active sources was bursty. The degree of burstiness intensified as the number of sources increased. Their results were obtained via offline analyses of trace data and visual inspection of graphical plots.

  We believe that *automatic identification and modeling of self-similar patterns* can facilitate and speed up the modeling process for a wide variety of emerging high-speed networks. Modeling results provide insight on the statistical distributions and characteristics of network traffic. This insight, in turn, can be used for performance evaluation and the tuning of network subsystems, projecting future capacity requirements, and influencing the design of next-generation network systems and protocols.

- In this thesis, we explored ARIMA time series to model temporal I/O behaviors at the *application level*. We believe that fractional ARIMA time series can be used to investigate I/O behaviors at the *system level*. In his trace-based analyses on file system usage in Windows NT, Vogels observed that, for various time scales, the arrival rates of file requests were bursty and had large, non-constant variances [62]. Similarly, the read data volumes per file open session also showed large variances. As a result, designs for future file systems will need to take into consideration the self-similar behavior of various file system parameters.

10.4 System Resource Predictions

Forecasts from time series analysis can also be used to anticipate the utilization and availability of various system resources – processor, memory, disk, and network bandwidth. Online predictions, created from online models, are particularly useful for scheduling and controlling tasks in real-time systems. In these feedback systems, such predictions are crucial to adapting to underlying

system dynamics and responding to changes in resource supply and demand. For example, to avoid performance degradation and maintain a guaranteed level of service, the admission control module of a multimedia file server will not admit additional file request streams when resource usages are predicted to exceed supply.

References

[1] Akaike, H. Information Theory and an Extension of the Maximum Likelihood Principle. In *Second International Symposium on Information Theory* (1973).

[2] Astrom, K. J., and Wittenmark, B. *Adaptive Control.* Addison-Wesley, 1989.

[3] Barlett, M. S. On the Theoretical Specification of Sampling Properties of Autocorrelated Time Series. In *Journal of the Royal Statistical Society* (1946), vol. B8.

[4] Belady, L. A. A Study of Replacement Algorithms for Virtual Storage Computers. In *IBM Systems Journal* (1966).

[5] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W. K. Myrinet: A Gibabit Per Second Local Area Network. In *IEEE Micro* (Feb 1995), vol. 15, pp. 29–36.

[6] Box, G., and Cox, D. An Analysis of Transformations. In *Journal of the Royal Statistics Society* (1964), vol. 26, pp. 211–252.

[7] Box, G. E., and Jenkins, G. M. *Time Series Analysis Forecasting and Control.* 2nd ed. San Francisco: Holden-day, 1976.

[8] Brockwell, P. J., and Davis, R. A. *Introduction to Time Series and Forecasting.* Springer-Verlag, 1996.

[9] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. A Study of Integrated Prefetcing and Caching Strategies. In *Proceedings of the ACM Sigmetrics* (May 1995), pp. 188–197.

[10] Chang, F., and Gibson, G. A. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Feb. 1999).

[11] Chen, H.-F., and Guo, L. *Identification and Stochastic Adaptive Control.* Birkhauser, Boston, 1991.

[12] Chen, T. F., and Baer, J.-L. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (1994), pp. 223–232.

[13] Crandall, P. E., Aydt, R. A., Chien, A. A., and Reed, D. A. Characterization of a Suite of Input/Output Intensive Applications. In *Proceedings of Supercomputing 95* (Dec. 1995).

[14] Dahlgren, F., Dubois, M., and Stenstrom, P. Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing* (1993), vol. I.

[15] Dahlgren, F., and Stenstrom, P. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems* (Apr. 1996), vol. 7, No. 4.

[16] Davies, R. *Newmat09: A C++ Matrix Library.* Available at http://webnz.com/robert/cpp_lib.htm, 2000.

[17] Dinda, P. A., and O'Hallaron, D. R. An Extensible Toolkit for Resource Prediction in Distributed Systems. In *Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, Pittsburg* (Jul. 1999).

[18] DOE, U. *United States Department of Energy Accelerated Strategic Computing Initiative ASCI.* Available at http://www.llnl.gov/asci, Jan. 1997.

[19] Durbin, J. The Fitting of Time Series Models. In *Review of the International Institute of Statistics* (1960), vol. 28.

[20] Fu, J., and Patel, J. H. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th International Symposium on Computer Architecture* (1991).

[21] Gomez, M. E., and Santonja, V. Analysis of Self-Similarity in I/O Workload Using Structural Modeling. In *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (1999), pp. 234–242.

[22] Griffioen, J., and Appleton, R. Reducing File Latencies Using a Predictive Approach. In *USENIX Technical Conference* (Summer 1994).

[23] Griffioen, J., and Appleton, R. Performance Measurements of Automatic Prefetching. In *Proceedings of the International Conference on Parallel and Distributed Computer Systems* (Sept. 1995).

[24] Groschwitz, N. C., and Polyzos, G. C. A Time Series Model of Long-term NSFNET Backbone Traffic. In *Proceedings of the IEEE International Conference on Communications* (May 1994), vol. 3, pp. 1400–4.

[25] Hagersten, E. *Towards Scalable Cache Only Memory Architectures.* PhD thesis, SICS Dissertation Series 8, Swedish Institute of Computer Science, Oct. 1992.

[26] Henderson, R. D., and Karniadakis, G. E. Unstructured Spectral Element Methods for Simulation of Turbulent Flows. In *Journal of Computational Physics 122* (1995), vol. 2, pp. 191–217.

[27] Hurvich, C. M., and Tsai, C. L. Regression and Time Series Model Selection in Small Samples. In *Biometrika*, vol. 76, pp. 297–307.

[28] Jiang, Z., and Kleinrock, L. An Adaptive Network Prefetch Scheme. In *IEEE Journal on Selected Areas in Communications* (Apr. 1998), vol. 16, No. 3.

[29] Joseph, D., and Grunwald, D. Prefetching Using Markov Predictors. In *IEEE Transactions on Computers* (Feb. 1999), vol. 48, No. 2.

[30] Jouppi, N. P. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associate Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium Computer Architecture* (May 1990).

[31] Kimbrel, T., Tomkins, A., Patterson, R., Bershad, B., Cao, P., Felten, E., Gibson, G., Karlin, A., and Li, K. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation, Seattle, WA.* (Oct. 1996), pp. 19–34.

[32] Kotz, D., and Ellis, C. S. Caching and Writeback Policies in Parallel File Systems. In *IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press* (Dec. 1991), pp. 60–67.

[33] Kotz, D., and Ellis, C. S. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems* (Miami Beach, Florida, 1991).

[34] Kumar, P. R. *Identification of Linear Systems - ECE 417 Lecture Notes.* University of Illinois at Urbana-Champaign, Fall 1999.

[35] Leland, W. E., Taqqu, M. S., Willinger, W., and Wilson, D. V. On the Self-Similar Nature of Ethernet Traffic (Extended Version). In *Proceedings of IEEE/ACM Transactions on Networking* (Feb. 1994), vol. 2, No. 1.

[36] Ljung, L., and Soderstrom, T. *Theory and Practice of Recursive Identification.* Massachusetts Institute of Technology Press, Cambridge, 1983.

[37] Madhyastha, T. M., and Reed, D. A. Intelligent, Adaptive File System Policy Selection. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation* (Oct. 1996), IEEE Computer Society Press, pp. 172–179.

[38] Miller, E. L., and Katz, R. H. Input Output Behaviors of Super Computing Applications. In *Proceedings of SuperComputing '91* (Nov. 1991), pp. 567–576.

[39] Miller, R. B., and Wichern, D. W. *Intermediate Business Statistics, Analysis of Variance, Regression and Time Series.* Holt, Rinehart and Winston, 1977.

[40] Mowry, T., Demke, A., and Krieger, O. Automatic Compiler Inserted I/O Prefetching for Out of Core Applications. In *Proceedings of the 2nd OSDI* (Oct. 1996).

[41] Oly, J. P. Markov Model Prediction of I/O Requests for Scientific Applications. In *Master Thesis, Department of Computer Science, University of Illinois at Urbana Champaign* (Spring 2000).

[42] Pankratz, A. *Forecasting With Univariate Box-Jenkins Models, Concepts and Cases.* John Wiley and Sons, 1983.

[43] Patterson, D. A., and Hennessy, J. L. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, 2nd Edition.

[44] Patterson, H., Gibson, G., Ginting, E., Stodolsky, D., and Zelenka, J. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Copper Mountain, CO.* (Dec. 1995), pp. 79–95.

[45] Paxson, V., and Floyd, S. Wide Area Traffic: The Failure of Poisson Modeling. In *Proceedings of IEEE/ACM Transactions on Networking* (Jun. 1995), vol. 3.

[46] Reed, D. A., and et al. *Pablo Scalable Performance Tools.* Pablo Research Group Homepage, Department of Computer Science, University of Illinois at Urbana-Champaign. Available at http://www-pablo.cs.uiuc.edu, 2001.

[47] Ribler, R. L., Vetter, J. S., Simitci, H., and Reed, D. A. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Computing* (Jul. 1998).

[48] Ruemmler, C., and Wilkes, J. An Introduction to Disk Drive Modeling. In *IEEE Computer* (Mar. 1994), pp. 17–28.

[49] Seidel, E., and et al. *The Cactus Code.* NCSA and Max Planck Institute for Gravitational Physics. Available at http://www.cactuscode.org, 2000.

[50] Shalf, J. *IEEEIO.* NCSA, University of Illinois at Urbana Champaign. Available at http://zeus.ncsa.uiuc.edu/~jshalf/FlexIO/IEEEIO.html, 2000.

[51] Simitci, H., Reed, D. A., Fox, R., Medina, M., Oly, J., Tran, N., and Wang, G. A Framework for Adaptive Storage Input/Output on Computational Grids. In *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP)* (Apr. 1999).

[52] Sin, K. S., and Goodwin, G. C. Stochastic Adaptive Control Using Modified Least Squares Algorithms. In *Automatica* (1982), vol. 18, No. 3, pp. 315–321.

[53] Smirni, E., and Reed, D. A. Workload Characterization of Input Output Intensive Parallel Applications. In *Proceedings of the Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Springer-Verlag Lecture Notes in Computer Science* (Jun. 1997), vol. 1245, pp. 169–180.

[54] Smith, A. J. Sequential Program Prefetching in Memory Hierarchies. In *IEEE Computer* (Dec. 1978), vol. 11.

[55] Smith, A. J. Cache Memories. In *ACM Computing Surveys* (Sept. 1982), vol. 14, pp. 473–530.

[56] Stollnitz, E. J., Derose, T. D., and Salesin, D. H. *Wavelets for Computer Graphics Theory and Applications.* Morgan Kaufmann, 1996.

[57] Strang, G., and Nguyen, T. *Wavelets and Filter Banks.* Wellesley - Cambridge Press, 1997.

[58] Sweldens, W., and Schroder, P. Building Your Own Wavelets at Home. In *Wavelets in Computer Graphics.* ACM SIGGRAPH Course Notes, 1996, pp. 15–87.

[59] The HDF5 Project. *HDF5 - A New Generation of HDF.* NCSA. University of Illinois at Urbana Champaign. Available at http://hdf.ncsa.uiuc.edu/HDF5, 2000.

[60] Tomkins, A., Patterson, R., and Gibson, G. Informed Multi-Process Prefetching and Caching. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics), Seattle, WA.* (Jun. 1997).

[61] Vitter, J. S., and Krishnan, P. Optimal Prefetching via Data Compression. In *Proceedings of the 1993 SIGMOD* (May 1993), pp. 257–266.

[62] Vogels, W. File System Usage in Windows NT 4.0. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)* (Dec. 1999), pp. 93–109.

[63] Wei, W. S. *Time Series Analysis, Univariate and Multivariate Methods.* Addison-Wesley, 1990.

[64] Wiel, S. P. V., and Lilja, D. J. When Caches Aren't Enough: Data Prefetching Techniques. In *IEEE Computer* (Jul. 1997), vol. 30, Issue 7.

[65] Willinger, W., Taqqu, M. S., Sherman, R., and Wilson, D. V. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet Lan Traffic at the Source Level. In *Proceedings of IEEE Compcon* (Spring 1989), pp. 112–117.

[66] Winstead, C., and McCoy, V. Studies of Electron-Molecule Collisions on Massively Parallel Computers. In *Modern Electronic Structure Theory, D. R. Yarkony, Ed., World Scientific* (1994), vol. 2.

[67] Wonnacott, R. J., and Wonnacott, T. H. *Introductory Statistics.* John Wiley and Sons, 1990.

[68] Wonnacott, T. H., and Wonnacott, R. J. *Regression, A Second Course in Statistics.* John Wiley and Sons, 1981.

[69] Zucker, D. F., Lee, R. B., and Flynn, M. J. Hardware and Software Prefetching Techniques for MPEG Benchmarks. In *IEEE Transactions on Circuits and Systems for Video Technology* (Aug. 2000), vol. 10, No. 5.

Vita

Nancy Ngoc Tran worked in sunny southern California at Honda R&D and McDonnell Douglas Corporation for a couple of years. Having had her share of system development work, she decided to pursue higher education at the University of Illinois at Urbana-Champaign in 1993.

Throughout her graduate studies, her research focused on adaptation in dynamic systems. In 1997, she completed her master thesis on active adaptation by program delegation in video on demand. She built a prototype to demonstrate the encapsulation of control intelligence in mobile programs and the use of such programs to regulate the flows of video streams in time-varying environments.

In 1998, she joined the Pablo group to work on adaptive file systems under the direction of Professor Daniel A. Reed. Fascinated by applied statistics and adaptive control engineering, she investigated the automation of time series analysis to construct real-time models of input/output behaviors. She used time series model predictions to improve scientific applications' I/O performance via adaptive, just-in-time prefetching. She earned her Ph.D. in Computer Science in December 2001.

1. N. Tran and D. A. Reed, "ARIMA Time Series Modeling and Forecasting For Adaptive I/O Prefetching". In *Proceedings of the 15th ACM International Conference on Supercomputing*, June 2001.

2. H.Simitci, D. A. Reed, R. Fox, M. Medina, J. Oly, N. Tran, and G. Wang, "A Framework for Adaptive Storage Input/Output on Computing Grids". In *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP)*, April 1999.

3. N. Tran and K. Nahrstedt, "Active Adaptation by Program Delegation in Video On Demand". In *Proceedings of the IEEE Conference for Multimedia*, June 1998.